Владимир Дронов

PRO
ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Django 3.0
Практика создания веб-сайтов на Python

BBCode-теги
CAPTCHA
Аутентификация через социальные сети
REST
Bootstrap
Angular
PostgreSQL
Memcached
ASGI + Uvicorn
Redis

Материалы на www.bhv.ru

bhv®

Владимир Дронов

Django 3.0. Практика создания веб-сайтов на Python

Санкт-Петербург
«БХВ-Петербург»
2021

УДК 004.738.5+004.438Python
ББК 32.973.26-018.1
Д75

Дронов В. А.

Д75 Django 3.0. Практика создания веб-сайтов на Python. — СПб.: БХВ-Петербург, 2021. — 704 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-6691-9

Книга посвящена созданию веб-сайтов на языке Python с использованием веб-фреймворка Django 3.0. Рассмотрены новинки Django 3.0 и дано наиболее полное описание его инструментов: моделей, контроллеров, шаблонов, средств обработки пользовательского ввода, включая выгруженные файлы, разграничения доступа, посредников, сигналов, инструментов для отправки электронной почты, кэширования и пр. Рассмотрены дополнительные библиотеки, производящие обработку BBCode-тегов, CAPTCHA, вывод графических миниатюр, аутентификацию через социальные сети (в частности, "ВКонтакте"), интеграцию с Bootstrap. Рассказано о программировании веб-служб REST, использовании и настройке административного веб-сайта Django, публикации сайтов с помощью веб-сервера Uvicorn, работе с базами данных PostgreSQL, кэшировании сайтов с помощью Memcached и Redi. Подробно описано создание полнофункционального веб-сайта — электронной доски объявлений, веб-службы, работающей в его составе, и тестового фронтенда для нее, написанного на Angular.

Электронное приложение-архив на сайте издательства содержит коды всех примеров.

Для веб-программистов

УДК 004.738.5+004.438Python
ББК 32.973.26-018.1

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Сависте*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн серии	*Марины Дамбиевой*
Оформление обложки	*Карины Соловьевой*

Подписано в печать 31.07.20.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 56,76.
Тираж 1500 экз. Заказ № 6169.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано в ОАО «Можайский полиграфический комбинат».
143200, Россия, г. Можайск, ул. Мира, 93.
www.oaompk.ru, тел.: (495) 745-84-28, (49638) 20-685

ISBN 978-5-9775-6691-9

Оглавление

Введение

Django — популярнейший в мире веб-фреймворк, написанный на языке Python, и один из наиболее распространенных веб-фреймворков в мире. Появившись в 2005 году — именно тогда вышла его первая версия, — он до сих пор остается "на коне".

Что такое веб-фреймворк?

Фреймворк (от англ. framework — каркас) — это программная библиотека, реализующая бóльшую часть типовой функциональности разрабатываемого продукта. Своего рода каркас, на который разработчик конкретного продукта "навешивает" свои узлы, механизмы и детали декора.

Веб-фреймворк — это фреймворк для программирования веб-сайтов. Как правило, он обеспечивает следующую типовую функциональность:

- взаимодействие с базой данных — посредством единых инструментов, независимых от конкретной СУБД;
- обработка клиентских запросов — в частности, определение, какая страница запрашивается;
- генерирование запрашиваемых веб-страниц на основе шаблонов;
- разграничение доступа — допуск к закрытым страницам только зарегистрированных пользователей и только после выполнения ими входа;
- обработка данных, занесенных посетителями в веб-формы, — в частности, проверка их на корректность;
- получение и сохранение файлов, выгруженных пользователями;
- рассылка электронных писем;
- кэширование сгенерированных страниц на стороне сервера — для повышения производительности.

Почему Django?

- Django — это современные стандарты веб-разработки: схема "модель-представление-контроллер", использование миграций для внесения изменений в базу данных и принцип "написанное однажды применяется везде" (или, другими словами, "не повторяйся").
- Django — это полнофункциональный фреймворк. Для написания типичного сайта достаточно его одного. Никаких дополнительных библиотек, необходимых, чтобы наше веб-творение хотя бы заработало, ставить не придется.
- Django — это высокоуровневый фреймворк. Типовые задачи, наподобие соединения с базой данных, обработки данных, полученных от пользователя, сохранения выгруженных пользователем файлов, он выполняет самостоятельно. А еще он предоставляет полнофункциональную подсистему разграничения доступа и исключительно мощный и удобно настраиваемый административный веб-сайт, которые, в случае применения любого другого фреймворка, нам пришлось бы писать самостоятельно.
- Django — это удобство разработки. Легкий и быстрый отладочный веб-сервер, развитый механизм миграций, уже упомянутый административный веб-сайт — все это существенно упрощает программирование.
- Django — это дополнительные библиотеки. Нужен вывод графических миниатюр? Требуется обеспечить аутентификацию посредством социальных сетей? Необходима поддержка CAPTCHA? На диске копятся "мусорные" файлы? Ставьте соответствующую библиотеку — и дело в шляпе!
- Django — это Python. Исключительно мощный и, вероятно, самый лаконичный язык из всех, что применяются в промышленном программировании.

Эта книга посвящена Django. Она описывает его наиболее важные и часто применяемые на практике функциональные возможности, ряд низкоуровневых инструментов, которые также могут пригодиться во многих случаях, и некоторые дополнительные библиотеки. А в конце, в качестве практического упражнения, рассказывает о разработке полнофункционального сайта электронной доски объявлений.

> ***Внимание!***
>
> Автор предполагает, что читатели этой книги знакомы с языками HTML, CSS, JavaScript, Python, принципами работы СУБД и имеют базовые навыки в веб-разработке. В книге все это описываться не будет.

> ***Электронное приложение***
>
> Содержит программный код сайта электронной доски объявлений и доступно на FTP-сервере издательства "БХВ-Петербург" по ссылке **ftp://ftp.bhv.ru/9785977566919.zip** и на странице книги на сайте **www.bhv.ru** (см. *приложение*).

Что нового в Django 3.0 и новой книге?

С момента написания автором предыдущей книги, посвященной Django 2.1, вышли версии 2.2 и 3.0 этого фреймворка. В них появились следующие нововведения:

- поле модели `SmallAutoField` — описано в *разд. 4.2.2*;
- расширенные средства для создания полей со списком — описаны в *разд. 4.2.3*;
- расширенные средства для описания индексов (в параметре `indexes` модели) — в *разд. 4.4*;
- условия модели (параметр `constraints`) — в *разд. 4.4*;
- метод `bulk_update()` модели — в *разд. 6.8*;
- новые функции СУБД — в *разд. 7.6.2*;
- новые средства для извлечения заголовков запросов (атрибут `headers` класса `HttpRequest`) — в *разд. 9.4*;
- метод `setup()` контроллеров-классов — в *разд. 10.2.1*;
- метод `get_user_permissions()` класса `User` — в *разд. 15.5.1*;
- метод `with_perms()` диспетчера записей модели `User` — в *разд. 15.5.2*;
- новые средства для работы со связями "многие-со-многими" — в *разд. 16.2*;
- атрибут `ordering_widget` классов наборов форм — в *разд. 17.3.3*;
- параметр безопасности `SECURE_REFERRER_POLICY` — в *разд. 30.1.5*;
- поддержка интерфейса ASGI для взаимодействия с веб-сервером — в *разд. 30.2.1*.

В новое издание книги по Django добавлен следующий материал:

- программное разрешение интернет-адресов — в *разд. 9.10*;
- работа с СУБД PostgreSQL — в *разд. 18.1*;
- библиотека django-localflavor — в *разд. 18.2*;
- кэширование посредством Memcached — в *разд. 26.1.1.3*;
- кэширование посредством Redis — в *разд. 26.2*;
- управление кэшированием на стороне клиента в посредниках — в *разд. 26.3.3*;
- публикация Django-сайта посредством веб-сервера Uvicorn — в *разд. 30.2.1*.

Использованные программные продукты

Автор применял в работе над книгой следующие версии ПО:

- Microsoft Windows 10, русская 64-разрядная редакция со всеми установленными обновлениями;
- Python — 3.8.1 (для разработки) и 3.7.6 (для публикации), в обоих случаях — 64-разрядные редакции;
- Django — 3.0.2;

- Django Simple Captcha — 0.5.12;
- django-precise-bbcode — 1.2.12;
- django-localflavor — 2.2;
- django-bootstrap4 — 1.1.1;
- Pillow — 7.0.0;
- django-cleanup — 4.0.0;
- easy-thumbnails — 2.7;
- Python Social Auth — 3.1.0;
- django-redis — 4.11.0;
- Django REST framework — 3.11.0;
- django-cors-headers — 3.2.1;
- Uvicorn — 0.11.2;
- mod-wsgi — 4.7.0, 64-разрядная редакция;
- Apache HTTP Server — 2.4.41VC15, 64-разрядная редакция;
- Node.js — 13.7.0, 64-разрядная редакция;
- Angular — 8.2.14.

Типографские соглашения

В книге будут часто приводиться форматы написания различных языковых конструкций, применяемых в Python и Django. В них использованы особые типографские соглашения, перечисленные далее.

- В угловые скобки (<>) заключаются наименования различных значений, которые дополнительно выделяются курсивом. В реальный код, разумеется, должны быть подставлены конкретные значения. Например:

  ```
  django-admin startproject <имя проекта>
  ```

  Здесь вместо подстроки *имя проекта* должно быть подставлено реальное имя проекта.

- В квадратные скобки ([]) заключаются необязательные фрагменты кода. Например:

  ```
  django-admin startproject <имя проекта> [<путь к папке проекта>]
  ```

  Здесь *путь к папке проекта* может указываться, а может и отсутствовать.

- Вертикальной чертой (|) разделяются различные варианты языковой конструкции, из которых следует указать лишь какой-то один. Пример:

  ```
  get_next_by_<имя поля> | get_previous_by_<имя поля>([<условия поиска>])
  ```

  Здесь следует поставить либо `get_next_by_`*<имя поля>*, либо `get_previous_by_`*<имя поля>*.

- Слишком длинные, не помещающиеся на одной строке языковые конструкции автор разрывал на несколько строк и в местах разрывов ставил знак ↳. Например:

```
background: url("bg.jpg") left / auto 100% no-repeat, ↳
url("bg.jpg") right / auto 100% no-repeat;
```

  Приведенный код разбит на две строки, но должен быть набран в одну. Символ ↳ при этом нужно удалить.

- Троеточием (. . .) помечены фрагменты кода, пропущенные ради сокращения объема текста. Пример:

```
INSTALLED_APPS = [
    . . .
    'bboard.apps.BboardConfig',
]
```

  Здесь пропущены все элементы списка, присваиваемого переменной `INSTALLED_APPS`, кроме последнего.

  Обычно такое можно встретить в исправленных впоследствии фрагментах кода — приведены лишь собственно исправленные выражения, а оставшиеся неизмененными пропущены. Также троеточие используется, чтобы показать, в какое место должен быть вставлен вновь написанный код — в начало исходного фрагмента, в его конец или в середину, между уже присутствующими в нем выражениями.

- Полужирным шрифтом выделен вновь добавленный и исправленный код. Пример:

```
class Bb(models.Model):
    . . .
    rubric = models.ForeignKey('Rubric', null=True,
                               on_delete=models.PROTECT, verbose_name='Рубрика')
```

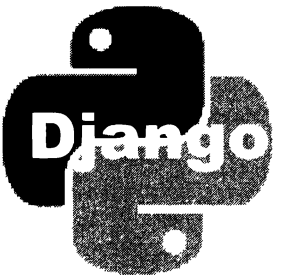
  Здесь вновь добавлен код, объявляющий в модели `Bb` поле `rubric`.

- Зачеркнутым шрифтом выделяется код, подлежащий удалению. Пример:

```
<a class="nav-link" href="#">Грузовой</a>
{% for rubric in rubrics %}
. . .
```

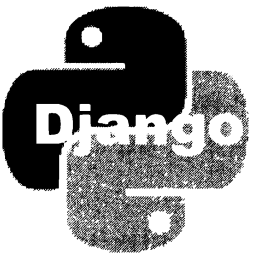
  Тег `<a>`, создающий гиперссылку, следует удалить.

> ***ЕЩЕ РАЗ ВНИМАНИЕ!***
>
> Все приведенные здесь типографские соглашения имеют смысл лишь в форматах написания языковых конструкций Python и Django. В реальном программном коде используются только знак ↳, троеточие, полужирный и зачеркнутый шрифт.



ЧАСТЬ I

Вводный курс



ГЛАВА 1

Основные понятия Django. Вывод данных

В этой главе мы начнем знакомство с фреймворком Django с разработки простенького веб-сайта — электронной доски объявлений.

> ***На заметку***
>
> Эта книга не содержит описания Python. Документацию по этому языку программирования можно найти на его "домашнем" сайте **https://www.python.org/**.

1.1. Установка фреймворка

Установить Django проще всего посредством утилиты pip, поставляемой в составе Python и выполняющей установку дополнительных библиотек из интернет-репозитория PyPI. Запустим командную строку и введем в ней такую команду:

```
pip install django
```

> ***Внимание!***
>
> Если исполняющая среда Python установлена в папке *Program Files* или *Program Files (x86)*, то для установки любых дополнительных библиотек командную строку следует запустить с повышенными правами. Для этого надо найти в меню **Пуск** пункт **Командная строка** (в зависимости от версии Windows он может находиться в группе **Стандартные** или **Служебные**), щелкнуть на нем правой кнопкой мыши и выбрать в появившемся контекстном меню пункт **Запуск от имени администратора** (в Windows 10 этот пункт находится в подменю **Дополнительно**).

Помимо Django, будут установлены библиотеки pytz (обрабатывает *временны́е отметки* — комбинации даты и времени), sqlparse (служит для разбора SQL-кода) и asgiref (реализует интерфейс ASGI, посредством которого эксплуатационный веб-сервер взаимодействует с сайтом, написанным на Django, и который мы рассмотрим в *главе 30*), необходимые фреймворку для работы. Не удаляйте эти библиотеки!

Спустя некоторое время установка закончится, о чем pip нам обязательно сообщит (приведены номера версий Django и дополнительных библиотек, актуальные на момент подготовки книги; порядок следования библиотек может быть другим):

```
Successfully installed Django-3.0.2 asgiref-3.2.3 pytz-2019.3
sqlparse-0.3.0
```

Теперь мы можем начинать разработку нашего первого веб-сайта.

1.2. Проект Django

Первое, что нам нужно сделать, — создать новый проект. *Проектом* называется совокупность всего программного кода, составляющего разрабатываемый сайт. Физически он представляет собой папку, в которой находятся папки и файлы с исходным кодом (назовем ее *папкой проекта*).

Создадим новый, пока еще пустой проект Django, которому дадим имя `samplesite`. Для этого в запущенной ранее командной строке перейдем в папку, в которой должна находиться папка проекта, и отдадим команду:

```
django-admin startproject samplesite
```

Утилита django-admin служит для выполнения разнообразных административных задач. В частности, команда `startproject` указывает ей создать новый проект с именем, записанным после этой команды.

В папке, в которую мы ранее перешли, будет создана следующая структура файлов и папок:

```
samplesite
  manage.py
  samplesite
    __init__.py
    asgi.py
    settings.py
    urls.py
    wsgi.py
```

"Внешняя" папка samplesite — это, как нетрудно догадаться, и есть папка проекта. Как видим, ее имя совпадает с именем проекта, записанным в вызове утилиты django-admin. А содержимое этой папки таково:

- manage.py — программный файл с кодом одноименной служебной утилиты, выполняющей различные действия над проектом;
- "внутренняя" папка samplesite — пакет языка Python, содержащий модули, которые относятся к проекту целиком и задают его конфигурацию (в частности, ключевые настройки). Название этого пакета совпадает с названием проекта, и менять его не стоит — в противном случае придется вносить в код обширные правки.

  В документации по Django этот пакет не имеет какого-либо ясного и однозначного названия. Поэтому, чтобы избежать путаницы, давайте назовем его *пакетом конфигурации*.

Пакет конфигурации включает в себя такие модули:

- __init__.py — пустой файл, сообщающий Python, что папка, в которой он находится, является полноценным пакетом;
- settings.py — модуль с настройками самого проекта. Включает описание конфигурации базы данных проекта, пути ключевых папок, важные параметры, связанные с безопасностью, и пр.;
- urls.py — модуль с маршрутами уровня проекта (о них мы поговорим позже);
- wsgi.py — модуль, связывающий проект с веб-сервером посредством интерфейса WSGI;
- asgi.py (начиная с Django 3.0) — модуль, связывающий проект с веб-сервером через интерфейс ASGI.

  Модули wsgi.py и asgi.py используются при публикации готового сайта в Интернете. Мы будет рассматривать их в *главе 30*.

Еще раз отметим, что пакет конфигурации хранит настройки, относящиеся к самому проекту и влияющие на все приложения, которые входят в состав этого проекта (о приложениях мы поведем разговор очень скоро).

Проект Django мы можем поместить в любое место файловой системы компьютера. Мы также можем переименовать папку проекта. В результате всего этого проект не потеряет своей работоспособности.

1.3. Отладочный веб-сервер Django

В состав Django входит *отладочный веб-сервер*, написанный на самом языке Python. Чтобы запустить его, следует в командной строке перейти непосредственно в папку проекта (именно в нее, а не в папку, в которой находится папка проекта!) и отдать команду:

```
manage.py runserver
```

Здесь мы пользуемся уже утилитой manage.py, сгенерированной программой django-admin при создании проекта. Команда `runserver`, которую мы записали после имени этой утилиты, как раз и запускает отладочный веб-сервер.

Последний выдаст сообщение о том, что сайт успешно запущен (конечно, если его код не содержит ошибок) и доступен по интернет-адресу **http://127.0.0.1:8000/** (или **http://localhost:8000/**). Как видим, отладочный сервер по умолчанию работает через TCP-порт 8000 (впрочем, при необходимости можно использовать другой порт).

Запустим веб-обозреватель и наберем в нем один из интернет-адресов нашего сайта. Мы увидим информационную страничку, предоставленную самим Django и сообщающую, что сайт, хоть еще и "пуст", но в целом работает (рис. 1.1).

Для остановки отладочного веб-сервера достаточно нажать комбинацию клавиш <Ctrl>+<Break>.

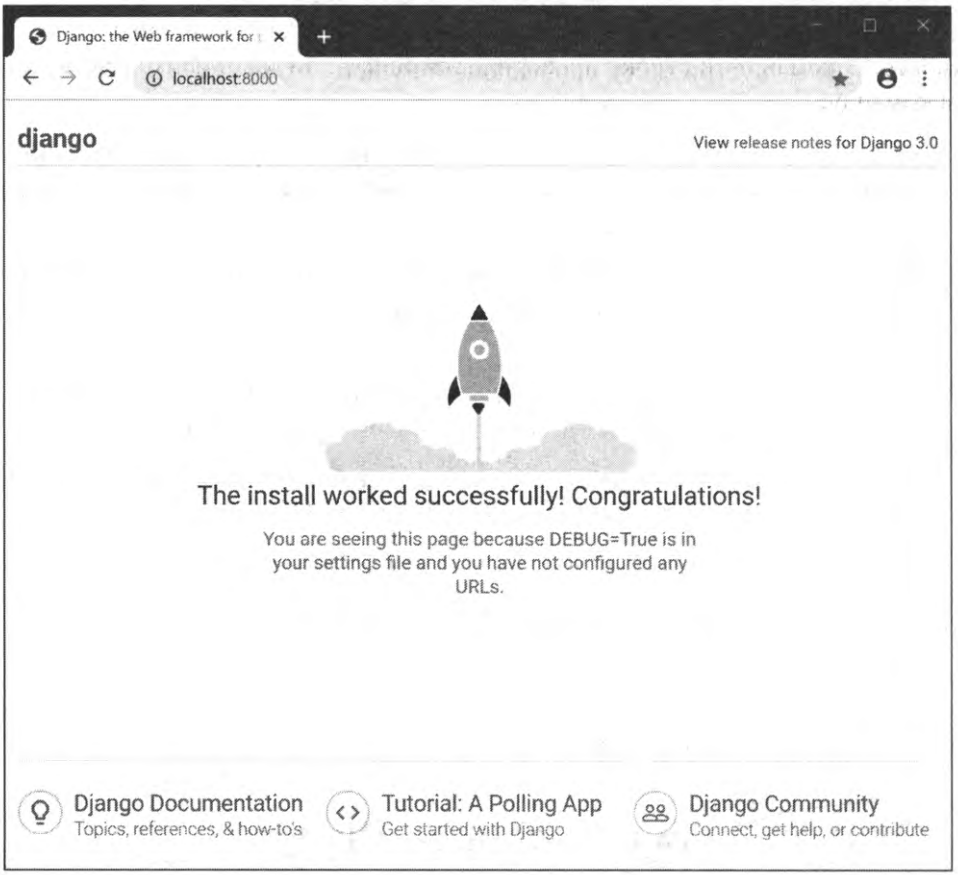


Рис. 1.1. Информационная веб-страница Django, сообщающая о работоспособности вновь созданного "пустого" веб-сайта

1.4. Приложения

Приложение в терминологии Django — это отдельный фрагмент функциональности разрабатываемого сайта, более или менее независимый от других таких же фрагментов и входящий в состав проекта. Приложение может реализовывать работу всего сайта, его раздела или же какой-либо внутренней подсистемы сайта, используемой другими приложениями.

Любое приложение представляется обычным пакетом Python (*пакет приложения*), в котором содержатся модули с программным кодом. Этот пакет находится в папке проекта — там же, где располагается пакет конфигурации. Имя пакета приложения станет именем самого приложения.

Нам нужно сделать так, чтобы наш сайт выводил перечень объявлений, оставленных посетителями. Для этого мы создадим новое приложение, которое незатейливо назовем `bboard`.

Новое приложение создается следующим образом. Сначала остановим отладочный веб-сервер. В командной строке проверим, находимся ли мы в папке проекта, и наберем команду:

```
manage.py startapp bboard
```

Команда `startapp` утилиты manage.py запускает создание нового "пустого" приложения, имя которого указано после этой команды.

Посмотрим, что создала утилита manage.py. Прежде всего это папка bboard, формирующая одноименный пакет приложения и расположенная в папке проекта. В ней находятся следующие папки и файлы:

- migrations — папка вложенного пакета, в котором будут храниться сгенерированные Django миграции (о них разговор пойдет позже). Пока что в папке находится лишь пустой файл __init__.py, помечающий ее как полноценный пакет Python;
- __init__.py — пустой файл, сигнализирующий исполняющей среде Python, что эта папка — пакет;
- admin.py — модуль административных настроек и классов-редакторов;
- apps.py — модуль с настройками приложения;
- models.py — модуль с моделями;
- tests.py — модуль с тестирующими процедурами;
- views.py — модуль с контроллерами.

> ***Внимание!***
>
> Подсистема тестирования кода, реализованная в Django, в этой книге не рассматривается, поскольку автор не считает ее сколь-нибудь полезной.

Зарегистрируем только что созданное приложение в проекте. Найдем в пакете конфигурации файл settings.py (о котором уже упоминалось ранее), откроем его в текстовом редакторе и отыщем следующий фрагмент кода:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Список, хранящийся в переменной `INSTALLED_APPS`, перечисляет все приложения, зарегистрированные в проекте и участвующие в его работе. Изначально в этом списке присутствуют только стандартные приложения, входящие в состав Django и реализующие различные встроенные подсистемы фреймворка. Так, приложение `django.contrib.auth` реализует подсистему разграничения доступа, а приложение `django.contrib.sessions` — подсистему, обслуживающую серверные сессии.

В этой "теплой" компании явно не хватает нашего приложения bboard. Добавим его, включив в список новый элемент:

```
INSTALLED_APPS = [
    . . .
    'bboard.apps.BboardConfig',
]
```

Мы указали строку с путем к классу BboardConfig, описывающему конфигурацию приложения и объявленному в модуле apps.py пакета приложения bboard.

Сохраним и закроем файл settings.py. Но запускать отладочный веб-сервер пока не станем. Вместо этого сразу же напишем первый в нашей практике Django-программирования контроллер.

1.5. Контроллеры

Контроллер Django — это код, запускаемый при обращении по интернет-адресу определенного формата и в ответ выводящий на экран определенную веб-страницу.

> ***Внимание!***
>
> В документации по Django используется термин "view" (вид, или представление). Автор книги считает его неудачным и предпочитает применять термин "контроллер", тем более что это устоявшееся название программных модулей такого типа.

Контроллер Django может представлять собой как функцию (*контроллер-функция*), так и класс (*контроллер-класс*). Первые более универсальны, но зачастую трудоемки в программировании, вторые позволяют выполнить типовые задачи, наподобие вывода списка каких-либо позиций, минимумом кода. И первые, и вторые мы обязательно рассмотрим в последующих главах.

Для хранения кода контроллеров изначально предназначается модуль views.py, создаваемый в каждом пакете приложения. Однако ничто не мешает нам поместить контроллеры в другие модули.

Напишем контроллер, который будет выводить... нет, не список объявлений — этого списка у нас пока нет (у нас и базы данных-то нет), а пока только текст, сообщающий, что будущие посетители сайта со временем увидят на этой страничке список объявлений. Это будет контроллер-функция.

Откроем модуль views.py пакета приложения bboard, удалим имеющийся там небольшой код и заменим его кодом из листинга 1.1.

Листинг 1.1. Простейший контроллер-функция, выводящий текстовое сообщение

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Здесь будет выведен список объявлений.")
```

Наш контроллер — это, собственно, функция `index()`. Единственное, что она делает, — отправляет клиенту текстовое сообщение: **Здесь будет выведен список объявлений**. Но это только пока...

Любой контроллер-функция в качестве единственного обязательного параметра принимает экземпляр класса `HttpRequest`, хранящий различные сведения о полученном запросе: запрашиваемый интернет-адрес, данные, полученные от посетителя, служебную информацию от самого веб-обозревателя и пр. По традиции этот параметр называется `request`. В нашем случае мы его никак не используем.

В теле функции мы создаем экземпляр класса `HttpResponse` (он объявлен в модуле `django.http`), который будет представлять ответ, отправляемый клиенту. Содержимое этого ответа — собственно текстовое сообщение — указываем единственным параметром конструктора этого класса. Готовый экземпляр класса возвращаем в качестве результата.

Что ж, теперь мы с гордостью можем считать себя программистами — поскольку уже самостоятельно написали какой-никакой программный код. Осталось запустить отладочный веб-сервер, набрать в любимом веб-обозревателе адрес вида **http://localhost:8000/bboard/** и посмотреть, что получится...

Минуточку! А с чего мы взяли, что при наборе такого интернет-адреса Django запустит на выполнение именно написанный нами контроллер-функцию `index()`? Ведь мы нигде явно не связали интернет-адрес с контроллером. Но как это сделать?..

1.6. Маршруты и маршрутизатор

Сделать это очень просто. Нужно всего лишь:

- объявить связь пути определенного формата (*шаблонного пути*) с определенным контроллером — иначе говоря, *маршрут*.

  Путь — это часть интернет-адреса, находящаяся между адресом хоста и набором GET-параметров (например, интернет-адрес **http://localhost:8000/bboard/34/edit/** содержит путь **bboard/34/edit/**).

  Шаблонный путь должен завершаться символом слеша. Напротив, начальный слеш в нем не ставится;

- оформить все объявленные нами маршруты в виде *списка маршрутов*;
- оформить маршруты в строго определенном формате, чтобы подсистема *маршрутизатора* смогла использовать готовый список в работе.

При поступлении любого запроса от клиента Django выделяет из запрашиваемого интернет-адреса путь, который передает маршрутизатору. Последний последовательно сравнивает его с шаблонными путями из списка маршрутов. Как только будет найдено совпадение, маршрутизатор передает управление контроллеру, связанному с совпавшим шаблонным путем.

Чтобы при запросе по интернет-адресу **http://localhost:8000/bboard/** запускался только что написанный нами контроллер `index()`, нам нужно связать таковой с шаблонным путем **bboard/**.

В *разд. 1.2*, знакомясь с проектом, мы заметили хранящийся в пакете конфигурации модуль urls.py, в котором записываются маршруты уровня проекта. Откроем этот модуль в текстовом редакторе и посмотрим, что он содержит (листинг 1.2).

Листинг 1.2. Изначальное содержимое модуля urls.py пакета конфигурации

```
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Список маршрутов, оформленный в виде обычного списка Python, присваивается переменной `urlpatterns`. Каждый элемент списка маршрутов (т. е. каждый маршрут) должен представляться в виде результата, возвращаемого функцией `path()` из модуля `django.urls`. Последняя в качестве параметров принимает строку с шаблонным путем и ссылку на контроллер-функцию.

В качестве второго параметра функции `path()` также можно передать список маршрутов уровня приложения. Кстати, этот вариант демонстрируется в выражении, задающем единственный маршрут в листинге 1.2. Мы рассмотрим его потом.

А сейчас добавим в список новый маршрут, связывающий шаблонный путь **bboard/** и контроллер-функцию `index()`. Для чего дополним имеющийся в модуле urls.py код согласно листингу 1.3.

Листинг 1.3. Новое содержимое модуля urls.py пакета конфигурации

```
from django.contrib import admin
from django.urls import path

from bboard.views import index

urlpatterns = [
    path('bboard/', index),
    path('admin/', admin.site.urls),
]
```

Сохраним исправленный файл, запустим отладочный веб-сервер и наберем в веб-обозревателе интернет-адрес **http://localhost:8000/bboard/**. Мы увидим текстовое сообщение, сгенерированное нашим контроллером (рис. 1.2).

Что ж, наши первые контроллер и маршрут работают, и по этому поводу можно порадоваться. Но лишь до поры до времени. Как только мы начнем создавать

сложные сайты, состоящие из нескольких приложений, количество маршрутов в списке вырастет до таких размеров, что мы просто запутаемся в них. Поэтому создатели Django настоятельно рекомендуют применять для формирования списков маршрутов другой подход, о котором мы сейчас поговорим.

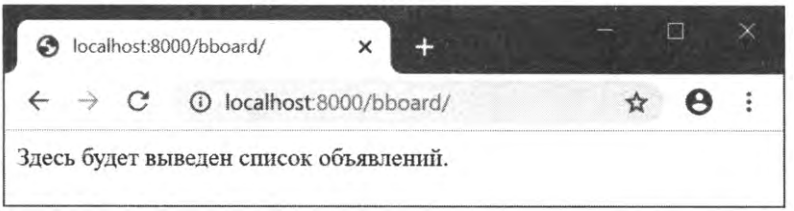


Рис. 1.2. Результат работы нашего первого контроллера — простое текстовое сообщение

Маршрутизатор Django при просмотре списка маршрутов не требует, чтобы путь, полученный из клиентского запроса (реальный), и шаблонный путь, записанный в очередном маршруте, совпадали полностью. Достаточно лишь того факта, что шаблонный путь совпадает с началом реального.

Как было сказано ранее, функция `path()` позволяет указать во втором параметре вместо ссылки на контроллер-функцию другой, *вложенный*, список маршрутов. В таком случае маршрутизатор, найдя совпадение, удалит из реального пути его начальную часть (префикс), совпавшую с шаблонным путем, и приступит к просмотру маршрутов из вложенного списка, используя для сравнения реальный путь уже без префикса.

Исходя из всего этого, мы можем создать иерархию списков маршрутов. В списке из пакета конфигурации (*списке маршрутов уровня проекта*) запишем маршруты, которые указывают на вложенные списки маршрутов, принадлежащие отдельным приложениям (*списки маршрутов уровня приложения*). А в последних непосредственно запишем нужные контроллеры.

Начнем со списка маршрутов уровня приложения `bboard`. Создадим в пакете этого приложения (т. е. в папке bboard) файл urls.py и занесем в него код из листинга 1.4.

Листинг 1.4. Код модуля urls.py пакета приложения `bboard`

```
from django.urls import path

from .views import index

urlpatterns = [
    path('', index),
]
```

Наконец, исправим код модуля urls.py из пакета конфигурации, как показано в листинге 1.5.

Листинг 1.5. Окончательный код модуля urls.py пакета конфигурации

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('bboard/', include('bboard.urls')),
    path('admin/', admin.site.urls),
]
```

Вложенный список маршрутов, указываемый во втором параметре функции `path()`, должен представлять собой результат, возвращенный функцией `include()` из модуля `django.urls`. В качестве единственного параметра эта функция принимает строку с путем к модулю, где записан вложенный список маршрутов.

Как только наш сайт получит запрос с интернет-адресом **http://localhost:8000/bboard/**, маршрутизатор обнаружит, что присутствующий в нем путь совпадает с шаблонным путем **bboard/**, записанным в первом маршруте из листинга 1.5. Он удалит из реального пути префикс, соответствующий шаблонному пути, и получит новый путь — пустую строку. Этот путь совпадет с единственным маршрутом из вложенного списка (см. листинг 1.4), в результате чего запустится записанный в этом маршруте контроллер-функция `index()`, и на экране появится уже знакомое нам текстовое сообщение (см. рис. 1.2).

Поскольку зашла речь о вложенных списках маршрутов, давайте посмотрим на выражение, создающее второй маршрут из списка уровня проекта:

```
path('admin/', admin.site.urls),
```

Этот маршрут связывает шаблонный путь **admin/** со списком маршрутов из свойства `urls` экземпляра класса `AdminSite`, который хранится в переменной `site` и представляет текущий административный веб-сайт Django. Следовательно, набрав интернет-адрес **http://localhost:8000/admin/**, мы попадем на этот административный сайт (разговор о нем будет позже).

1.7. Модели

Настала пора сделать так, чтобы вместо намозолившего глаза текстового сообщения выводились реальные объявления, хранящиеся в таблице базы данных. Для этого нам понадобится прежде всего объявить модель.

Модель — это класс, описывающий определенную таблицу в базе данных, в частности набор имеющихся в ней полей. Отдельный экземпляр класса модели представляет отдельную запись таблицы, позволяет получать значения, хранящиеся в полях записи, и заносить в них новые значения. Модель никак не привязана к конкретному формату базы данных.

Модели объявляются в модуле models.py пакета приложения. Изначально этот модуль пуст.

Объявим модель Bb, представляющую объявление, со следующими полями:

- title — заголовок объявления с названием продаваемого товара (тип — строковый, длина — 50 символов). Поле, обязательное к заполнению;
- content — сам текст объявления, описание товара (тип — memo);
- price — цена (тип — вещественное число);
- published — дата публикации (тип — временна́я отметка, значение по умолчанию — текущие дата и время, индексированное).

Завершим работу отладочного веб-сервера. Откроем модуль models.py пакета приложения bboard и запишем в него код из листинга 1.6.

Листинг 1.6. Код модуля models.py пакета приложения bboard

```
from django.db import models

class Bb(models.Model):
    title = models.CharField(max_length=50)
    content = models.TextField(null=True, blank=True)
    price = models.FloatField(null=True, blank=True)
    published = models.DateTimeField(auto_now_add=True, db_index=True)
```

Модель должна быть подклассом класса Model из модуля django.db.models. Отдельные поля модели объявляются в виде атрибутов класса, которым присваиваются экземпляры классов, представляющих поля разных типов и объявленных в том же модуле. Параметры полей указываются в конструкторах классов полей в виде значений именованных параметров.

Рассмотрим использованные нами классы полей и их параметры:

- CharField — обычное строковое поле фиксированной длины. Допустимая длина значения указывается параметром max_length конструктора;
- TextField — текстовое поле неограниченной длины, или memo-поле. Присвоив параметрам null и blank конструктора значения True, мы укажем, что это поле можно не заполнять (по умолчанию любое поле обязательно к заполнению);
- FloatField — поле для хранения вещественных чисел. Оно также необязательно для заполнения (см. параметры его конструктора);
- DateTimeField — поле для хранения временно́й отметки. Присвоив параметру auto_now_add конструктора значение True, мы предпишем Django при создании новой записи заносить в это поле текущие дату и время. А параметр db_index при присваивании ему значения True укажет создать для этого поля индекс (при выводе объявлений мы будем сортировать их по убыванию даты публикации, и индекс здесь очень пригодится).

Практически всегда таблицы баз данных имеют поле для хранения *ключей* — уникальных значений, однозначно идентифицирующих записи (*ключевое поле*). Как правило, это поле целочисленного типа и помечено как автоинкрементное — тогда

сама СУБД будет заносить в него уникальные номера. В моделях Django такое поле явно объявлять не надо — фреймворк создаст его самостоятельно.

Сохраним исправленный файл. Сейчас мы сгенерируем на его основе миграцию, которая создаст в базе данных все необходимые структуры.

> ***На заметку***
>
> По умолчанию вновь созданный проект Django настроен на использование базы данных в формате SQLite, хранящейся в файле *db.sqlite3* в папке проекта. Эта база данных создается при первом обращении к ней.

1.8. Миграции

Миграция — это программа, сгенерированная на основе заданной модели и создающая в базе данных все описанные этой моделью структуры: таблицу, поля, индексы, правила и связи.

Чтобы сгенерировать миграцию на основе модели `Bb`, переключимся в командную строку, проверим, остановлен ли отладочный веб-сервер и находимся ли мы в папке проекта, и дадим команду:

```
manage.py makemigrations bboard
```

Команда `makemigrations` утилиты manage.py запускает генерирование миграций для всех моделей, объявленных в указанном приложении (у нас — `bboard`) и не изменившихся с момента предыдущего генерирования миграций.

Модули с миграциями сохраняются в пакете `migrations`, находящемся в пакете приложения. Модуль с кодом нашей первой миграции будет иметь имя 0001_initial.py. Откроем его в текстовом редакторе и посмотрим на хранящийся в нем код (листинг 1.7).

Листинг 1.7. Код миграции, создающей структуры для модели `Bb` (приводится с незначительными сокращениями)

```
from django.db import migrations, models

class Migration(migrations.Migration):
    initial = True
    dependencies = [ ]

    operations = [
        migrations.CreateModel(
            name='Bb',
            fields=[
                ('id', models.AutoField(auto_created=True,
                      primary_key=True, serialize=False,
                      verbose_name='ID')),
```

```
                ('title', models.CharField(max_length=50)),
                ('content', models.TextField(blank=True, null=True)),
                ('price', models.FloatField(blank=True, null=True)),
                ('published', models.DateTimeField(auto_now_add=True,
                                db_index=True)),
            ],
        ),
    ]
```

Код миграции вполне понятен и напоминает код написанной ранее модели. Создаваемая в базе данных таблица будет содержать поля `id`, `title`, `content`, `price` и `published`. Ключевое поле `id` для хранения уникальных номеров записей Django создаст самостоятельно.

> ***На заметку***
>
> **Инструменты Django для программирования моделей описаны на страницах https://docs.djangoproject.com/en/3.0/topics/migrations/ и https://docs.djangoproject.com/en/3.0/howto/writing-migrations/.**

Миграция при *выполнении* порождает команды на языке SQL, создающие в базе необходимые структуры. Посмотрим на SQL-код, создаваемый нашей миграцией, задав в командной строке команду:

```
manage.py sqlmigrate bboard 0001
```

После команды `sqlmigrate`, выводящей SQL-код, мы поставили имя приложения и числовую часть имени модуля с миграцией. Прямо в командной строке мы получим такой результат (для удобства чтения был переформатирован):

```
BEGIN;
--
-- Create model Bb
--
CREATE TABLE "bboard_bb" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "title" varchar(50) NOT NULL,
    "content" text NULL,
    "price" real NULL,
    "published" datetime NOT NULL
);
CREATE INDEX "bboard_bb_published_58fde1b5" ON "bboard_bb" ("published");
COMMIT;
```

Этот код был сгенерирован для СУБД SQLite (вспомним — проект Django по умолчанию использует базу данных этого формата). Если применяется другая СУБД, результирующий SQL-код будет соответственно отличаться.

Налюбовавшись на нашу первую миграцию, выполним ее. Для этого наберем в командной строке команду:

```
manage.py migrate
```

Внимание!

Многие стандартные приложения, поставляющиеся в составе Django, хранят свои данные в базе и для создания всех необходимых таблиц и индексов включают в себя набор миграций. Команда `migrate` выполняет *все* миграции, входящие в состав всех приложений проекта и не выполнявшиеся ранее.

Судя по выводящимся в командной строке сообщениям, таких миграций много — десятка два. Дождемся, когда их выполнение завершится, и продолжим.

1.9. Консоль Django

Итак, у нас есть готовая модель для хранения объявлений. Но пока что нет ни одного объявления. Давайте создадим парочку для целей отладки.

Фреймворк включает в свой состав собственную редакцию консоли Python Shell, называемую *консолью Django*. От аналогичной командной среды Python она отличается тем, что в ней в состав путей поиска модулей добавляется путь к папке проекта, в которой запущена эта консоль.

В командной строке наберем команду для запуска консоли Django:

```
manage.py shell
```

И сразу увидим знакомое приглашение >>>, предлагающее нам ввести какое-либо выражение Python и получить результат его выполнения.

1.10. Работа с моделями

Создадим первое объявление — первую запись модели `Bb`:

```
>>> from bboard.models import Bb
>>> b1 = Bb(title='Дача', content='Общество "Двухэтажники". ' + \
        'Два этажа, кирпич, свет, газ, канализация', price=500000)
```

Запись модели создается аналогично экземпляру любого другого класса — вызовом конструктора. Значения полей можно указать в именованных параметрах.

Созданная таким образом запись модели не сохраняется в базе данных, а существует только в оперативной памяти. Чтобы сохранить ее, достаточно вызвать у нее метод `save()` без параметров:

```
>>> b1.save()
```

Проверим, сохранилось ли наше первое объявление, получив значение ключевого поля:

```
>>> b1.pk
1
```

Отлично! Сохранилось.

Атрибут класса `pk`, поддерживаемый всеми моделями, хранит значение ключевого поля текущей записи. А это значение может быть получено только после сохранения записи в базе.

Мы можем обратиться к любому полю записи, воспользовавшись соответствующим ему атрибутом класса модели:

```
>>> b1.title
'Дача'
>>> b1.content
'Общество "Двухэтажники". Два этажа, кирпич, свет, газ, канализация'
>>> b1.price
500000
>>> b1.published
datetime.datetime(2019, 11, 21, 15, 17, 31, 695200, tzinfo=<UTC>)
>>> b1.id
1
```

В последнем случае мы обратились непосредственно к ключевому полю `id`.

Создадим еще одно объявление:

```
>>> b2 = Bb()
>>> b2.title = 'Автомобиль'
>>> b2.content = '"Жигули"'
>>> b2.save()
>>> b2.pk
2
```

Да, можно поступить и так: создать "пустую" запись модели, записав вызов конструктора ее класса без параметров и занеся нужные значения в поля позже.

Что-то во втором объявлении маловато информации о продаваемой машине... Давайте дополним ее:

```
>>> b2.content = '"Жигули", 1980 года, ржавая, некрашеная, сильно битая'
>>> b2.save()
>>> b2.content
'"Жигули", 1980 года, ржавая, некрашеная, сильно битая'
```

Добавим еще одно объявление:

```
>>> Bb.objects.create(title='Дом', content='Трехэтажный, кирпич',
                                   price=50000000)
<Bb: Bb object (3)>
```

Все классы моделей поддерживают атрибут класса `objects`. Он хранит *диспетчер записей* — объект, представляющий все имеющиеся в модели записи и являющийся экземпляром класса `Manager`.

Метод `create()` диспетчера записей создает новую запись модели, принимая в качестве набора именованных параметров значения ее полей, сразу же сохраняет ее и возвращает в качестве результата.

Выведем ключи и заголовки всех объявлений, имеющихся в модели `Bb`:

```
>>> for b in Bb.objects.all():
...     print(b.pk, ': ', b.title)
...
```

```
1 : Дача
2 : Автомобиль
3 : Дом
```

Метод `all()` диспетчера записей возвращает *набор записей* — последовательность из всех записей модели, которую можно перебрать в цикле. Сам набор записей представляется экземпляром класса `QuerySet`, а отдельные записи — экземплярами соответствующего класса модели.

Отсортируем записи модели по заголовку:

```
>>> for b in Bb.objects.order_by('title'):
...     print(b.pk, ': ', b.title)
...
2 : Автомобиль
1 : Дача
3 : Дом
```

Метод `order_by()` диспетчера записей сортирует записи по значению поля, имя которого указано в параметре, и сразу же возвращает набор записей, получившийся в результате сортировки.

Извлечем объявления о продаже домов:

```
>>> for b in Bb.objects.filter(title='Дом'):
...     print(b.pk, ': ', b.title)
...
3 : Дом
```

Метод `filter()` диспетчера записей фильтрует записи по заданным критериям. В частности, чтобы получить только записи, у которых определенное поле содержит заданное значение, следует указать в вызове этого метода именованный параметр, чье имя совпадает с именем поля, и присвоить ему значение, которое должно содержаться в указанном поле. Метод возвращает другой диспетчер записей, содержащий только отфильтрованные записи.

Объявление о продаже автомобиля имеет ключ 2. Отыщем его:

```
>>> b = Bb.objects.get(pk=2)
>>> b.title
'Автомобиль'
>>> b.content
'"Жигули", 1980 года, ржавая, некрашеная, сильно битая'
```

Метод `get()` диспетчера записей имеет то же назначение, что и метод `filter()`, и вызывается аналогичным образом. Однако он ищет не все подходящие записи, а лишь одну и возвращает ее в качестве результата.

Давайте удалим это ржавое позорище:

```
>>> b.delete()
(1, {'bboard.Bb': 1})
```

Метод `delete()` модели, как уже понятно, удаляет текущую запись и возвращает сведения о количестве удаленных записей, обычно малополезные.

Ладно, хватит пока! Выйдем из консоли Django, набрав команду `exit()`.

И сделаем так, чтобы контроллер `index()` выводил список объявлений, отсортированный по убыванию даты их публикации.

Откроем модуль views.py пакета приложения `bboard` и исправим хранящийся в нем код согласно листингу 1.8.

Листинг 1.8. Код модуля views.py пакета приложения bboard

```
from django.http import HttpResponse

from .models import Bb

def index(request):
    s = 'Список объявлений\r\n\r\n\r\n'
    for bb in Bb.objects.order_by('-published'):
        s += bb.title + '\r\n' + bb.content + '\r\n\r\n'
    return HttpResponse(s, content_type='text/plain; charset=utf-8')
```

Чтобы отсортировать объявления по убыванию даты их публикации, мы в вызове метода `order_by()` диспетчера записей предварили имя поля `published` символом "минус". Список объявлений мы представили в виде обычного текста, разбитого на строки символами `\r\n`.

При создании экземпляра класса `HttpResponse`, представляющего отсылаемый клиенту ответ, в именованном параметре `content_type` конструктора указали тип отправляемых данных: обычный текст, набранный в кодировке UTF-8 (если мы этого не сделаем, веб-обозреватель посчитает текст HTML-кодом и выведет его одной строкой, скорее всего, в нечитаемом виде).

Сохраним исправленный файл и запустим отладочный веб-сервер. На рис. 1.3 показан результат наших столь долгих трудов. Теперь наш сайт стал больше похож на доску объявлений.

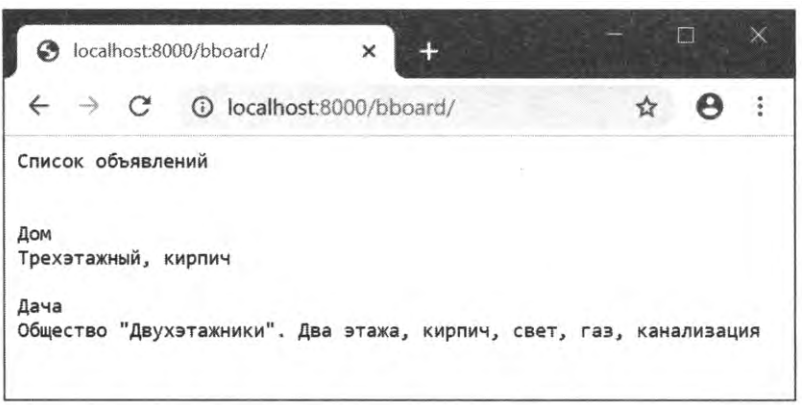


Рис. 1.3. Вывод списка объявлений в виде обычного текста

Точно так же можно сгенерировать полноценную веб-страницу. Но есть более простой способ — применение шаблонов.

1.11. Шаблоны

Шаблон — это образец для генерирования веб-страницы, отправляемой клиенту в составе ответа. Генерированием страниц на основе шаблонов занимается подсистема Django, называемая *шаблонизатором*.

Шаблон Django — это файл с HTML-кодом страницы, содержащий особые команды шаблонизатора: директивы, теги и фильтры. *Директивы* указывают поместить в заданное место HTML-кода какое-либо значение, *теги* управляют генерированием содержимого, а *фильтры* выполняют какие-либо преобразования указанного значения перед выводом.

По умолчанию шаблонизатор ищет все шаблоны в папках templates, вложенных в папки пакетов приложений (это поведение можно изменить, задав соответствующие настройки, о которых мы поговорим в *главе 11*). Сами файлы шаблонов веб-страниц должны иметь расширение html.

Остановим отладочный сервер. Создадим в папке пакета приложения `bboard` папку templates, а в ней — вложенную папку bboard. Сохраним в этой папке наш первый шаблон index.html, код которого приведен в листинге 1.9.

Листинг 1.9. Код шаблона bboard\index.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Главная :: Доска объявлений</title>
    </head>
    <body>
        <h1>Объявления</h1>
        {% for bb in bbs %}
        <div>
            <h2>{{ bb.title }}</h2>
            <p>{{ bb.content }}</p>
            <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
        </div>
        {% endfor %}
    </body>
</html>
```

В целом здесь нам все знакомо. За исключением команд шаблонизатора. Давайте познакомимся с ними.

Начнем вот с этого тега шаблонизатора:

```
{% for bb in bbs %}
    . . .
{% endfor %}
```

Аналогично циклу `for...in` языка Python, он перебирает последовательность, хранящуюся в переменной `bbs` (которая входит в состав контекста шаблона, о котором мы поговорим чуть позже), присваивая очередной элемент переменной `bb`, которая доступна в теле цикла. У нас переменная `bbs` будет хранить перечень объявлений, и, таким образом, переменной `bb` будет присваиваться очередное объявление.

Теперь познакомимся с директивой шаблонизатора:

```
{{ bb.title }}
```

Она указывает извлечь значение из атрибута `title` объекта, хранящегося в переменной `bb`, и вставить это значение в то место кода, в котором находится она сама.

И, наконец, фильтр `date`:

```
<p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
```

Он преобразует значение из атрибута `published` объекта `bb`, т. е. временну́ю отметку публикации объявления, в формат, указанный в виде строки после двоеточия. Строка `"d.m.Y H:i:s"` задает формат *<число>.<номер месяца>.<год из четырех цифр> <часы в 24-часовом формате>:<минуты>:<секунды>*.

1.12. Контекст шаблона, рендеринг и сокращения

Откроем модуль views.py пакета приложения `bboard` и внесем исправления в его код согласно листингу 1.10.

Листинг 1.10. Код модуля views.py пакета приложения bboard (используются низкоуровневые инструменты)

```
from django.http import HttpResponse
from django.template import loader

from .models import Bb

def index(request):
    template = loader.get_template('bboard/index.html')
    bbs = Bb.objects.order_by('-published')
    context = {'bbs': bbs}
    return HttpResponse(template.render(context, request))
```

Сначала загружаем шаблон, воспользовавшись функцией `get_template()` из модуля `django.template.loader`. В качестве параметра указываем строку с путем к файлу

шаблона от папки templates. Функция вернет экземпляр класса `Template`, представляющий загруженный из заданного файла шаблон.

Далее формируем *контекст шаблона* — набор данных, которые будут выведены на генерируемой странице. Контекст шаблона должен представлять собой обычный словарь Python, элементы которого преобразуются в доступные внутри шаблона переменные, одноименные ключам этих элементов. Так, элемент `bbs` создаваемого нами контекста шаблона, содержащий перечень объявлений, будет преобразован в переменную `bbs`, доступную в шаблоне.

Наконец, выполняем *рендеринг* шаблона, т. е. генерирование на его основе веб-страницы. Для этого вызываем метод `render()` класса `Template`, передав ему подготовленный ранее контекст шаблона и экземпляр класса `HttpRequest`, представляющий клиентский запрос и полученный контроллером-функцией через параметр `request`. Результат — строку с HTML-кодом готовой веб-страницы — передаем конструктору класса `HttpResponse` для формирования ответа.

Сохраним оба исправленных файла, запустим отладочный веб-сервер и посмотрим на результат. Вот теперь это действительно веб-страница (рис. 1.4)!

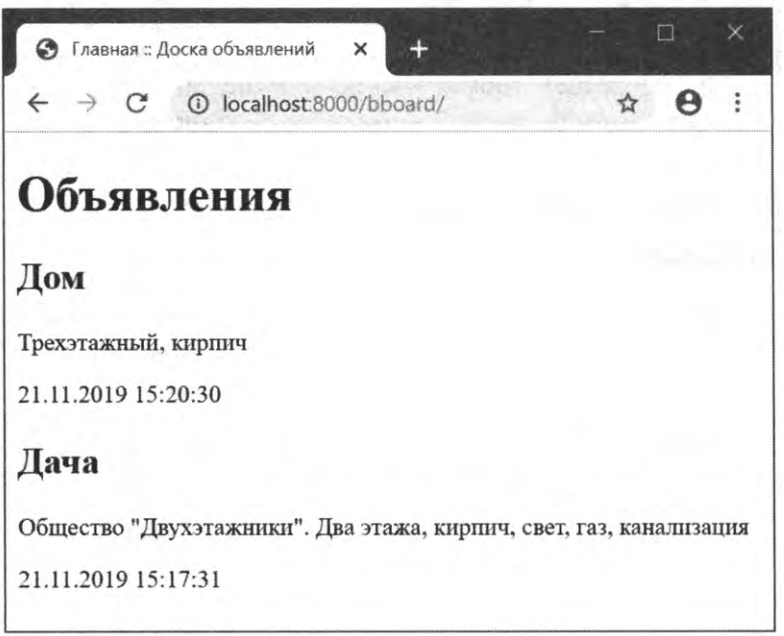


Рис. 1.4. Веб-страница, сформированная с применением шаблона

В коде контроллера `index()` (см. листинг 1.10) для рендеринга мы использовали низкоуровневые инструменты, несколько усложнив код. Но Django предоставляет средства более высокого уровня — функции-*сокращения* (shortcuts). Так, функция-сокращение `render()` из модуля `django.shortcuts` выполняет и загрузку, и рендеринг шаблона. Попробуем ее в деле, исправив код модуля views.py, как показано в листинге 1.11.

Листинг 1.11. Код модуля views.py пакета приложения bboard (используется функция-сокращение render())

```
from django.shortcuts import render

from .models import Bb

def index(request):
    bbs = Bb.objects.order_by('-published')
    return render(request, 'bboard/index.html', {'bbs': bbs})
```

Обновим веб-страницу со списком объявлений и убедимся, что этот код работает точно так же, как и написанный ранее, однако при этом имеет меньший объем.

1.13. Административный веб-сайт Django

Все-таки два объявления — это слишком мало... Давайте добавим еще несколько. Только сделаем это не в консоли Django, а на встроенном в этот фреймворк административном сайте.

Административный веб-сайт предоставляет доступ ко всем моделям, объявленным во всех приложениях проекта. Мы можем просматривать, добавлять, править и удалять записи, фильтровать и сортировать их. Помимо этого, административный сайт не пускает к данным сайта посторонних, используя для этого встроенную во фреймворк подсистему разграничения доступа.

Эта подсистема реализована в стандартном приложении `django.contrib.auth`. А работу самого административного сайта обеспечивает стандартное приложение `django.contrib.admin`. Оба этих приложения заносятся в список зарегистрированных в проекте изначально.

Стандартное приложение `django.contrib.auth` использует для хранения списков зарегистрированных пользователей, групп и их привилегий особые модели. Для них в базе данных должны быть созданы таблицы, и создание этих таблиц выполняют особые миграции. Следовательно, чтобы встроенные в Django средства разграничения доступа работали, нужно хотя бы один раз выполнить миграции (см. *разд. 1.8*).

Еще нужно создать зарегистрированного пользователя сайта с максимальными правами — *суперпользователя*. Для этого остановим отладочный веб-сервер и отдадим в командной строке команду:

```
manage.py createsuperuser
```

Утилита manage.py запросит у нас имя создаваемого суперпользователя, его адрес электронной почты и пароль, который потребуется ввести дважды:

```
Username (leave blank to use 'vlad'): admin
Email address: admin@somesite.ru
Password:
Password (again):
```

Пароль должен содержать не менее 8 символов, буквенных и цифровых, набранных в разных регистрах. Если пароль не удовлетворяет этим требованиям, утилита выдаст соответствующие предупреждения и спросит, создавать ли суперпользователя:

```
This password is too short. It must contain at least 8 characters.
This password is too common.
This password is entirely numeric.
Bypass password validation and create user anyway? [y/N]: y
```

Для создания суперпользователя с таким паролем следует ввести символ "y" и нажать <Enter>.

Как только суперпользователь будет успешно создан, появится уведомление:

```
Superuser created successfully.
```

После этого русифицируем проект Django. Откроем модуль settings.py пакета конфигурации и найдем в нем вот такое выражение:

```
LANGUAGE_CODE = 'en-us'
```

Переменная `LANGUAGE_CODE` задает код языка, используемого при выводе системных сообщений и страниц административного сайта. Изначально это американский английский язык (код `en-us`). Исправим это выражение, занеся в него код русского языка:

```
LANGUAGE_CODE = 'ru'
```

Сохраним исправленный модуль и закроем его — более он нам не понадобится.

Запустим отладочный веб-сервер и войдем на административный сайт, перейдя по интернет-адресу **http://localhost:8000/admin/**. Будет выведена страница входа с формой (рис. 1.5), в которой нужно набрать имя и пароль, введенные при создании суперпользователя, и нажать кнопку **Войти**.

Если мы ввели имя и пароль пользователя без ошибок, то увидим страницу со списком приложений, зарегистрированных в проекте и объявляющих какие-либо

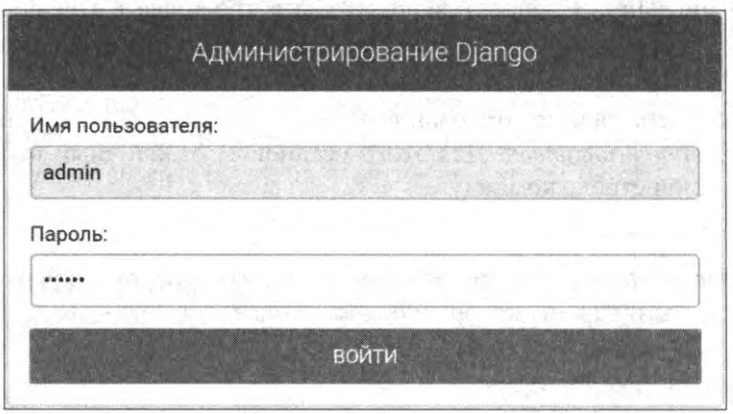


Рис. 1.5. Веб-страница входа административного веб-сайта Django

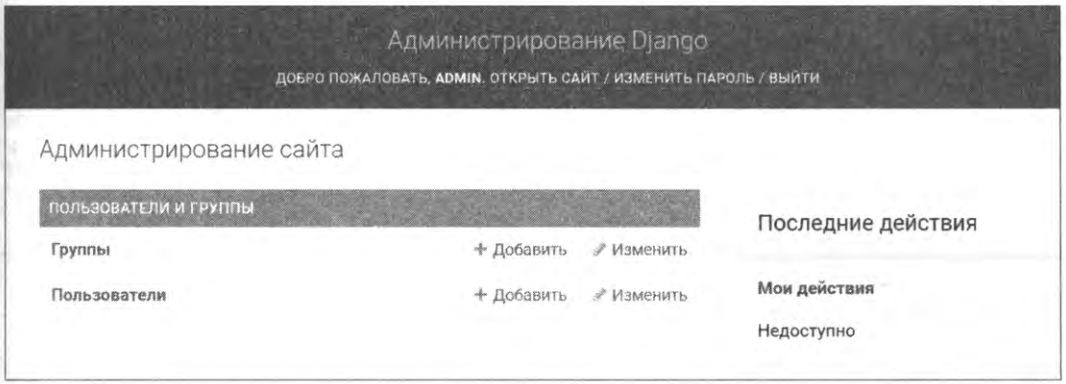


Рис. 1.6. Веб-страница списка приложений административного сайта

модели (рис. 1.6). Под названием каждого приложения перечисляются объявленные в нем модели.

Но постойте! В списке присутствует только одно приложение — **Пользователи и группы** (так обозначается встроенное приложение `django.contrib.auth`) — с моделями **Группы** и **Пользователи**. Где же наше приложение `bboard` и его модель `Bb`?

Чтобы приложение появилось в списке административного сайта, его нужно явно зарегистрировать там. Сделать это очень просто. Откроем модуль административных настроек admin.py пакета приложения `bboard` и заменим имеющийся в нем небольшой код фрагментом, представленным в листинге 1.12.

Листинг 1.12. Код модуля admin.py пакета приложения `bboard` (выполнена регистрация модели `Bb` на административном сайте)

```
from django.contrib import admin

from .models import Bb

admin.site.register(Bb)
```

Мы вызвали метод `register()` у экземпляра класса `AdminSite`, представляющего сам административный сайт и хранящегося в переменной `site` модуля `django.contrib.admin`. Этому методу мы передали в качестве параметра ссылку на класс нашей модели `Bb`.

Как только мы сохраним модуль и обновим открытую в веб-обозревателе страницу списка приложений, сразу увидим, что наше приложение также присутствует в списке (рис. 1.7). Совсем другое дело!

Каждое название модели в этом списке представляет собой гиперссылку, щелкнув на которой мы попадем на страницу списка записей этой модели. Например, на рис. 1.8 показана страница списка записей, хранящихся в модели `Bb`.

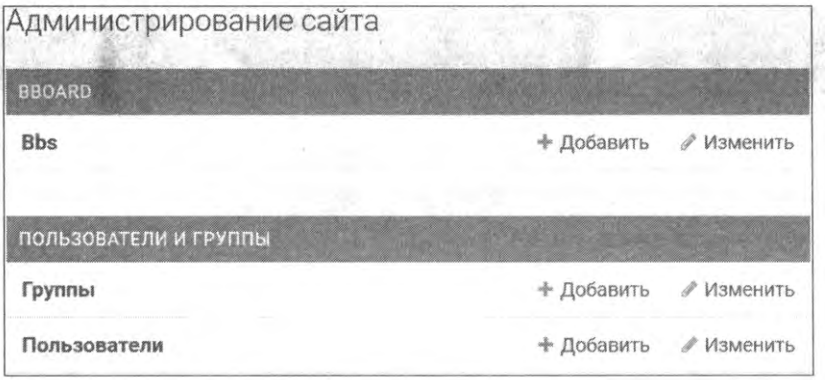


Рис. 1.7. Приложение `bboard` в списке приложений административного веб-сайта

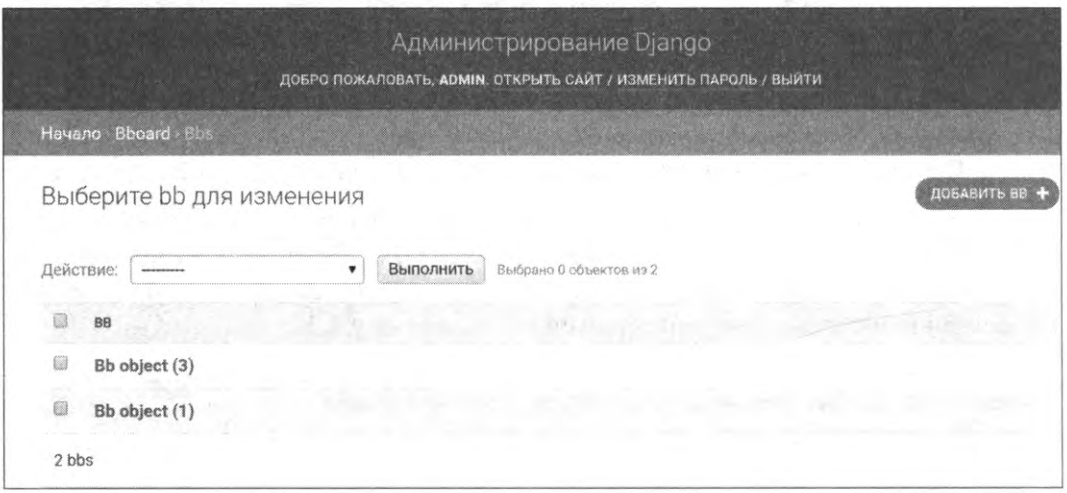


Рис. 1.8. Список записей, хранящихся в модели `Bb`

Здесь мы можем:

- щелкнуть на гиперссылке **Добавить *<имя класса модели>***, чтобы вывести страницу добавления новой записи (рис. 1.9). Занеся в элементы управления нужные данные, нажмем кнопку:
  - **Сохранить** — для сохранения записи и возврата на страницу списка записей;
  - **Сохранить и продолжить редактирование** — для сохранения записи с возможностью продолжить ее редактирование;
  - **Сохранить и добавить другой объект** — для сохранения записи и подготовки формы для ввода новой записи;
- щелкнуть на нужной записи, чтобы вывести страницу ее правки. Эта страница похожа на страницу для добавления записи (см. рис. 1.9), за исключением того, что в наборе имеющихся на ней кнопок будет присутствовать еще одна — **Удалить**, выполняющая удаление текущей записи;

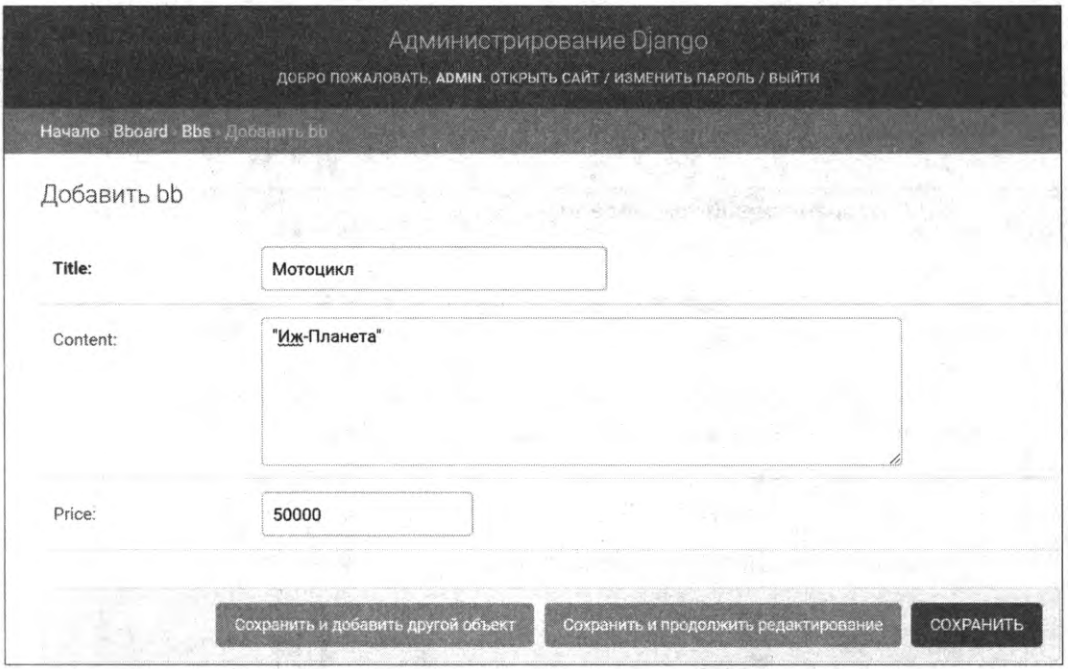


Рис. 1.9. Веб-страница для добавления записи в модель

- выполнить над выбранными записями модели какое-либо из поддерживаемых действий.

  Чтобы выбрать запись, следует установить флажок, находящийся в левой колонке. Можно выбрать сколько угодно записей.

  Все поддерживаемые действия перечислены в раскрывающемся списке **Действия** — нужно лишь выбрать требуемое и нажать расположенную правее кнопку **Выполнить**. Веб-обозреватель покажет страницу подтверждения, где будут перечислены записи, над которыми выполнится действие; текст с вопросом, уверен ли пользователь; и кнопки с "говорящими" надписями **Да, я уверен** и **Нет, отменить и вернуться к выбору**.

Мы можем попробовать ради эксперимента добавить несколько записей в модель `Bb`, исправить кое-какие записи и удалить ненужные. Но сначала давайте доведем нашу модель до ума.

1.14. Параметры полей и моделей

Во-первых, наша модель представляется какой-то непонятной аббревиатурой **Bbs**, а не простым и ясным текстом **Объявления**. Во-вторых, на страницах добавления и правки записи в качестве надписей у элементов управления проставлены имена полей модели (**title**, **content** и **price**), что обескуражит пользователя. И в-третьих, объявления было бы неплохо отсортировать по убыванию даты публикации.

Одним словом, нам надо задать параметры как для полей модели, так и для самой модели.

Откроем модуль models.py пакета приложения bboard и исправим код класса модели Bb, как показано в листинге 1.13.

Листинг 1.13. Исправленный код модели Bb

```
class Bb(models.Model):
    title = models.CharField(max_length=50, verbose_name='Товар')
    content = models.TextField(null=True, blank=True,
                                          verbose_name='Описание')
    price = models.FloatField(null=True, blank=True, verbose_name='Цена')
    published = models.DateTimeField(auto_now_add=True, db_index=True,
                                     verbose_name='Опубликовано')

    class Meta:
        verbose_name_plural = 'Объявления'
        verbose_name = 'Объявление'
        ordering = ['-published']
```

В вызов каждого конструктора класса поля мы добавили именованный параметр verbose_name. Он указывает "человеческое" название поля, которое будет выводиться на экран.

В классе модели мы объявили вложенный класс Meta, а в нем — атрибуты класса, которые зададут параметры уже самой модели:

- verbose_name_plural — название модели во множественном числе;
- verbose_name — название модели в единственном числе.

  Эти названия также будут выводиться на экран;
- ordering — последовательность полей, по которым по умолчанию будет выполняться сортировка записей.

Если теперь сохраним исправленный модуль и обновим открытую в веб-обозревателе страницу, то увидим, что:

- в списке приложений наша модель обозначается надписью **Объявления** (значение атрибута verbose_name_plural вложенного класса Meta) вместо **Bbs**;
- на странице списка записей сама запись модели носит название **Объявление** (значение атрибута verbose_name вложенного класса Meta) вместо **Bb**;
- на страницах добавления и правки записи элементы управления имеют надписи **Товар**, **Описание** и **Цена** (значения параметра verbose_name конструкторов классов полей) вместо **title**, **content** и **price**.

Кстати, теперь мы можем исправить код контроллера index() (объявленного в модуле views.py пакета приложения), убрав из выражения, извлекающего список записей, указание сортировки:

```
def index(request):
    bbs = Bb.objects.all()
    . . .
```

Как видим, сортировка по умолчанию, заданная в параметрах модели, действует не только в административном сайте.

1.15. Редактор модели

Стало лучше, не так ли? Но до идеала все же далековато. Так, на странице списка записей все позиции представляются невразумительными строками вида ***<имя класса модели>* object (*<значение ключа>*)** (см. рис. 1.8), из которых невозможно понять, что же хранится в каждой из этих записей.

Таково представление модели на административном сайте по умолчанию. Если же оно нас не устраивает, мы можем задать свои параметры представления модели, объявив для нее класс-*редактор*.

Редактор объявляется в модуле административных настроек admin.py пакета приложения. Откроем его и заменим имеющийся в нем код фрагментом, показанным в листинге 1.14.

Листинг 1.14. Код модуля admin.py пакета приложения bboard (для модели Bb объявлен редактор BbAdmin)

```
from django.contrib import admin

from .models import Bb

class BbAdmin(admin.ModelAdmin):
    list_display = ('title', 'content', 'price', 'published')
    list_display_links = ('title', 'content')
    search_fields = ('title', 'content', )

admin.site.register(Bb, BbAdmin)
```

Класс редактора объявляется как производный от класса `ModelAdmin` из модуля `django.contrib.admin`. Он содержит набор атрибутов класса, которые и задают параметры представления модели. Мы использовали следующие атрибуты класса:

- `list_display` — последовательность имен полей, которые должны выводиться в списке записей;
- `list_display_links` — последовательность имен полей, которые должны быть преобразованы в гиперссылки, ведущие на страницу правки записи;
- `search_fields` — последовательность имен полей, по которым должна выполняться фильтрация.

У нас в списке записей будут присутствовать поля названия, описания продаваемого товара, его цены и временно́й отметки публикации объявления. Значения из пер-

вых двух полей преобразуются в гиперссылки на страницы правки соответствующих записей. Фильтрация записей будет выполняться по тем же самым полям.

Обновим открытый в веб-обозревателе административный сайт и перейдем на страницу со списком записей модели `Bb`. Она должна выглядеть так, как показано на рис. 1.10.

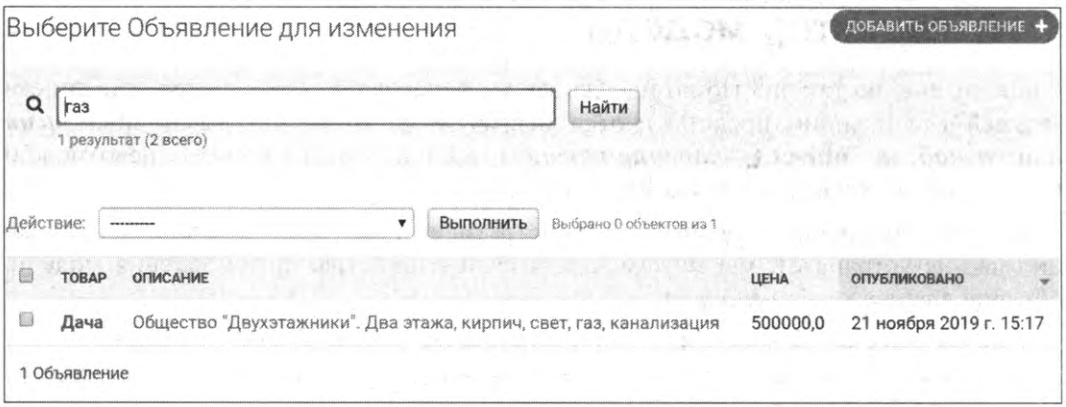
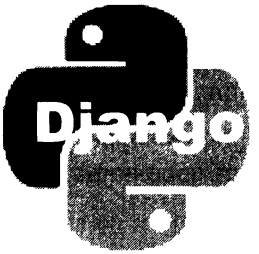


Рис. 1.10. Список записей модели `Bb` после указания для нее редактора (была выполнена фильтрация по слову "газ")

Помимо всего прочего, мы можем выполнять фильтрацию записей по значениям полей, перечисленных в последовательности, которая была присвоена атрибуту `search_fields` класса редактора. Для этого достаточно занести в расположенное над списком поле ввода искомое слово и нажать находящуюся правее кнопку **Найти**. Так, на рис. 1.10 показан список записей, отфильтрованных по слову "газ".

Теперь можно сделать небольшой перерыв. На досуге побродите по административному сайту, выясните, какие полезные возможности, помимо рассмотренных здесь, он предлагает. Можете также добавить в модель `Bb` еще пару объявлений.



ГЛАВА 2

Связи. Ввод данных. Статические файлы

Ладно, немного передохнули, повозились с административным сайтом — и хватит бездельничать! Пора заканчивать работу над электронной доской объявлений.

Предварительно выйдем с административного сайта и остановим отладочный веб-сервер.

2.1. Связи между моделями

На всех приличных онлайновых досках объявлений все объявления разносятся по тематическим рубрикам: недвижимость, транспорт, бытовая техника и др. Давайте сделаем так и мы.

Сначала объявим класс модели `Rubric`, которая будет представлять рубрики объявлений, дописав в модуль **models.py** пакета приложения `bboard` код из листинга 2.1.

Листинг 2.1. Код модели `Rubric`

```
class Rubric(models.Model):
    name = models.CharField(max_length=20, db_index=True,
                                           verbose_name='Название')

    class Meta:
        verbose_name_plural = 'Рубрики'
        verbose_name = 'Рубрика'
        ordering = ['name']
```

Модель содержит всего одно поле `name`, которое будет хранить название рубрики. Для него мы сразу велели создать индекс, т. к. будем выводить перечень рубрик отсортированным по их названиям.

Теперь нужно добавить в модель `Bb` *поле внешнего ключа*, устанавливающее связь между текущей записью этой модели и записью модели `Rubric`, т. е. между объяв-

лением и рубрикой, к которой оно относится. Таким образом будет создана *связь "один-со-многими"*, при которой одна запись модели `Rubric` (рубрика) будет связана с произвольным количеством записей модели `Bb` (объявлений). Модель `Rubric` станет первичной, а `Bb` — вторичной.

Создадим во вторичной модели такое поле, назвав его `rubric`[1]:

```
class Bb(models.Model):
    . . .
    rubric = models.ForeignKey('Rubric', null=True,
                               on_delete=models.PROTECT, verbose_name='Рубрика')

    class Meta:
        . . .
```

Класс `ForeignKey` представляет поле внешнего ключа, в котором фактически будет храниться ключ записи из первичной модели. Первым параметром конструктору этого класса передается строка с именем класса первичной модели, поскольку вторичная модель у нас объявлена раньше первичной.

Все поля моделей по умолчанию обязательны к заполнению. Следовательно, добавить новое, обязательное к заполнению поле в модель, которая уже содержит записи, нельзя — сама СУБД откажется делать это и выведет сообщение об ошибке. Нам придется явно пометить добавляемое поле `rubric` как необязательное, присвоив параметру `null` значение `True`.

Именованный параметр `on_delete` управляет каскадными удалениями записей вторичной модели после удаления записи первичной модели, с которой они были связаны. Значение `PROTECT` этого параметра запрещает каскадные удаления (чтобы какой-нибудь несообразительный администратор не стер разом уйму объявлений, удалив рубрику, к которой они относятся).

Сохраним исправленный модуль и сгенерируем миграции, которые внесут необходимые изменения в базу данных:

```
manage.py makemigrations bboard
```

В результате в папке migrations будет создан модуль миграции с именем вида 0002_auto_*<отметка текущих даты и времени>*.py. Если хотите, откройте этот модуль и посмотрите на его код (который слишком велик, чтобы приводить его здесь). Если вкратце, то новая миграция создаст таблицу для модели `Rubric` и добавит в таблицу модели `Bb` новое поле `rubric`.

> ***На заметку***
>
> Помимо всего прочего, эта миграция задаст для полей модели `Bb` параметры `verbose_name`, а для самой модели — параметры `verbose_name_plural`, `verbose_name` и `ordering`, которые мы указали в *главе 1*. Скорее всего, это делается, как говорится, для галочки — подобные изменения, произведенные в классе модели, в действительности никак не отражаются на базе данных.

[1] Полужирным шрифтом помечен добавленный или исправленный код (подробности о типографских соглашениях — во *введении*).

Выполним созданную миграцию:

```
manage.py migrate
```

Сразу же зарегистрируем новую модель на административном сайте, добавив в модуль admin.py пакета приложения два выражения:

```
from .models import Rubric
admin.site.register(Rubric)
```

Запустим отладочный веб-сервер, войдем на административный сайт и добавим в модель Rubric рубрики "Недвижимость" и "Транспорт".

2.2. Строковое представление модели

Все хорошо, только в списке записей модели Rubric все рубрики представляются строками вида ***<имя класса модели>*** **object (*<значение ключа записи>*)** (нечто подобное поначалу выводилось у нас в списке записей модели Bb — см. рис. 1.8). Работать с таким списком неудобно, а потому давайте что-то с ним сделаем.

Можно объявить для модели Rubric класс редактора и задать в нем перечень полей, которые должны выводиться в списке (см. *разд. 1.15*). Но этот способ лучше подходит только для моделей с несколькими значащими полями, а в нашей модели такое поле всего одно.

Еще можно переопределить в классе модели метод __str__(self), возвращающий строковое представление класса. Давайте так и сделаем.

Откроем модуль models.py, если уже закрыли его, и добавим в объявление класса модели Rubric вот этот код:

```
class Rubric(models.Model):
    . . .
    def __str__(self):
        return self.name

    class Meta:
        . . .
```

В качестве строкового представления мы выводим название рубрики (собственно, более там выводить нечего).

Сохраним файл, обновим страницу списка рубрик в административном сайте и посмотрим, что у нас получилось. Совсем другой вид (рис. 2.1)!

Перейдем на список записей модели Bb и исправим каждое имеющееся в ней объявление, задав для него соответствующую рубрику. Обратим внимание, что на странице правки записи рубрика выбирается с помощью раскрывающегося списка, в котором перечисляются строковые представления рубрик (рис. 2.2).

Справа от этого списка (в порядке слева направо) присутствуют кнопки для правки выбранной в списке записи первичной модели, добавления новой записи и удале-

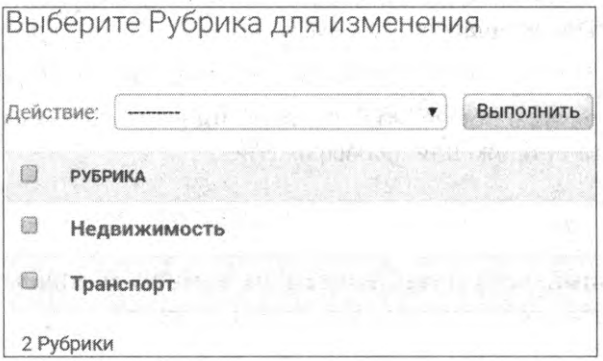


Рис. 2.1. Список рубрик (выводятся строковые представления записей модели)

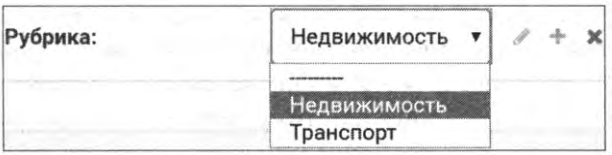


Рис. 2.2. Для указания значения поля внешнего ключа применяется раскрывающийся список

ния выбранной. Так что мы сразу же сможем при необходимости добавить новую рубрику, исправить или удалить существующую.

Осталось сделать так, чтобы в списке записей модели `Bb`, помимо всего прочего, выводились рубрики объявлений. Для чего достаточно добавить в последовательность имен полей, присвоенную атрибуту `list_display` класса `BbAdmin`, поле `rubric`:

```
class BbAdmin(admin.ModelAdmin):
    list_display = ('title', 'content', 'price', 'published', 'rubric')
    . . .
```

Обновим страницу списка объявления — и сразу увидим в нем новый столбец **Рубрика**. Отметим, что и здесь в качестве значения поля выводится строковое представление связанной записи модели.

2.3. URL-параметры и параметризованные запросы

Логичным шагом выглядит разбиение объявлений по рубрикам не только при хранении, но и при выводе на экран. Давайте создадим на пока что единственной странице нашего сайта панель навигации, в которой выведем список рубрик, и при щелчке на какой-либо рубрике будем выводить лишь относящиеся к ней объявления. Остановим отладочный сервер и подумаем.

Чтобы контроллер, выводящий объявления, смог выбрать из модели лишь относящиеся к указанной рубрике, он должен получить ключ рубрики. Его можно передать через GET-параметр: **/bboard/?rubric=*<ключ рубрики>***.

Однако Django позволяет поместить параметр непосредственно в составе интернет-адреса: **bboard/*<ключ рубрики>*/**. То есть через *URL-параметр*.

Для этого нужно указать маршрутизатору, какую часть интернет-адреса считать URL-параметром, каков тип значения этого параметра и какое имя должно быть у параметра контроллера, которому будет присвоено значение URL-параметра, извлеченного из адреса. Откроем модуль urls.py пакета приложения `bboard` и внесем в него такие правки (добавленный код выделен полужирным шрифтом):

```
. . .
from .views import index, by_rubric

urlpatterns = [
    path('<int:rubric_id>/', by_rubric),
    path('', index),
]
```

Мы добавили в начало набора маршрутов еще один, с шаблонным путем **<int:rubric_id>/**. В нем угловые скобки помечают описание URL-параметра, языковая конструкция `int` задает целочисленный тип этого параметра, а `rubric_id` — имя параметра контроллера, которому будет присвоено значение этого URL-параметра. Созданному маршруту мы сопоставили контроллер-функцию `by_rubric()`, который вскоре напишем.

Получив запрос по интернет-адресу **http://localhost:8000/bboard/2/**, маршрутизатор выделит путь **bboard/2/**, удалит из него префикс **bboard** и выяснит, что полученный путь совпадает с первым маршрутом из приведенного ранее списка. После чего запустит контроллер `by_rubric`, передав ему в качестве параметра выделенный из интернет-адреса ключ рубрики 2.

Маршруты, содержащие URL-параметры, носят название *параметризованных*.

Откроем модуль views.py и добавим в него код контроллера-функции `by_rubric()` (листинг 2.2).

Листинг 2.2. Код контроллера-функции by_rubric()

```
from .models import Rubric

def by_rubric(request, rubric_id):
    bbs = Bb.objects.filter(rubric=rubric_id)
    rubrics = Rubric.objects.all()
    current_rubric = Rubric.objects.get(pk=rubric_id)
    context = {'bbs': bbs, 'rubrics': rubrics,
               'current_rubric': current_rubric}
    return render(request, 'bboard/by_rubric.html', context)
```

В объявление функции мы добавили параметр `rubric_id` — именно ему будет присвоено значение URL-параметра, выбранное из интернет-адреса. В состав контек-

ста шаблона поместили список объявлений, отфильтрованных по полю внешнего ключа rubric, список всех рубрик и текущую рубрику (она нужна нам, чтобы вывести на странице ее название). Остальное нам уже знакомо.

Создадим в папке templates\bboard пакета приложения шаблон by_rubric.html с кодом, приведенным в листинге 2.3.

Листинг 2.3. Код шаблона bboard\by_rubric.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{{ current_rubric.name }} :: Доска объявлений</title>
    </head>
    <body>
        <h1>Объявления</h1>
        <h2>Рубрика: {{ current_rubric.name }}</h2>
        <div>
            <a href="/bboard/">Главная</a>
            {% for rubric in rubrics %}
            <a href="/bboard/{{ rubric.pk }}/">{{ rubric.name }}</a>
            {% endfor %}
        </div>
        {% for bb in bbs %}
        <div>
            <h2>{{ bb.title }}</h2>
            <p>{{ bb.content }}</p>
            <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
        </div>
        {% endfor %}
    </body>
</html>
```

Обратим внимание, как формируются интернет-адреса в гиперссылках, находящихся в панели навигации и представляющих отдельные рубрики.

Исправим контроллер index() и шаблон bboard\index.html таким образом, чтобы на главной странице нашего сайта выводилась та же самая панель навигации. Помимо этого, сделаем так, чтобы в составе каждого объявления выводилось название рубрики, к которой оно относится, выполненное в виде гиперссылки. Код исправленного контроллера index() показан в листинге 2.4.

Листинг 2.4. Исправленный код контроллера-функции index()

```
def index(request):
    bbs = Bb.objects.all()
```

```
    rubrics = Rubric.objects.all()
    context = {'bbs': bbs, 'rubrics': rubrics}
    return render(request, 'bboard/index.html', context)
```

Полный код исправленного шаблона bboard\index.html приведен в листинге 2.5.

Листинг 2.5. Код шаблона bboard\index.html (реализован вывод панели навигации и рубрики, к которой относится каждое объявление)

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Главная :: Доска объявлений</title>
    </head>
    <body>
        <h1>Объявления</h1>
        <div>
            <a href="/bboard/">Главная</a>
            {% for rubric in rubrics %}
            <a href="/bboard/{{ rubric.pk }}/">{{ rubric.name }}</a>
            {% endfor %}
        </div>
        {% for bb in bbs %}
        <div>
            <h2>{{ bb.title }}</h2>
            <p>{{ bb.content }}</p>
            <p><a href="/bboard/{{ bb.rubric.pk }}/">
            {{ bb.rubric.name }}</a></p>
            <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
        </div>
        {% endfor %}
    </body>
</html>
```

Сохраним исправленные файлы, запустим отладочный веб-сервер и перейдем на главную страницу. Мы сразу увидим панель навигации, располагающуюся непосредственно под заголовком, и гиперссылки на рубрики, к которым относятся объявления (рис. 2.3). Перейдем по какой-либо гиперссылке-рубрике и посмотрим на страницу со списком относящихся к ней объявлений (рис. 2.4).

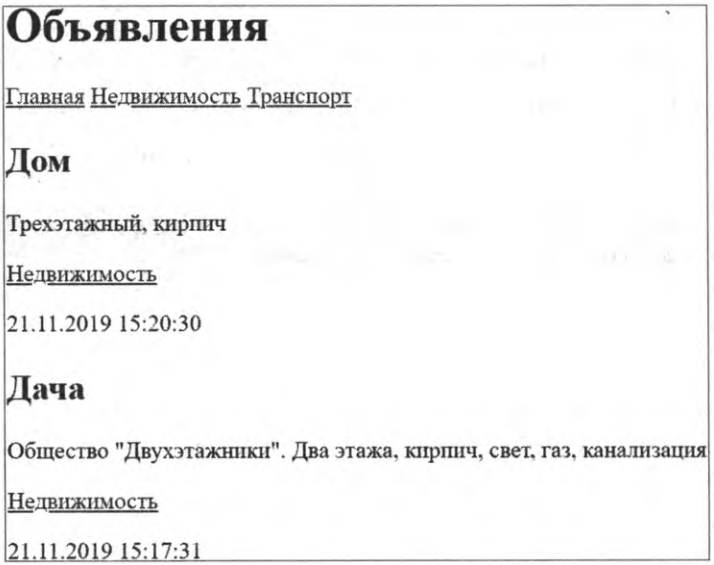


Рис. 2.3. Исправленная главная веб-страница с панелью навигации и обозначениями рубрик

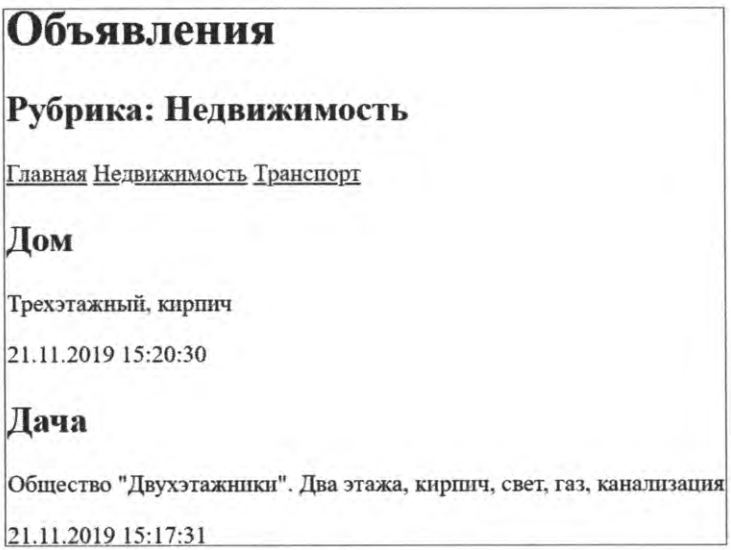


Рис. 2.4. Веб-страница объявлений, относящихся к выбранной рубрике

2.4. Обратное разрешение интернет-адресов

Посмотрим на HTML-код гиперссылок, указывающих на страницы рубрик (листинг 2.3):

```
<a href="/bboard/{{ rubric.pk }}/">{{ rubric.name }}</a>
```

Интернет-адрес целевой страницы, подставляемый в атрибут `href` тега `<a>`, здесь формируется явно из префикса **bboard** и ключа рубрики, и, как мы убедились, все

это работает. Но предположим, что мы решили поменять префикс, например, на **bulletinboard**. Неужели придется править код гиперссылок во всех шаблонах?

Избежать неприятностей такого рода позволяет инструмент Django, называемый *обратным разрешением* интернет-адресов. Это особый тег шаблонизатора, формирующий интернет-адрес на основе имени маршрута, в котором он записан, и набора URL-параметров, если это параметризованный маршрут.

Сначала нужно дать маршрутам имена, создав тем самым *именованные маршруты*. Откроем модуль urls.py пакета приложения и исправим код, создающий набор маршрутов, следующим образом:

```
. . .
urlpatterns = [
    path('<int:rubric_id>/', by_rubric, name='by_rubric'),
    path('', index, name='index'),
]
```

Имя маршрута указывается в именованном параметре `name` функции `path()`.

Далее следует вставить в код гиперссылок теги шаблонизатора `url`, которые и выполняют обратное разрешение интернет-адресов. Откроем шаблон bboard\index.html, найдем в нем фрагмент кода:

```
<a href="/bboard/{{ rubric.pk }}/">
```

и заменим его на:

```
<a href="{% url 'by_rubric' rubric.pk %}">
```

Имя маршрута указывается первым параметром тега `url`, а значения URL-параметров — последующими.

Найдем код, создающий гиперссылку на главную страницу:

```
<a href="/bboard/">
```

Заменим его на:

```
<a href="{% url 'index' %}">
```

Маршрут `index` не является параметризованным, поэтому URL-параметры здесь не указываются.

Исправим код остальных гиперссылок в наших шаблонах. Обновим страницу, открытую в веб-обозревателе, и попробуем выполнить несколько переходов по гиперссылкам. Все должно работать.

2.5. Формы, связанные с моделями

Осталось создать еще одну страницу — для добавления новых объявлений.

Для ввода данных Django предоставляет *формы* — объекты, "умеющие" выводить на страницу веб-формы с необходимыми элементами управления и проверять занесенные в них данные на корректность. А *форма, связанная с моделью*, даже может самостоятельно сохранить данные в базе!

Создадим форму, связанную с моделью Bb и служащую для ввода новых объявлений. Дадим ее классу имя BbForm.

Остановим отладочный веб-сервер. Создадим в пакете приложения bboard модуль forms.py, в который занесем код из листинга 2.6.

Листинг 2.6. Код класса формы BbForm, связанной с моделью Bb

```
from django.forms import ModelForm

from .models import Bb

class BbForm(ModelForm):
    class Meta:
        model = Bb
        fields = ('title', 'content', 'price', 'rubric')
```

Класс формы, связанной с моделью, является производным от класса ModelForm из модуля django.forms. Во вложенном классе Meta указываются параметры формы: класс модели, с которой она связана (атрибут класса model), и последовательность из имен полей модели, которые должны присутствовать в форме (атрибут класса fields).

2.6. Контроллеры-классы

Обрабатывать формы, связанные с моделью, можно в контроллерах-функциях. Но лучше использовать высокоуровневый контроллер-класс, который сам выполнит бóльшую часть действий по выводу и обработке формы.

Наш первый контроллер-класс будет носить имя BbCreateView. Его мы объявим в модуле views.py пакета приложения. Код этого класса показан в листинге 2.7.

Листинг 2.7. Код контроллера-класса BbCreateView

```
from django.views.generic.edit import CreateView

from .forms import BbForm

class BbCreateView(CreateView):
    template_name = 'bboard/create.html'
    form_class = BbForm
    success_url = '/bboard/'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['rubrics'] = Rubric.objects.all()
        return context
```

Контроллер-класс мы сделали производным от класса `CreateView` из модуля `django.views.generic.edit`. Базовый класс "знает", как создать форму, вывести на экран страницу с применением указанного шаблона, получить занесенные в форму данные, проверить их, сохранить в новой записи модели и перенаправить пользователя в случае успеха на заданный интернет-адрес.

Все необходимые сведения мы указали в атрибутах объявленного класса:

- `template_name` — путь к файлу шаблона, создающего страницу с формой;
- `form_class` — ссылка на класс формы, связанной с моделью;
- `success_url` — интернет-адрес для перенаправления после успешного сохранения данных (в нашем случае это адрес главной страницы).

Поскольку на каждой странице сайта должен выводиться перечень рубрик, мы переопределили метод `get_context_data()`, формирующий контекст шаблона. В теле метода получаем контекст шаблона от метода базового класса, добавляем в него список рубрик и, наконец, возвращаем его в качестве результата.

Займемся шаблоном bboard\create.html. Его код представлен в листинге 2.8.

Листинг 2.8. Код шаблона bboard\create.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Добавление объявления :: Доска объявлений</title>
    </head>
    <body>
        <h1>Добавление объявления</h1>
        <div>
            <a href="{% url 'index' %}">Главная</a>
            {% for rubric in rubrics %}
            <a href="{% url 'by_rubric' rubric.pk %}">
            {{ rubric.name }}</a>
            {% endfor %}
        </div>
        <form method="post">
            {% csrf_token %}
            {{ form.as_p }}
            <input type="submit" value="Добавить">
        </form>
    </body>
</html>
```

Здесь нас интересует только код, создающий веб-форму. Обратим внимание на четыре важных момента:

- форма в контексте шаблона хранится в переменной `form`, создаваемой контроллером-классом;

- для вывода формы, элементы управления которой находятся на отдельных абзацах, применяется метод `as_p()` класса формы;
- метод `as_p()` генерирует только код, создающий элементы управления. Тег `<form>`, необходимый для создания самой формы, и тег `<input>`, формирующий кнопку отправки данных, придется писать самостоятельно.

  В теге `<form>` мы указали метод отправки данных POST, но не записали интернет-адрес, по которому будут отправлены данные из формы. В этом случае данные будут отправлены по тому же интернет-адресу, с которого была загружена текущая страница, т. е. в нашем случае тому же контроллеру-классу `BbCreateView`, который благополучно обработает и сохранит их;
- в теге `<form>` мы поместили тег шаблонизатора `csrf_token`. Он создает в форме скрытое поле, хранящее цифровой жетон, получив который контроллер "поймет", что данные были отправлены с текущего сайта и им можно доверять. Это часть подсистемы безопасности Django.

Добавим в модуль urls.py пакета приложения маршрут, указывающий на контроллер `CreateView`:

```
. . .
from .views import index, by_rubric, BbCreateView
urlpatterns = [
    path('add/', BbCreateView.as_view(), name='add'),
    . . .
]
```

В вызов функции `path()` подставляется результат, возвращенный методом `as_view()` контроллера-класса.

Не забудем создать в панели навигации всех страниц гиперссылку на страницу добавления объявления:

```
<a href="{% url 'add' %}">Добавить</a>
```

Запустим отладочный веб-сервер, откроем сайт и щелкнем на гиперссылке **Добавить**. На странице добавления объявления (рис. 2.5) введем новое объявление и нажмем кнопку **Добавить**. Когда на экране появится главная страница, мы сразу же увидим новое объявление.

В объявлении класса `BbCreateView` мы опять указали интернет-адрес перенаправления (в атрибуте класса `success_url`) непосредственно, что считается дурным тоном программирования. Сгенерируем его путем обратного разрешения.

Откроем модель views.py и внесем следующие правки в код класса `BbCreateView`:

```
. . .
from django.urls import reverse_lazy

class BbCreateView(CreateView):
    . . .
    success_url = reverse_lazy('index')
    . . .
```

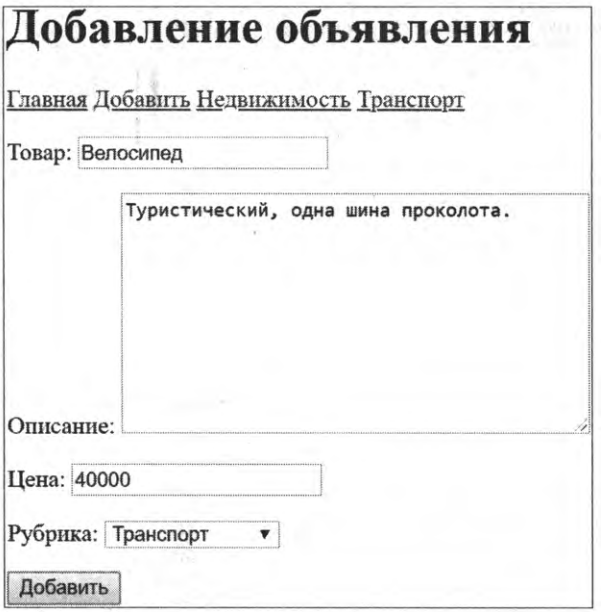


Рис. 2.5. Веб-страница добавления нового объявления

Функция `reverse_lazy()` из модуля `django.urls` принимает имя маршрута и значения всех входящих в маршрут URL-параметров (если они там есть). Результатом станет готовый интернет-адрес.

2.7. Наследование шаблонов

Еще раз посмотрим на листинги 2.3, 2.5 и 2.8. Что сразу бросается в глаза? Большой объем совершенно одинакового HTML-кода: секция заголовка, панель навигации, всевозможные служебные теги. Это увеличивает совокупный объем шаблонов и усложняет сопровождение — если мы решим изменить что-либо в этом коде, то будем вынуждены вносить правки в каждый из имеющихся у нас шаблонов.

Решить эту проблему можно, применив *наследование шаблонов*, аналогичное наследованию классов. Шаблон, являющийся базовым, содержит повторяющийся код, в нужных местах которого объявлены *блоки*, помечающие места, куда будет вставлено содержимое из производных шаблонов. Каждый блок имеет уникальное в пределах шаблона имя.

Создадим базовый шаблон со следующими блоками:

- `title` — будет помещаться в теге `<title>` и служить для создания уникального названия для каждой страницы;
- `content` — будет использоваться для размещения уникального содержимого страниц.

Создадим в папке templates папку layout. Сохраним в ней наш базовый шаблон basic.html, код которого приведен в листинге 2.9.

Листинг 2.9. Код базового шаблона layout\basic.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Главная{% endblock %} ::
        Доска объявлений</title>
    </head>
    <body>
        <header>
            <h1>Объявления</h1>
        </header>
        <nav>
            <a href="{% url 'index' %}">Главная</a>
            <a href="{% url 'add' %}">Добавить</a>
            {% for rubric in rubrics %}
            <a href="{% url 'by_rubric' rubric.pk %}">
            {{ rubric.name }}</a>
            {% endfor %}
        </nav>
        <section>
        {% block content %}
        {% endblock %}
        </section>
    </body>
</html>
```

Начало объявляемого блока помечается тегом шаблонизатора `block`, за которым должно следовать имя блока. Завершается блок тегом `endblock`.

Объявленный в базовом шаблоне блок может быть пустым:

```
{% block content %}
{% endblock %}
```

или иметь какое-либо изначальное содержимое:

```
{% block title %}Главная{% endblock %}
```

Оно будет выведено на страницу, если производный шаблон не задаст для блока свое содержимое.

Сделаем шаблон bboard\index.html производным от шаблона layout\basic.html. Новый код этого шаблона приведен в листинге 2.10.

Листинг 2.10. Код производного шаблона bboard\index.html

```
{% extends "layout/basic.html" %}

{% block content %}
{% for bb in bbs %}
```

```
<div>
    <h2>{{ bb.title }}</h2>
    <p>{{ bb.content }}</p>
    <p><a href="{% url 'by_rubric' bb.rubric.pk %}">
    {{ bb.rubric.name }}</a></p>
    <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
</div>
{% endfor %}
{% endblock %}
```

В самом начале кода любого производного шаблона ставится тег шаблонизатора `extends`, в котором указывается путь к базовому шаблону. Далее следуют объявления блоков, обозначаемые теми же тегами `block` и `endblock`, в которых записывается их содержимое.

Если мы теперь сохраним исправленные файлы и обновим открытую в веб-обозревателе главную страницу, то увидим, что она выводится точно так же, как ранее.

Аналогично исправим шаблоны **bboard\by_rubric.html** (листинг 2.11) и **bboard\create.html** (листинг 2.12). Сразу видно, насколько уменьшился их объем.

Листинг 2.11. Код производного шаблона bboard\by_rubric.html

```
{% extends "layout/basic.html" %}

{% block title %}{{ current_rubric.name }}{% endblock %}

{% block content %}
<h2>Рубрика: {{ current_rubric.name }}</h2>
{% for bb in bbs %}
<div>
    <h2>{{ bb.title }}</h2>
    <p>{{ bb.content }}</p>
    <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
</div>
{% endfor %}
{% endblock %}
```

Листинг 2.12. Код производного шаблона bboard\create.html

```
{% extends "layout/basic.html" %}

{% block title %}Добавление объявления{% endblock %}

{% block content %}
<h2>Добавление объявления</h2>
<form method="post">
    {% csrf_token %}
```

```
    {{ form.as_p }}
    <input type="submit" value="Добавить">
</form>
{% endblock %}
```

2.8. Статические файлы

Наш первый сайт практически готов. Осталось навести небольшой лоск, применив таблицу стилей style.css из листинга 2.13.

Листинг 2.13. Таблица стилей style.css

```
header h1 {
    font-size: 40pt;
    text-transform: uppercase;
    text-align: center;
    background: url("bg.jpg") left / auto 100% no-repeat;
}
nav {
    font-size: 16pt;
    width: 150px;
    float: left;
}
nav a {
    display: block;
    margin: 10px 0px;
}
section {
    margin-left: 170px;
}
```

Но где нам сохранить эту таблицу стилей и файл с фоновым изображением bg.jpg? И вообще, как привязать таблицу стилей к базовому шаблону?

Файлы, содержимое которых не обрабатывается программно, а пересылается клиенту как есть, в терминологии Django носят название *статических*. К таким файлам относятся, например, таблицы стилей и графические изображения, помещаемые на страницы.

Остановим отладочный веб-сервер. Создадим в папке пакета приложения `bboard` папку static, а в ней — вложенную папку bboard. В последней сохраним файлы style.css и bg.jpg (можно использовать любое подходящее изображение, загруженное из Интернета).

Откроем базовый шаблон layout\basic.html и вставим в него следующий код:

```
{% load static %}

<!DOCTYPE html>
```

```
<html>
    <head>
        . . .
        <link rel="stylesheet" type="text/css"
        href="{% static 'bboard/style.css' %}">
    </head>
    . . .
</html>
```

Тег шаблонизатора `load` загружает указанный в нем *модуль расширения*, содержащий дополнительные теги и фильтры. В нашем случае выполняется загрузка модуля `static`, который содержит теги для вставки ссылок на статические файлы.

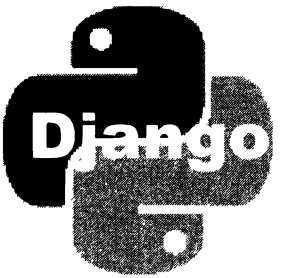
Один из этих тегов — `static` — генерирует полный интернет-адрес статического файла на основе заданного в нем пути относительно папки static. Мы используем его, чтобы вставить в HTML-тег `<link>` интернет-адрес таблицы стилей bboard\style.css.

Запустим отладочный сервер и откроем главную страницу сайта. Теперь она выглядит гораздо презентабельнее (рис. 2.6).

Занесем на сайт еще несколько объявлений (специально придумывать текст для них совершенно необязательно — для целей отладки можно записать любую тарабарщину, как это сделал автор). Попробуем при наборе очередного объявления не вводить какое-либо обязательное для занесения значение, скажем, название товара, и посмотрим, что случится. Попробуем добавить на страницы, например, поддон.

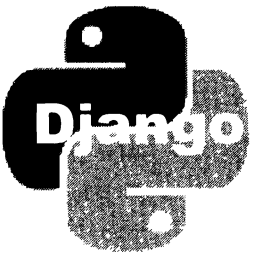
И закончив тем самым вводный курс Django-программирования, приступим к изучению базовых возможностей этого замечательного веб-фреймворка.

Рис. 2.6. Главная страница сайта после применения к ней таблицы стилей



ЧАСТЬ II

Базовые инструменты Django



ГЛАВА 3

Создание и настройка проекта

Прежде чем начать писать код сайта, следует создать проект этого сайта, указать его настройки и сформировать все необходимые приложения.

3.1. Подготовка к работе

Перед началом разработки сайта на Django, даже перед созданием его проекта, следует выполнять действия, перечисленные далее.

1. Проверить версии установленного ПО: исполняющей среды Python и серверных СУБД, если используемые проектом базы данных хранятся на них:
   - Python — должен быть версии 3.6 и новее. Python 2 не поддерживается.

     Дистрибутив исполняющей среды Python можно загрузить со страницы **https://www.python.org/downloads/**;
   - MySQL — 5.6 и новее;
   - MariaDB — 10.1 и новее;
   - PostgreSQL — 9.5 и новее.
2. Установить сам фреймворк Django. Сделать это можно, указав в командной строке следующую команду:

   ```
   pip install django
   ```

   Описание других способов установки Django (в том числе путем клонирования Git-репозитория с ресурса GitHub) можно найти по интернет-адресу **https://docs.djangoproject.com/en/3.0/topics/install/**.
3. Установить клиентские программы и дополнительные библиотеки для работы с СУБД:
   - SQLite — ничего дополнительно устанавливать не нужно;
   - MySQL — понадобится установить:
     - клиент этой СУБД — по интернет-адресу **https://dev.mysql.com/downloads/mysql/** находится универсальный дистрибутив, позволяющий установить

клиент, а также, при необходимости, сервер, коннекторы (соединители) к различным средам исполнения, документацию и примеры;

- mysqlclient — Python-библиотеку, служащую коннектором между Python и MySQL, для чего достаточно отдать команду:

```
pip install mysqlclient
```

В качестве альтернативы обеим этим программам можно установить коннектор Connector/Python, также входящий в состав универсального дистрибутива MySQL и не требующий установки клиента данной СУБД. Однако его разработка несколько "отстает" от разработки Python, и в составе дистрибутива может не оказаться редакции коннектора под современную версию этого языка (так, во время написания книги поставлялась редакция под версию Python 3.7, хотя уже была в наличии версия 3.8);

- MariaDB — используются те же программы, что и в случае MySQL;
- PostgreSQL — понадобится установить:
  - клиент этой СУБД — по интернет-адресу **https://www.postgresql.org/download/** можно загрузить универсальный дистрибутив, позволяющий установить клиент, а также, при необходимости, сервер и дополнительные утилиты;
  - psycopg2 — Python-библиотеку, служащую коннектором между Python и PostgreSQL, для чего следует отдать команду:

```
pip install psycopg2
```

4. Создать базу данных, в которой будут храниться данные сайта. Процедура ее создания зависит от формата базы:
   - SQLite — база создается автоматически при первом обращении к ней;
   - MySQL и MariaDB — в утилите MySQL Workbench, поставляемой в составе универсального дистрибутива MySQL;
   - PostgreSQL — посредством программы pgAdmin, входящей в комплект поставки PostgreSQL.

Также Django поддерживает работу с базами данных Oracle, Microsoft SQL Server, Firebird, IBM DB2 и механизмом ODBC. Действия, которые необходимо выполнить для успешного подключения к таким базам, описаны на странице **https://docs.djangoproject.com/en/3.0/ref/databases/**.

3.2. Создание проекта Django

Новый проект Django создается командой `startproject` утилиты django-admin, отдаваемой в следующем формате:

```
django-admin startproject <имя проекта> [<путь к папке проекта>]
```

Если *путь к папке проекта* не указан, папка проекта будет создана в текущей папке и получит то же *имя*, что и сам проект. В противном случае папкой проекта станет папка с указанным в команде *путем*.

Папку проекта можно впоследствии переместить в любое другое место файловой системы, а также переименовать. Никакого влияния на работу проекта эти действия не оказывают.

3.3. Настройки проекта

Настройки проекта указываются в модуле settings.py пакета конфигурации. Значительная их часть имеет значения по умолчанию, оптимальные в большинстве случаев.

Настройки хранятся в обычных переменных Python. Имя переменной и есть имя соответствующего ей параметра.

3.3.1. Основные настройки

- `BASE_DIR` — путь к папке проекта. По умолчанию вычисляется автоматически;
- `DEBUG` — режим работы сайта: отладочный (значение `True`) или эксплуатационный (`False`). По умолчанию — `False` (эксплуатационный режим), однако сразу при создании проекта для этого параметра указано значение `True` (т. е. сайт для облегчения разработки сразу же вводится в отладочный режим).

  Если сайт работает в *отладочном режиме*, то при возникновении любой ошибки в коде сайта Django выводит веб-страницу с детальным описанием этой ошибки. В *эксплуатационном режиме* в таких случаях выводятся стандартные сообщения веб-сервера наподобие "Страница не найдена" или "Внутренняя ошибка сервера". Помимо того, в эксплуатационном режиме действуют более строгие настройки безопасности;
- `DEFAULT_CHARSET` — кодировка веб-страниц по умолчанию. По умолчанию: `utf-8`;
- `ROOT_URLCONF` — путь к модулю, в котором записаны маршруты уровня проекта, в виде строки. Значение этого параметра указывается сразу при создании проекта;
- `SECRET_KEY` — секретный ключ в виде строки с произвольным набором символов. Используется программным ядром Django и подсистемой разграничения доступа для шифрования важных данных.

  Значение параметра по умолчанию — пустая строка. Однако непосредственно при создании проекта ему присваивается секретный ключ, сгенерированный утилитой django-admin.

  Менять этот секретный ключ без особой необходимости не стоит. Также его следует хранить в тайне, в противном случае он может попасть в руки злоумышленникам, которые используют его для атаки на сайт.

На заметку

Настройка FILE_CHARSET, указывающая кодировку текстовых файлов, в частности файлов шаблонов, с версии 2.2 объявлена устаревшей и не рекомендованной к применению. Все текстовые файлы, используемые Django, теперь должны быть сохранены в кодировке UTF-8.

3.3.2. Параметры баз данных

Все базы данных, используемые проектом, записываются в параметре DATABASES. Его значением должен быть словарь Python. Ключи элементов этого словаря задают псевдонимы баз данных, зарегистрированных в проекте. Можно указать произвольное число баз данных. Если при выполнении операций с моделями база данных не указана явно, то будет использоваться база с псевдонимом default.

В качестве значений элементов словаря также указываются словари, хранящие, собственно, параметры соответствующей базы данных. Каждый элемент вложенного словаря указывает отдельный параметр.

Значение параметра DATABASES по умолчанию — пустой словарь. Однако при создании проекта ему дается следующее значение:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Оно указывает единственную базу данных, применяемую по умолчанию. База записывается в формате SQLite и хранится в файле db.sqlite3 в папке проекта.

Вот параметры баз данных, поддерживаемые Django:

- ENGINE — формат используемой базы данных. Указывается как путь к модулю, реализующему работу с нужным форматом баз данных, в виде строки. Доступны следующие значения:
  - django.db.backends.sqlite3 — SQLite;
  - django.db.backends.mysql — MySQL;
  - django.db.backends.postgresql — PostgreSQL;
  - django.db.backends.oracle — Oracle;
- NAME — путь к файлу базы данных, если используется SQLite, или имя базы данных в случае серверных СУБД;
- TIME_ZONE — временна́я зона для значений даты и времени, хранящихся в базе. Используется в том случае, если формат базы данных не поддерживает хранение значений даты и времени с указанием временно́й зоны. Значение по умолчанию — None (значение временно́й зоны берется из одноименного параметра проекта, описанного в *разд. 3.3.5*).

Следующие параметры используются только в случае серверных СУБД:

- HOST — интернет-адрес компьютера, на котором работает СУБД;
- PORT — номер TCP-порта, через который выполняется подключение к СУБД. По умолчанию — пустая строка (используется порт по умолчанию);
- USER — имя пользователя, от имени которого Django подключается к базе данных;
- PASSWORD — пароль пользователя, от имени которого Django подключается к базе;
- CONN_MAX_AGE — время, в течение которого соединение с базой данных будет открыто, в виде целого числа в секундах. Если задано значение 0, соединение будет закрываться сразу после обработки запроса. Если задано значение None, соединение будет открыто всегда. По умолчанию — 0;
- OPTIONS — дополнительные параметры подключения к базе данных, специфичные для используемой СУБД. Записываются в виде словаря, в котором каждый элемент указывает отдельный параметр. По умолчанию — пустой словарь.

Вот пример кода, задающего параметры для подключения к базе данных site формата MySQL, обслуживаемой СУБД, которая работает на том же компьютере, от имени пользователя siteuser с паролем sitepassword:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'HOST': 'localhost',
        'USER': 'siteuser',
        'PASSWORD': 'sitepassword',
        'NAME': 'site'
    }
}
```

На заметку

В абсолютном большинстве случаев веб-сайты хранят данные в одной базе, поэтому в книге будет описана работа только с одной базой данных. Конфигурирование Django для использования нескольких баз данных описано на странице https://docs.djangoproject.com/en/3.0/topics/db/multi-db/.

3.3.3. Список зарегистрированных приложений

Список приложений, зарегистрированных в проекте, задается параметром INSTALLED_APPS. Все приложения, составляющие проект (написанные самим разработчиком сайта, входящие в состав Django и дополнительных библиотек), должны быть приведены в этом списке.

На заметку

Если приложение содержит только контроллеры, то его можно и не указывать в параметре INSTALLED_APPS. Однако делать так все же не рекомендуется.

Значение этого параметра по умолчанию — пустой список. Однако сразу при создании проекта ему присваивается следующий список:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Он содержит такие стандартные приложения:

- `django.contrib.admin` — административный веб-сайт Django;
- `django.contrib.auth` — встроенная подсистема разграничения доступа. Используется административным сайтом (приложением `django.contrib.admin`);
- `django.contrib.contenttypes` — хранит список всех моделей, объявленных во всех приложениях сайта. Необходимо при создании полиморфных связей между моделями (см. *главу 16*), используется административным сайтом, подсистемой разграничения доступа (приложениями `django.contrib.admin` и `django.contrib.auth`);
- `django.contrib.sessions` — обрабатывает серверные сессии (см. *главу 23*). Требуется при задействовании сессий и используется административным сайтом (приложением `django.contrib.admin`);
- `django.contrib.messages` — выводит всплывающие сообщения (о них будет рассказано в *главе 23*). Требуется для обработки всплывающих сообщений и используется административным сайтом (приложением `django.contrib.admin`);
- `django.contrib.staticfiles` — обрабатывает статические файлы (см. *главу 11*). Необходимо, если в составе сайта имеются статические файлы.

Если какое-либо из указанных приложений не нужно для работы сайта, его можно удалить из этого списка. Также следует убрать используемые удаленным приложением посредники (о них мы скоро поговорим).

3.3.4. Список зарегистрированных посредников

Посредник (middleware) Django — это программный модуль, выполняющий предварительную обработку клиентского запроса перед передачей его контроллеру и окончательную обработку ответа, сгенерированного контроллером, перед его отправкой клиенту.

Список посредников, зарегистрированных в проекте, указывается в параметре `MIDDLEWARE`. Все посредники, используемые в проекте (написанные разработчиком сайта, входящие в состав Django и дополнительных библиотек), должны быть приведены в этом списке.

Значение параметра `MIDDLEWARE` по умолчанию — пустой список. Однако сразу при создании проекта ему присваивается следующий список:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

В нем перечислены стандартные посредники:

- `django.middleware.security.SecurityMiddleware` — реализует дополнительную защиту сайта от сетевых атак;
- `django.contrib.sessions.middleware.SessionMiddleware` — обрабатывает серверные сессии на низком уровне. Используется подсистемами разграничения доступа, сессий и всплывающих сообщений (приложениями `django.contrib.auth`, `django.contrib.sessions` и `django.contrib.messages`);
- `django.middleware.common.CommonMiddleware` — участвует в предварительной обработке запросов;
- `django.middleware.csrf.CsrfViewMiddleware` — осуществляет защиту от межсайтовых атак при обработке данных, переданных сайту HTTP-методом POST;
- `django.contrib.auth.middleware.AuthenticationMiddleware` — добавляет в объект запроса атрибут, хранящий текущего пользователя. Через этот атрибут в контроллерах и шаблонах можно выяснить, какой пользователь выполнил вход на сайт и выполнил ли вообще. Используется административным сайтом и подсистемой разграничения доступа (приложениями `django.contrib.admin` и `django.contrib.auth`);
- `django.contrib.messages.middleware.MessageMiddleware` — обрабатывает всплывающие сообщения на низком уровне. Используется административным сайтом и подсистемой всплывающих сообщений (приложениями `django.contrib.admin` и `django.contrib.messages`);
- `django.middleware.clickjacking.XFrameOptionsMiddleware` — реализует дополнительную защиту сайта от сетевых атак.

Если какой-либо из этих посредников не нужен, его можно удалить из списка. Исключение составляют посредники

`django.middleware.security.SecurityMiddleware`,
`django.middleware.common.CommonMiddleware`,
`django.middleware.clickjacking.XFrameOptionsMiddleware`,
`django.middleware.csrf.CsrfViewMiddleware`

— их удалять не стоит.

3.3.5. Языковые настройки

- LANGUAGE_CODE — код языка, на котором будут выводиться системные сообщения и страницы административного сайта, в виде строки. По умолчанию: "en-us" (американский английский). Для задания русского языка следует указать:

```
LANGUAGE_CODE = 'ru'
```

- USE_I18N — если True, будет активизирована встроенная в Django система автоматического перевода на язык, записанный в параметре LANGUAGE_CODE, после чего все системные сообщения и страницы административного сайта будут выводиться на этом языке. Если False, автоматический перевод выполняться не будет, и сообщения и страницы станут выводиться на английском языке. По умолчанию — True.
- USE_L18N — если True, числа, значения даты и времени при выводе будут форматироваться по правилам языка из параметра LANGUAGE_CODE. Если False, все эти значения будут форматироваться согласно настройкам, заданным в проекте. По умолчанию — False, однако при создании проекта ему дается значение True.
- TIME_ZONE — обозначение временно́й зоны в виде строки. По умолчанию — "America/Chicago". Однако сразу же при создании проекта ему присваивается значение "UTC" (всемирное координированное время). Список всех доступных временны́х зон можно найти по интернет-адресу **https://en.wikipedia.org/wiki/List_of_tz_database_time_zones**.
- USE_TZ — если True, Django будет хранить значения даты и времени с указанием временно́й зоны, в этом случае параметр TIME_ZONE указывает временну́ю зону по умолчанию. Если False, значения даты и времени будут храниться без отметки временно́й зоны, и временну́ю зону для них укажет параметр TIME_ZONE. По умолчанию — False, однако при создании проекта ему дается значение True.

Следующие параметры будут приниматься Django в расчет только в том случае, если отключено автоматическое форматирование выводимых чисел, значений даты и времени (параметру USE_L18N дано значение False):

- DECIMAL_SEPARATOR — символ-разделитель целой и дробной частей вещественных чисел. По умолчанию: "." (точка);
- NUMBER_GROUPING — количество цифр в числе, составляющих группу, в виде целого числа. По умолчанию — 0 (группировка цифр не используется);
- THOUSAND_SEPARATOR — символ-разделитель групп цифр в числах. По умолчанию: "," (запятая);
- USE_THOUSANDS_SEPARATOR — если True, числа будут разбиваться на группы, если False — не будут. По умолчанию — False;
- SHORT_DATE_FORMAT — "короткий" формат значений даты. По умолчанию: "m/d/Y" (*<месяц>/<число>/<год из четырех цифр>*). Для указания формата *<число>.<месяц>.<год из четырех цифр>* следует добавить в модуль settings.py выражение:

```
SHORT_DATE_FORMAT = 'j.m.Y'
```

- SHORT_DATETIME_FORMAT — "короткий" формат временны́х отметок (значений даты и времени). По умолчанию: "m/d/Y P" (*<месяц>/<число>/<год из четырех цифр> <часы в 12-часовом формате>*). Задать формат *<число>.<месяц>.<год из четырех цифр> <часы в 24-часовом формате>:<минуты>:<секунды>* можно, добавив в модуль settings.py выражение:

  ```
  SHORT_DATETIME_FORMAT = 'j.m.Y H:i:s'
  ```

  То же самое, только без секунд:

  ```
  SHORT_DATETIME_FORMAT = 'j.m.Y H:i'
  ```

- DATE_FORMAT — "полный" формат значений даты. По умолчанию: "N j, Y" (*<название месяца по-английски> <число>, <год из четырех цифр>*). Чтобы указать формат *<число> <название месяца> <год из четырех цифр>*, следует добавить в модуль settings.py выражение:

  ```
  DATE_FORMAT = 'j E Y'
  ```

- DATETIME_FORMAT — "полный" формат временны́х отметок. По умолчанию: "N j, Y, P" (*<название месяца по-английски> <число>, <год из четырех цифр> <часы в 12-часовом формате>*). Для указания формата *<число> <название месяца> <год из четырех цифр> <часы в 24-часовом формате>:<минуты>:<секунды>* нужно добавить в модуль settings.py выражение:

  ```
  DATETIME_FORMAT = 'j E Y H:i:s'
  ```

  То же самое, только без секунд:

  ```
  DATETIME_FORMAT = 'j E Y H:i'
  ```

- TIME_FORMAT – формат значений времени. По умолчанию: "P" (*только часы в 12-часовом формате*). Для задания формата *<часы в 24-часовом формате>:<минуты>:<секунды>* достаточно добавить в модуль settings.py выражение:

  ```
  TIME_FORMAT = 'H:i:s'
  ```

  То же самое, только без секунд:

  ```
  TIME_FORMAT = 'H:i'
  ```

- MONTH_DAY_FORMAT — формат для вывода месяца и числа. По умолчанию: "F, j" (*<название месяца по-английски>, <число>*);

- YEAR_MONTH_FORMAT — формат для вывода месяца и года. По умолчанию: "F Y" (*<название месяца по-английски> <год из четырех цифр>*).

  В значениях всех этих параметров применяются специальные символы, используемые в фильтре шаблонизатора `date` (см. *главу 11*);

- DATE_INPUT_FORMATS — список форматов, в которых посетителям допускается заносить значения даты в поля ввода. Получив из веб-формы значение даты в виде строки, Django будет последовательно сравнивать его со всеми форматами, имеющимися в этом списке, пока не найдет подходящий для преобразования строки в дату.

Значение по умолчанию — довольно длинный список форматов, среди которого, к сожалению, нет формата *<число>.<месяц>.<год из четырех цифр>*. Чтобы указать его, следует записать в модуле settings.py выражение:

```
DATE_INPUT_FORMATS = ['%d.%m.%Y']
```

- `DATETIME_INPUT_FORMATS` — список форматов, в которых посетителям допускается заносить временны́е отметки в поля ввода. Получив из веб-формы временну́ю отметку в виде строки, Django будет последовательно сравнивать его со всеми форматами из списка, пока не найдет подходящий для преобразования строки в значение даты и времени.

  Значение по умолчанию — довольно длинный список форматов, среди которого, к сожалению, нет формата *<число>.<месяц>.<год из четырех цифр> <часы в 24-часовом формате>:<минуты>:<секунды>*. Чтобы указать его, следует записать в модуле settings.py выражение:

  ```
  DATETIME_INPUT_FORMATS = ['%d.%m.%Y %H:%M:%S']
  ```

  То же самое, только без секунд:

  ```
  DATETIME_INPUT_FORMATS = ['%d.%m.%Y %H:%M']
  ```

- `TIME_INPUT_FORMATS` — список форматов, в которых посетителям допускается заносить значения времени в поля ввода. Получив из веб-формы значение времени в виде строки, Django будет последовательно сравнивать его со всеми форматами из списка, пока не найдет подходящий для преобразования строки в значение времени.

  Значение по умолчанию: `['%H:%M:%S', '%H:%M:%S.%f', '%H:%M']` (в том числе форматы *<часы в 24-часовом формате>:<минуты>:<секунды>* и *<часы в 24-часовом формате>:<минуты>*).

  В значениях всех этих параметров применяются специальные символы, поддерживаемые функциями `strftime()` и `strptime()` Python (их перечень приведен в документации по Python и на странице **https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior**).

- `FIRST_DAY_OF_WEEK` — номер дня, с которого начинается неделя, в виде целого числа от `0` (воскресенье) до `6` (суббота). По умолчанию: `0` (воскресенье).

Здесь описаны не все параметры проекта, доступные для указания. Остальные параметры, затрагивающие работу отдельных подсистем и касающиеся функционирования сайта в эксплуатационном режиме, мы рассмотрим в последующих главах.

3.4. Создание, настройка и регистрация приложений

Приложения реализуют отдельные части функциональности сайта Django. Любой проект должен содержать по крайней мере одно приложение.

3.4.1. Создание приложений

Создание приложение выполняется командой `startapp` утилиты manage.py, записываемой в формате:

```
manage.py startapp <имя приложения> [<путь к папке пакета приложения>]
```

Если *путь к папке пакета приложения* не указан, то папка пакета приложения с заданным *именем* будет создана в папке проекта. В противном случае в пакет приложения будет преобразована папка, расположенная по указанному *пути*.

3.4.2. Настройка приложений

Модуль apps.py пакета приложения содержит объявление *конфигурационного класса*, хранящего настройки приложения. Код такого класса, задающего параметры приложения `bboard` из *глав 1* и *2*, можно увидеть в листинге 3.1.

Листинг 3.1. Пример конфигурационного класса

```
from django.apps import AppConfig

class BboardConfig(AppConfig):
    name = 'bboard'
```

Он является подклассом класса `AppConfig` из модуля `django.apps` и содержит набор атрибутов класса, задающих немногочисленные параметры приложения:

- `name` — полный путь к пакету приложения, записанный относительно папки проекта, в виде строки. Единственный параметр приложения, обязательный для указания. Задается утилитой manage.py непосредственно при создании приложения, и менять его не нужно;
- `label` — псевдоним приложения в виде строки. Используется в том числе для указания приложения в вызовах утилиты manage.py. Должен быть уникальным в пределах проекта. Если не указан, то в качестве псевдонима принимается последний компонент пути из атрибута `name`;
- `verbose_name` — название приложения, выводимое на страницах административного сайта Django. Если не указан, то будет выводиться псевдоним приложения;
- `path` — файловый путь к папке пакета приложения. Если не указан, то Django определит его самостоятельно.

3.4.3. Регистрация приложения в проекте

Чтобы приложение успешно работало, оно должно быть зарегистрировано в списке приложений `INSTALLED_APPS`, который находится в параметрах проекта (см. *разд. 3.3.3*). Зарегистрировать его можно двумя способами.

1. Добавить в список приложений путь к конфигурационному классу этого приложения, заданному в виде строки:

```
INSTALLED_APPS = [
    . . .
    'bboard.apps.BboardConfig',
]
```

2. В модуле __init__.py пакета приложения присвоить путь к конфигурационному классу переменной `default_app_config`:

```
default_app_config = 'bboard.apps.BboardConfig'
```

После чего добавить в список приложений строку с путем к пакету приложения:

```
INSTALLED_APPS = [
    . . .
    'bboard',
]
```

3.5. Отладочный веб-сервер Django

Запуск отладочного веб-сервера Django выполняется утилитой manage.py при получении ею команды `runserver`, записываемой в формате:

```
manage.py runserver [[<интернет-адрес>:]<порт>] [--noreload]
[--nothreading] [--ipv6] [-6]
```

Отладочный веб-сервер Django самостоятельно отслеживает изменения в программных модулях, при их сохранении сам выполняет перезапуск. Нам об этом беспокоиться не следует.

> ***Внимание!***
>
> Тем не менее в некоторых случаях (в частности, при создании новых модулей или шаблонов) отладочный сервер не перезапускается вообще или перезапускается некорректно, вследствие чего совершенно правильный код перестает работать. Поэтому перед внесением значительных правок в программный код рекомендуется остановить сервер, нажав комбинацию клавиш <Ctrl>+<Break>, а после правок вновь запустить его.
>
> Вообще если сайт стал вести себя странно, притом что код не содержит никаких критических ошибок, прежде всего попробуйте перезапустить отладочный сервер — возможно, после этого все заработает. Автор не раз сталкивался с подобной ситуацией.

По умолчанию отладочный сервер доступен с локального хоста через TCP-порт № 8000 — по интернет-адресам **http://localhost:8000/** и **http://127.0.0.1:8000/**.

Можно указать другой *порт* и, при необходимости, *интернет-адрес*. Например, так задается использование порта № 4000:

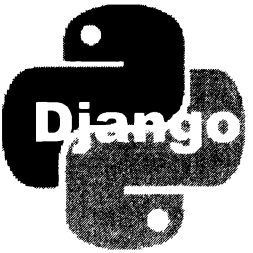
```
manage.py runserver 4000
```

Задаем использование интернет-адреса **1.2.3.4** и порта № 4000:

```
manage.py runserver 1.2.3.4:4000
```

Команда `runserver` поддерживает следующие дополнительные ключи:

- `--noreload` — отключить автоматический перезапуск при изменении программного кода;
- `--nothreading` — принудительный запуск отладочного сервера в однопоточном режиме (если не указан, сервер запускается в многопоточном режиме);
- `--ipv6` или `-6` — работать через протокол IPv6 вместо IPv4. В этом случае по умолчанию будет использоваться интернет-адрес **::1**.



ГЛАВА 4

Модели: базовые инструменты

Следующим шагом после создания и настройки проекта и входящих в него приложений обычно становится объявление моделей.

Модель — это класс, описывающий одну из таблиц в базе данных и предоставляющий инструменты для работы с ней: выборки записей, их фильтрации, сортировки и пр. Отдельный экземпляр класса модели представляет отдельную запись и содержит средства для работы с ней: получения значений полей, записи в них новых значений, добавления, правки и удаления.

4.1. Объявление моделей

Модели объявляются на уровне отдельного приложения, в модуле models.py пакета этого приложения.

Класс модели должен быть производным от класса `Model` из модуля `django.db.models`. Также имеется возможность сделать класс модели производным от другого класса модели (об особенностях объявления производных моделей мы поговорим в *главе 16*).

Чтобы модели были успешно обработаны программным ядром Django, содержащее их приложение должно быть зарегистрировано в списке приложений проекта (см. *разд. 3.4.3*).

Модель может быть создана для представления как еще не существующей в базе таблицы (тогда для ее создания следует сгенерировать миграцию), так и уже имеющейся (в этом случае при объявлении модели придется дополнительно указать имя таблицы и имена всех входящих в нее полей).

4.2. Объявление полей модели

Для представления отдельного поля таблицы в модели создается атрибут класса, которому присваивается экземпляр класса, представляющего поле нужного типа (строкового, целочисленного, логического и т. д.). Дополнительные параметры соз-

даваемого поля указываются в соответствующих им именованных параметрах конструктора класса поля.

4.2.1. Параметры, поддерживаемые полями всех типов

- `verbose_name` — "человеческое" название поля, которое будет выводиться на веб-страницах. Если не указано, то будет выводиться имя поля;
- `help_text` — дополнительный поясняющий текст, выводимый на экран. По умолчанию — пустая строка.

  Содержащиеся в этом тексте специальные символы HTML не преобразуются в литералы и выводятся как есть. Это позволяет отформатировать поясняющий текст HTML-тегами;
- `default` — значение по умолчанию, записываемое в поле, если в него явно не было занесено никакого значения. Может быть указано двумя способами:
  - как обычное значение любого неизменяемого типа:

    ```
    is_active = models.BooleanField(default=True)
    ```

    Если в качестве значения по умолчанию должно выступать значение изменяемого типа (список или словарь Python), то для его указания следует использовать второй способ;
  - как ссылка на функцию, вызываемую при создании каждой новой записи и возвращающую в качестве результата заносимое в поле значение:

    ```
    def is_active_default():
        return not is_all_posts_passive
    . . .
    is_active = models.BooleanField(default=is_active_default)
    ```
- `unique` — если `True`, то в текущее поле может быть занесено только уникальное в пределах таблицы значение (*уникальное поле*). При попытке занести значение, уже имеющееся в том же поле другой записи, будет возбуждено исключение `IntegrityError` из модуля `django.db`.

  Если поле помечено как уникальное, по нему автоматически будет создан индекс. Поэтому явно задавать для него индекс не нужно.

  Если `False`, то текущее поле может хранить любые значения. Значение по умолчанию — `False`;
- `unique_for_date` — если в этом параметре указать представленное в виде строки имя поля даты (`DateField`) или даты и времени (`DateTimeField`), то текущее поле может хранить только значения, уникальные в пределах даты, которая хранится в указанном поле. Пример:

  ```
  title = models.CharField(max_length=50, unique_for_date='published')
  published = models.DateTimeField()
  ```

В этом случае Django позволит сохранить в поле `title` только значения, уникальные в пределах даты, хранящейся в поле `published`;

- `unique_for_month` — то же самое, что и `unique_for_date`, но в расчет принимается не всё значение даты, а лишь месяц;
- `unique_for_year` — то же самое, что и `unique_for_date`, но в расчет принимается не всё значение даты, а лишь год;
- `null` — если `True`, то поле в таблице базы данных может хранить значение `null` и, таким образом, являться необязательным к заполнению. Если `False`, то поле в таблице должно иметь какое-либо значение, хотя бы пустую строку. Значение по умолчанию — `False`.

  Отметим, что у строковых и текстовых полей, даже обязательных к заполнению (т. е. при их объявлении параметру `null` было присвоено значение `False`), вполне допустимое значение — пустая строка. Если сделать поля необязательными к заполнению, задав `True` для параметра `null`, то они вдобавок к этому могут хранить значение `null`. Оба значения фактически представляют отсутствие каких-либо данных в поле, и эту ситуацию придется как-то обрабатывать. Поэтому, чтобы упростить обработку отсутствия значения в таком поле, его не стоит делать необязательным к заполнению.

  Параметр `null` затрагивает только поле таблицы, но не поведение Django. Даже если какое-то поле присвоением параметру значения `True` было помечено как необязательное, фреймворк по умолчанию все равно не позволит занести в него пустое значение;
- `blank` — если `True`, то Django позволит занести в поле пустое значение, тем самым сделав поле необязательным к заполнению, если `False` — не позволит. По умолчанию — `False`.

  Параметр `blank` задает поведение самого фреймворка при выводе на экран веб-форм и проверке введенных в них данных. Если этому параметру дано значение `True`, то Django позволит занести в поле пустое значение (например, для строкового поля — пустую строку), даже если это поле было помечено как обязательное к заполнению (параметру `null` было дано значение `False`);
- `db_index` — если `True`, то по текущему полю в таблице будет создан индекс, если `False` — не будет. По умолчанию — `False`;
- `primary_key` — если `True`, то текущее поле станет ключевым. Такое поле будет помечено как обязательное к заполнению и уникальное (параметру `null` неявно будет присвоено значение `False`, а параметру `unique` — `True`), и по нему будет создан ключевой индекс.

  Внимание!

  В модели может присутствовать только одно ключевое поле.

  Если `False`, то поле не будет преобразовано в ключевое. Значение по умолчанию — `False`.

Если ключевое поле в модели не было задано явно, сам фреймворк создаст в ней целочисленное автоинкрементное ключевое поле с именем `id`;

- `editable` — если `True`, то поле будет выводиться на экран в составе формы, если `False` — не будет (даже если явно создать его в форме). По умолчанию — `True`;
- `db_column` — имя поля таблицы в виде строки. Если не указано, то поле таблицы получит то же имя, что и поле модели.

  Например, в таблице, представляемой моделью `Bb` приложения `bboard` (см. листинг 1.6), будут созданы поля `id` (ключевое поле неявно формируется самим фреймворком), `title`, `content`, `price` и `published`.

Здесь приведены не все параметры, поддерживаемые конструкторами классов полей. Некоторые параметры мы изучим позже.

4.2.2. Классы полей моделей

Все классы полей, поддерживаемые Django, объявлены в модуле `django.db.models`. Каждое такое поле позволяет хранить значения определенного типа. Многие типы полей поддерживают дополнительные параметры, также описанные далее.

- `CharField` — *строковое поле*, хранящее строку ограниченной длины. Занимает в базе данных объем, необходимый для хранения числа символов, указанного в качестве размера этого поля. Поэтому, по возможности, лучше не создавать в моделях строковые поля большой длины.

  Обязательный параметр `max_length` указывает максимальную длину заносимого в поле значения в виде целого числа в символах.

- `TextField` — *текстовое поле*, хранящее строку неограниченной длины. Такие поля рекомендуется применять для сохранения больших текстов, длина которых заранее не известна и может варьироваться.

  Необязательный параметр `max_length` задает максимальную длину заносимого в поле текста. Если не указан, то в поле можно записать значение любой длины.

- `EmailField` — адрес электронной почты в строковом виде.

  Необязательный параметр `max_length` указывает максимальную длину заносимого в поле адреса в виде целого числа в символах. Значение по умолчанию: `254`.

- `URLField` — интернет-адрес.

  Необязательный параметр `max_length` указывает максимальную длину заносимого в поле интернет-адреса в виде целого числа в символах. Значение по умолчанию: `200`.

- `SlugField` — *слаг*, т. е. строка, однозначно идентифицирующая запись и включаемая в состав интернет-адреса. Поддерживает два необязательных параметра:

  - `max_length` — максимальная длина заносимой в поле строки в символах. По умолчанию: `50`;

- `allow_unicode` — если `True`, то хранящийся в поле слаг может содержать любые символы Unicode, если `False` — только символы из кодировки ASCII. По умолчанию — `False`.

Для каждого поля такого типа автоматически создается индекс, поэтому указывать параметр `db_index` со значением `True` нет нужды.

- `BooleanField` — логическое поле, хранящее значение `True` или `False`.

> ***Внимание!***
> Значение поля `BooleanField` по умолчанию — `None`, а не `False`, как можно было бы предположить.

Для поля этого типа можно указать параметр `null` со значением `True`, в результате чего оно получит возможность хранить еще и значение `null`.

- `NullBooleanField` — то же самое, что `BooleanField`, но дополнительно позволяет хранить значение `null`. Этот тип поля оставлен для совместимости с более старыми версиями Django, и использовать его во вновь разрабатываемых проектах не рекомендуется.
- `IntegerField` — знаковое целочисленное поле обычной длины (32-разрядное).
- `SmallIntegerField` — знаковое целочисленное поле половинной длины (16-разрядное).
- `BigIntegerField` — знаковое целочисленное значение двойной длины (64-разрядное).
- `PositiveIntegerField` — беззнаковое целочисленное поле обычной длины (32-разрядное).
- `PositiveSmallIntegerField` — беззнаковое целочисленное поле половинной длины (16-разрядное).
- `FloatField` — вещественное число.
- `DecimalField` — вещественное число фиксированной точности, представленное объектом типа `Decimal` из модуля `decimal` Python. Поддерживает два обязательных параметра:
  - `max_digits` — максимальное количество цифр в числе;
  - `decimal_places` — количество цифр в дробной части числа.

  Пример объявления поля для хранения чисел с шестью цифрами в целой части и двумя — в дробной:

```
price = models.DecimalField(max_digits=8, decimal_places=2)
```

- `DateField` — значение даты в виде объекта типа `date` из модуля `datetime` Python.

  Класс `DateField` поддерживает два необязательных параметра:
  - `auto_now` — если `True`, то при каждом сохранении записи в поле будет заноситься текущее значение даты. Это может использоваться для хранения даты последнего изменения записи.

Если `False`, то хранящееся в поле значение при сохранении записи никак не затрагивается. Значение по умолчанию — `False`;

- `auto_now_add` — то же самое, что `auto_now`, но текущая дата заносится в поле только при создании записи и при последующих сохранениях не изменяется. Может пригодиться для хранения даты создания записи.

Указание значения `True` для любого из этих параметров приводит к тому, что поле становится невидимым и необязательным для заполнения на уровне Django (т. е. параметру `editable` присваивается `False`, а параметру `blank` — `True`).

- `DateTimeField` — то же самое, что и `DateField`, но хранит значение временно́й отметки в виде объекта типа `datetime` из модуля `datetime`.
- `TimeField` — значение времени в виде объекта типа `time` из модуля `datetime` Python. Поддерживаются необязательные параметры `auto_now` и `auto_now_add` (см. описание класса `DateField`).
- `DurationField` — промежуток времени, представленный объектом типа `timedelta` из модуля `datetime` Python.
- `BinaryField` — двоичные данные произвольной длины. Значение этого поля представляется объектом типа `bytes`.
- `GenericIPAddressField` — IP-адрес, записанный для протокола IPv4 или IPv6, в виде строки. Поддерживает два необязательных параметра:
  - `protocol` — обозначение допустимого протокола для записи IP-адресов, представленное в виде строки. Поддерживаются значения `"IPv4"`, `"IPv6"` и `"both"` (поддерживаются оба протокола). По умолчанию — `"both"`;
  - `inpack_ipv4` — если `True`, то IP-адреса протокола IPv4, записанные в формате IPv6, будут преобразованы к виду, применяемому в IPv4. Если `False`, то такое преобразование не выполняется. Значение по умолчанию — `False`. Этот параметр принимается во внимание только в том случае, если для параметра `protocol` указано значение `"both"`.
- `AutoField` — *автоинкрементное поле*. Хранит уникальные, постоянно увеличивающиеся целочисленные значения обычной длины (32-разрядные). Практически всегда используется в качестве ключевого поля.

  Как правило, нет необходимости объявлять такое поле явно. Если модель не содержит явно объявленного ключевого поля любого типа, то Django самостоятельно создаст ключевое поле типа `AutoField`.
- `SmallAutoField` (начиная с Django 3.0) — то же самое, что `AutoField`, но хранит целочисленное значение половинной длины (16-разрядное).
- `BigAutoField` — то же самое, что `AutoField`, но хранит целочисленное значение двойной длины (64-разрядное).
- `UUIDField` — уникальный универсальный идентификатор, представленный объектом типа `UUID` из модуля `uuid` Python, в виде строки.

Поле такого типа может использоваться в качестве ключевого вместо поля `AutoField`, `SmallAutoField` или `BigAutoField`. Единственный недостаток: придется генерировать идентификаторы для создаваемых записей самостоятельно.

Вот пример объявления поля `UUIDField`:

```
import uuid
from django.db import models

class Bb(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4,
                                            editable=False)
    . . .
```

Типы полей, предназначенных для хранения файлов и изображений, будут описаны в *главе 20*.

4.2.3. Создание полей со списком

Поле со списком способно хранить значение из ограниченного набора, заданного в особом перечне. Поле со списком может иметь любой тип, но наиболее часто применяются строковые и целочисленные поля.

В веб-форме поля со списком представляются в виде раскрывающегося списка, перечисляющего все возможные значения для ввода (может быть заменен обычным списком; как это сделать, будет рассмотрено в *главе 13*).

Перечень значений для выбора заносится в параметр `choices` конструктора поля и может быть задан в виде:

- последовательности — списка или кортежа, каждый элемент которого представляет отдельное значение и записывается в виде последовательности из двух элементов:
  - значения, которое будет непосредственно записано в поле (назовем его *внутренним*). Оно должно принадлежать к типу, поддерживаемому полем (так, если поле имеет строковый тип, то значение должно представлять собой строку);
  - значения, которое будет выводиться в виде пункта раскрывающегося списка (*внешнее*), должно представлять собой строку.

Параметру `choices` присваивается непосредственно последовательность с перечнем значений.

Пример задания для поля `kind` списка из трех возможных значений:

```
class Bb(models.Model):
    KINDS = (
        ('b', 'Куплю'),
        ('s', 'Продам'),
        ('c', 'Обменяю'),
    )
```

```
    kind = models.CharField(max_length=1, choices=KINDS, default='s')
    . . .
```

Значения из перечня можно объединять в группы. Каждая группа создается последовательностью из двух элементов: текстового названия группы, выводимого на экран, и последовательности из значений описанного ранее формата.

Для примера разобьем перечень позиций `KINDS` на группы "Купля-продажа" и "Обмен":

```
class Bb(models.Model):
    KINDS = (
        ('Купля-продажа', (
            ('b', 'Куплю'),
            ('s', 'Продам'),
        )),
        ('Обмен', (
            ('c', 'Обменяю'),
        ))
    )
    kind = models.CharField(max_length=1, choices=KINDS, default='s')
    . . .
```

Если поле помечено как необязательное к заполнению на уровне фреймворка (параметру `blank` конструктора присвоено значение `True`) или у него не задано значение по умолчанию (отсутствует параметр `default`), то в списке, перечисляющем доступные для ввода в поле значения, появится пункт `---------` (9 знаков "минус"), обозначающий отсутствие в поле какого-либо значения. Можно задать для этого пункта другое внешнее значение, добавив в последовательность значений элемент вида `(None, '<Новое внешнее значение "пустого" пункта>')` (если поле имеет строковый тип, вместо `None` можно поставить пустую строку). Пример:

```
class Bb(models.Model):
    KINDS = (
        (None, 'Выберите тип публикуемого объявления'),
        ('b', 'Куплю'),
        ('s', 'Продам'),
        ('c', 'Обменяю'),
    )
    kind = models.CharField(max_length=1, choices=KINDS, blank=True)
    . . .
```

- перечисления со строковыми внутренними значениями (теми, что будут непосредственно сохраняться в поле таблицы) — если поле имеет строковый тип (поддерживается, начиная с Django 3.0).

  Перечисление должно являться подклассом класса `TextChoices` из модуля `django.db.models`. Каждый элемент перечисления представляет одно из значений, доступных для выбора. В качестве значения элемента можно указать:

- строку — послужит внутренним значением. В качестве внешнего значения будет использовано имя элемента перечисления;
- кортеж из двух строк — первая станет внутренним значением, вторая — внешним.

Параметру `choices` конструктора поля присваивается значение атрибута `choices` класса перечисления.

Зададим для поля `kind` перечень значений в виде перечисления:

```
class Bb(models.Model):
    class Kinds(models.TextChoices):
        BUY = 'b', 'Куплю'
        SELL = 's', 'Продам'
        EXCHANGE = 'c', 'Обменяю'
        RENT = 'r'

    kind = models.CharField(max_length=1, choices=Kinds.choices,
                                          default=Kinds.SELL)
    . . .
```

Последний элемент перечисления будет выводиться в списке как **Rent**, поскольку у него задано лишь внутреннее значение.

Для представления "пустого" пункта в перечислении следует создать элемент `__empty__` и присвоить ему строку с внешним значением:

```
class Bb(models.Model):
    class Kinds(models.TextChoices):
        BUY = 'b', 'Куплю'
        SELL = 's', 'Продам'
        EXCHANGE = 'c', 'Обменяю'
        RENT = 'r'
        __empty__ = 'Выберите тип публикуемого объявления'
    . . .
```

- перечисления с целочисленными внутренними значениями — если поле имеет один из целочисленных типов (поддерживается, начиная с Django 3.0).

Перечисление должно являться подклассом класса `IntegerChoices` из модуля `django.db.models` (начиная с Django 3.0). В качестве внутренних значений указываются целые числа. В остальном оно аналогично "строковому" перечислению, рассмотренному ранее. Пример:

```
class Bb(models.Model):
    class Kinds(models.IntegerChoices):
        BUY = 1, 'Куплю'
        SELL = 2, 'Продам'
        EXCHANGE = 3, 'Обменяю'
        RENT = 4
```

```
    kind = models.SmallIntegerField(choices=Kinds.choices,
                                    default=Kinds.SELL)
    . . .
```

- перечисления с внутренними значениями произвольного типа — если поле имеет тип, отличный от строкового или целочисленного (появилось в Django 3.0).

  Перечисление должно быть подклассом класса, представляющего тип внутренних значений, и класса `Choices` из модуля `django.db.models`. В начале кортежа, описывающего каждое значение перечня, указываются параметры, передаваемые конструктору класса, что представляет тип внутренних значений. В остальном оно аналогично "строковому" и "целочисленному" перечислениям, рассмотренным ранее.

  Пример перечисления, содержащего внутренние значения в виде вещественных чисел (тип `float`):

```
class Measure(models.Model):
    class Measurements(float, models.Choices):
        METERS = 1.0, 'Метры'
        FEET = 0.3048, 'Футы'
        YARDS = 0.9144, 'Ярды'

    measurement = models.FloatField(choices=Measurements.choices)
    . . .
```

4.3. Создание связей между моделями

Связи между моделями создаются объявлением в них полей, формируемых особыми классами из того же модуля `django.db.models`.

4.3.1. Связь "один-со-многими"

Связь "один-со-многими" связывает одну запись первичной модели с произвольным числом записей вторичной модели. Это наиболее часто применяемый на практике вид связей.

Для создания связи такого типа в классе *вторичной модели* следует объявить поле типа `ForeignKey`. Вот формат конструктора этого класса:

```
ForeingKey(<связываемая первичная модель>,
           on_delete=<поведение при удалении записи>[,
           <остальные параметры>])
```

Первым, позиционным, параметром указывается связываемая первичная модель в виде:

- непосредственно ссылки на класс модели, если объявление первичной модели находится перед объявлением вторичной модели (в которой и создается поле внешнего ключа):

```
class Rubric(models.Model):
    . . .
class Bb(models.Model):
    rubric = models.ForeignKey(Rubric, on_delete=models.PROTECT)
    . . .
```

- строки с именем класса, если первичная модель объявлена после вторичной:

```
class Bb(models.Model):
    rubric = models.ForeignKey('Rubric', on_delete=models.PROTECT)
    . . .
class Rubric(models.Model):
    . . .
```

Ссылка на модель из другого приложения проекта записывается в виде строки формата *<имя приложения>.<имя класса модели>*:

```
rubric = models.ForeingKey('rubrics.Rubric', on_delete=models.PROTECT)
```

Если нужно создать модель, ссылающуюся на себя (создать *рекурсивную связь*), то первым параметром конструктору следует передать строку `self`:

```
parent_rubric = models.ForeingKey('self', on_delete=models.PROTECT)
```

Вторым параметром `on_delete` указывается поведение фреймворка в случае, если будет выполнена попытка удалить запись первичной модели, на которую ссылаются какие-либо записи вторичной модели. Параметру присваивается значение одной из переменных, объявленных в модуле `django.db.models`:

- `CASCADE` — удаляет все связанные записи вторичной модели (*каскадное удаление*);
- `PROTECT` — возбуждает исключение `ProtectedError` из модуля `django.db.models`, тем самым предотвращая удаление записи первичной модели;
- `SET_NULL` — заносит в поле внешнего ключа всех связанных записей вторичной модели значение `null`. Сработает только в том случае, если поле внешнего ключа объявлено необязательным к заполнению на уровне базы данных (параметр `null` конструктора поля имеет значение `True`);
- `SET_DEFAULT` — заносит в поле внешнего ключа всех связанных записей вторичной модели заданное для него значение по умолчанию. Сработает только в том случае, если у поля внешнего ключа было указано значение по умолчанию (оно задается параметром `default` конструктора поля);
- `SET(<значение>)` — заносит в поле внешнего ключа указанное *значение*:

```
rubric = models.ForeingKey(Rubric, on_delete=models.SET(1))
```

Также можно указать ссылку на функцию, не принимающую параметров и возвращающую значение, которое будет записано в поле:

```
def get_first_rubric():
    return Rubric.objects.first()
. . .
rubric = models.ForeingKey(Rubric, on_delete=models.SET(get_first_rubric))
```

- `DO_NOTHING` — ничего не делает.

Внимание!

Если СУБД поддерживает межтабличные связи с сохранением ссылочной целостности, то попытка удаления записи первичной модели, с которой связаны записи вторичной модели, в этом случае все равно не увенчается успехом, и будет возбуждено исключение `IntegrityError` **из модуля** `django.db.models`.

Полю внешнего ключа рекомендуется давать имя, обозначающее связываемую сущность и записанное в единственном числе. Например, для представления рубрики в модели `Bb` мы объявили поле `rubric`.

На уровне базы данных поле внешнего ключа модели представляется полем таблицы, имеющим имя вида *<имя поля внешнего ключа>*`_id`. В веб-форме такое поле будет представляться раскрывающимся списком, содержащим строковые представления записей первичной модели.

Класс `ForeignKey` поддерживает следующие дополнительные необязательные параметры:

- `limit_choices_to` — позволяет вывести в раскрывающемся списке записей первичной модели, отображаемом в веб-форме, только записи, удовлетворяющие заданным критериям фильтрации.

  Критерии фильтрации записываются в виде словаря Python, имена элементов которого совпадают с именами полей первичной модели, по которым должна выполняться фильтрация, а значения элементов укажут значения для этих полей. Выведены будут записи, удовлетворяющие всем критериям, заданным в таком словаре (т. е. критерии объединяются по правилу логического И).

  Для примера укажем Django выводить только рубрики, поле `show` которых содержит значение `True`:

  ```
  rubric = models.ForeignKey(Rubric, on_delete=models.PROTECT,
                                     limit_choices_to={'show': True})
  ```

  В качестве значения параметра также может быть использован экземпляр класса `Q`, задающий сложные критерии фильтрации (класс `Q` мы рассмотрим в *главе 7*).

  Если параметр не указан, то список связываемых записей будет включать все записи первичной модели;

- `related_name` — имя атрибута записи первичной модели, предназначенного для доступа к связанным записям вторичной модели, в виде строки:

  ```
  class Bb(models.Model):
      rubric = models.ForeignKey(Rubric, on_delete=models.PROTECT,
                                         related_name='entries')
  . . .
  # Получаем первую рубрику
  first_rubric = Rubric.objects.first()
  # Получаем доступ к связанным объявлениям через атрибут entries,
  # указанный в параметре related_name
  bbs = first_rubric.entries.all()
  ```

Если доступ из записи первичной модели к связанным записям вторичной модели не требуется, можно указать Django не создавать такой атрибут и тем самым немного сэкономить системные ресурсы. Для этого достаточно присвоить параметру `related_name` символ "плюс".

Если параметр не указан, то атрибут такого рода получит стандартное имя вида *<имя связанной вторичной модели>*`_set`;

- `related_query_name` — имя фильтра, которое будет применяться во вторичной модели для фильтрации по значениям из записи первичной модели:

```
class Bb(models.Model):
    rubric = models.ForeignKey(Rubric, on_delete=models.PROTECT,
                                       related_query_name='entry')
. . .
# Получаем все рубрики, содержащие объявления о продаже домов,
# воспользовавшись фильтром, заданным в параметре related_query_name
rubrics = Rubric.objects.filter(entry__title='Дом')
```

Если параметр не указан, то фильтр такого рода получит стандартное имя, совпадающее с именем класса вторичной модели.

Более подробно работа с записями связанных моделей, применяемые для этого атрибуты и фильтры будут рассмотрены в *главе 7*;

- `to_field` — имя поля первичной модели, по которому будет выполнена связь, в виде строки. Такое поле должно быть помечено как уникальное (параметр `unique` конструктора должен иметь значение `True`).

  Если параметр не указан, связывание выполняется по ключевому полю первичной модели — неважно, созданному явно или неявно;

- `db_constraint` — если `True`, то в таблице базы данных будет создана связь, позволяющая сохранять ссылочную целостность; если `False`, ссылочная целостность будет поддерживаться только на уровне Django.

  Значение по умолчанию — `True`. Менять его на `False` имеет смысл, только если модель создается на основе уже существующей базы с некорректными данными.

4.3.2. Связь "один-с-одним"

Связь "один-с-одним" соединяет одну запись первичной модели с одной записью вторичной модели. Может применяться для объединения моделей, одна из которых хранит данные, дополняющие данные из другой модели.

Такая связь создается в классе *вторичной модели* объявлением поля типа `OneToOneField`. Вот формат конструктора этого класса:

```
OneToOneField(<связываемая первичная модель>,
              on_delete=<поведение при удалении записи>[,
              <остальные параметры>])
```

Первые два параметра точно такие же, как и у конструктора класса `ForeignField` (см. *разд. 4.3.1*).

Для хранения списка зарегистрированных на Django-сайте пользователей стандартная подсистема разграничения доступа использует особую модель. Ссылка на класс заменяемой модели пользователя хранится в параметре `AUTH_USER_MODEL` настроек проекта.

Давайте создадим модель `AdvUser`, хранящую дополнительные сведения о зарегистрированном пользователе. Свяжем ее со стандартной моделью пользователя `User` из модуля `django.contrib.auth.models`. Готовый код класса этой модели приведен в листинге 4.1.

Листинг 4.1. Пример создания связи "один-с-одним"

```
from django.db import models
from django.contrib.auth.models import User

class AdvUser(models.Model):
    is_activated = models.BooleanField(default=True)
    user = models.OneToOneField(User, on_delete=models.CASCADE)
```

На уровне базы данных связь такого рода представляется точно таким же полем, что и связь типа "один-со-многими" (см. *разд. 4.3.1*).

Конструктор класса поддерживает те же необязательные параметры, что и конструктор класса `ForeignField`, плюс параметр `parent_link`, применяемый при наследовании моделей (разговор о котором пойдет в *главе 16*).

4.3.3. Связь "многие-со-многими"

Связь "многие-со-многими" соединяет произвольное число записей одной модели с произвольным числом записей другой (обе модели здесь выступают как равноправные, и определить, какая из них первичная, а какая вторичная, не представляется возможным).

Для создания такой связи нужно объявить *в одной из моделей* (но не в обеих сразу!) поле внешнего ключа типа `ManyToManyField`. Вот формат его конструктора:

```
ManyToManyField(<вторая связываемая модель>[, <остальные параметры>])
```

Первый параметр задается в таком же формате, что и в конструкторах классов `ForeignField` и `OneToOneField` (см. *разд. 4.3.1* и *4.3.2*).

Модель, в которой было объявлено поле внешнего ключа, назовем *ведущей*, а вторую модель — *ведомой*.

Для примера создадим модели `Machine` и `Spare`, из которых первая, ведущая, будет хранить готовые машины, а вторая, ведомая, — отдельные детали для них. Код обеих моделей приведен в листинге 4.2.

Листинг 4.2. Пример создания связи "многие-со-многими"

```
class Spare(models.Model):
    name = models.CharField(max_length=30)

class Machine(models.Model):
    name = models.CharField(max_length=30)
    spares = models.ManyToManyField(Spare)
```

В отличие от связей описанных ранее типов, имя поля, образующего связь "многие-со-многими", рекомендуется записывать во множественном числе. Что и логично — ведь такая связь позволяет связать произвольное число записей, что называется, с обеих сторон.

На уровне базы данных для представления связи такого типа создается таблица, по умолчанию имеющая имя вида *<псевдоним приложения>_<имя класса ведущей модели>_<имя класса ведомой модели>* (*связующая таблица*). Она будет иметь ключевое поле `id` и по одному полю с именем вида *<имя класса связываемой модели>*`_id` на каждую из связываемых моделей. Так, в нашем случае будет создана связующая таблица с именем `samplesite_machine_spare`, содержащая поля `id`, `machine_id` и `spare_id`.

Если создается связь с той же самой моделью, связующая таблица будет иметь поля `id`, `from_`*<имя класса модели>*`_id` и `to_`*<имя класса модели>*`_id`.

Конструктор класса `ManyToManyField` поддерживает дополнительные необязательные параметры `limit_choices_to`, `related_name`, `related_query_name` и `db_constraint`, описанные в *разд. 4.3.1*, а также следующие:

- `symmetrical` — используется только в тех случаях, когда модель связывается сама с собой. Если `True`, Django создаст *симметричную связь*, действующую в обоих направлениях (применительно к нашему случаю: если какая-то деталь А входит в машину Б, то машина Б содержит деталь А). Если `False`, то связь будет *асимметричной* (чисто гипотетически: Иванов любит колбасу, однако колбаса не любит Иванова). Значение по умолчанию — `True`.

  Для асимметричной связи Django создаст в классе модели атрибут для доступа к записям связанной модели в обратном направлении (подробнее — в *главе 7*);

- `through` — класс модели, которая представляет связующую таблицу (*связующая модель*) либо в виде ссылки, либо в виде имени, представленном строкой. Если класс не указан, то связующая таблица будет создана самим Django.

  При использовании связующей модели нужно иметь в виду следующее:

  - поле внешнего ключа для связи объявляется и в ведущей, и в ведомой моделях. При создании этих полей следует указать как саму связующую модель (параметр `through`), так и поля внешних ключей, по которым будет установлена связь (параметр `through_fields`, описанный далее);
  - в связующей модели следует явно объявить поля внешних ключей для установления связи с обеими связываемыми моделями: и ведущей, и ведомой;

- through_fields — используется, если связь устанавливается через связующую модель, записанную в параметре through конструктора. Указывает поля внешних ключей, по которым будет создаваться связь. Значение параметра должно представлять собой кортеж из двух элементов: имени поля ведущей модели и имени поля ведомой модели, записанных в виде строк. Если параметр не указан, то поля будут созданы самим фреймворком.

  Пример использования связующей модели для установления связи "многие-со-многими" и правильного заполнения параметров through и through_fields будет приведен в *главе 16*;

- db_table — имя связующей таблицы. Обычно применяется, если связующая модель не используется. Если оно не указано, то связующая таблица получит имя по умолчанию.

4.4. Параметры самой модели

Параметры самой модели описываются различными атрибутами класса Meta, вложенного в класс модели и не являющегося производным ни от какого класса. Вот список этих атрибутов:

- verbose_name — название сущности, хранящейся в модели, которое будет выводиться на экран. Если не указано, используется имя класса модели;

- verbose_name_plural — название набора сущностей, хранящихся в модели, которая будет выводиться на экран. Если не указано, используется имя класса модели во множественном числе;

- ordering — параметры сортировки записей модели по умолчанию. Задаются в виде последовательности имен полей, по которым должна выполняться сортировка, представленных строками. Если перед именем поля поставить символ "минус", то порядок сортировки будет обратным. Пример:

```
class Bb(models.Model):
    . . .

    class Meta:
        ordering = ['-published', 'title']
```

  Сортируем записи модели сначала по убыванию значения поля published, а потом по возрастанию значения поля title;

- unique_together — последовательность имен полей, представленных в виде строк, которые должны хранить уникальные в пределах таблицы комбинации значений. При попытке занести в них уже имеющуюся в таблице комбинацию значений будет возбуждено исключение ValidationError из модуля django.core.exceptions. Пример:

```
class Bb(models.Model):
    . . .
```

```
    class Meta:
        unique_together = ('title', 'published')
```

Теперь комбинация названия товара и временно́й отметки публикации объявления должна быть уникальной в пределах модели. Добавить в тот же день еще одно объявление о продаже того же товара не получится.

Можно указать несколько подобных групп полей, объединив их в последовательность:

```
class Bb(models.Model):
    . . .

    class Meta:
        unique_together = (
          ('title', 'published'),
          ('title', 'price', 'rubric'),
        )
```

Теперь уникальными должны быть и комбинация названия товара и временно́й отметки публикации, и комбинация названия товара, цены и рубрики;

- `get_latest_by` — имя поля типа `DateField` или `DateTimeField`, которое будет взято в расчет при получении наиболее поздней или наиболее ранней записи с помощью метода `latest()` или `earliest()` соответственно, вызванного без параметров. Можно задать:
  - имя поля в виде строки — тогда в расчет будет взято только это поле:

```
class Bb(models.Model):
    . . .
    published = models.DateTimeField()

    class Meta:
        get_latest_by = 'published'
```

Теперь метод `latest()` вернет запись с наиболее поздним значением временно́й отметки, хранящемся в поле `published`.

Если имя поля предварить символом "минус", то порядок сортировки окажется обратным, и при вызове `latest()` мы получим, напротив, самую раннюю запись, а при вызове метода `earliest()` — самую позднюю:

```
    class Meta:
        get_latest_by = '-published'
```

  - последовательность имен полей — тогда в расчет будут взяты значения всех этих полей, и, если у каких-то записей первое поле хранит одинаковые значения, будет проверяться значение второго поля и т. д.:

```
class Bb(models.Model):
    . . .
    added = models.DateTimeField()
 published = models.DateTimeField()
```

```
    class Meta:
        get_latest_by = ['edited', 'published']
```

- `order_with_respect_to` — позволяет сделать набор записей произвольно упорядочиваемым. В качестве значения параметра задается строка с именем поля текущей модели, и в дальнейшем записи, в которых это поле хранит одно и то же значение, могут быть упорядочены произвольным образом. Пример:

```
class Bb(models.Model):
    . . .
    rubric = models.ForeignKey('Rubric')

    class Meta:
        order_with_respect_to = 'rubric'
```

Теперь объявления, относящиеся к одной и той же рубрике, могут быть переупорядочены произвольным образом.

При указании в модели этого параметра в таблице будет дополнительно создано поле с именем вида *<имя поля, заданного в качестве значения параметра>*`_order`. Оно будет хранить целочисленное значение, указывающее порядковый номер текущей записи в последовательности.

Одновременно вновь созданное поле с порядковым номером будет задано в качестве значения параметра `ordering` (см. ранее). Следовательно, записи, которые мы извлечем из модели, по умолчанию будут отсортированы по значению этого поля. Указать другие параметры сортировки в таком случае будет невозможно;

- `indexes` — последовательность индексов, включающих в себя несколько полей. Каждый элемент такой последовательности должен представлять собой экземпляр класса `Index` из модуля `django.db.models`. Формат конструктора класса:

```
Index(fields=[][, name=None][, condition=None])
```

В параметре `fields` указывается список или кортеж строк с именами полей, которые должны быть включены в индекс. По умолчанию сортировка значений поля выполняется по их возрастанию, а чтобы отсортировать по убыванию, нужно предварить имя поля знаком "минус".

Параметр `name` задает имя индекса — если он не указан, то имя будет создано самим фреймворком. Пример:

```
class Bb(models.Model):
    . . .

    class Meta:
        indexes = [
            models.Index(fields=['-published', 'title'],
                         name='bb_main'),
            models.Index(fields=['title', 'price', 'rubric']),
        ]
```

Начиная с Django 3.0, в имени индекса можно применять заменители `%(app_label)s` и `%(class)s`, обозначающие соответственно псевдоним приложения и имя класса модели. Пример:

```
class Bb(models.Model):
    . . .

    class Meta:
        indexes = [
            models.Index(fields=['-published', 'title'],
                         name='%(app_label)s_%(class)s_main')
        ]
```

Параметр `condition`, поддерживаемый начиная с Django 2.2, задает критерий, которому должны удовлетворять записи, включаемые в индекс (MySQL и MariaDB не поддерживают такие индексы). Критерий записывается с помощью экземпляров класса `Q` (см. *главу 7*). Если он указан, также следует задать имя индекса в параметре `name`.

Пример создания индекса, включающего только товары с ценой менее 10 000:

```
class Bb(models.Model):
    . . .

    class Meta:
        indexes = [
            models.Index(fields=['-published', 'title'],
                         name='bb_partial',
                         condition=models.Q(price__lte=10000))
        ]
```

MySQL и MariaDB не поддерживают подобного рода индексы, и параметр `condition` в их случае игнорируется;

- `index_together` — предлагает другой способ создания индексов, содержащих несколько полей. Строки с именами полей указываются в виде последовательности, а набор таких последовательностей также объединяется в последовательность. Пример:

```
class Bb(models.Model):
    . . .

    class Meta:
        index_together = [
            ['-published', 'title'],
            ['title', 'price', 'rubric'],
        ]
```

Если нужно создать всего один такой индекс, последовательность с именами его полей можно просто присвоить этому параметру:

```
class Bb(models.Model):
    . . .

    class Meta:
        index_together = ['-published', 'title']
```

- `default_related_name` — имя атрибута записи первичной модели, предназначенного для доступа к связанным записям вторичной модели, в виде строки. Соответствует параметру `related_name` конструкторов классов полей, предназначенных для установления связей между моделями (см. *разд. 4.3.1*). Неявно задает значения параметрам `related_name` и `related_query_name` конструкторов;
- `db_table` — имя таблицы, в которой хранятся данные модели. Если оно не указано, то таблица получит имя вида *<псевдоним приложения>_<имя класса модели>* (о псевдонимах приложений рассказывалось в *разд. 3.4.2*).

  Например, для модели `Bb` приложения `bboard` (см. листинг 1.6) в базе данных будет создана таблица `bboard_bb`;
- `constraints` (появился в Django 2.2) — условия, которым должны удовлетворять данные, заносимые в запись. В качестве значения указывается список или кортеж экземпляров классов, каждый из которых задает одно условие. Django включает два таких класса:
  - `CheckConstraint` — критерий, которому должны удовлетворять значения, заносимые в поля модели. Формат конструктора этого класса:

    ```
    CheckConstraint(check=<критерий>, name=<имя условия>)
    ```

    Критерий указывается экземплярами класса `Q` (см. *главу 7*). *Имя условия* задается в виде строки и должно быть уникальным в пределах проекта.

    Пример условия, требующего, чтобы задаваемая в объявлении цена находилась в диапазоне от 0 до 1 000 000:

    ```
    class Bb(models.Model):
        . . .

        class Meta:
            constraints = (
                models.CheckConstraint(check=models.Q(price__gte=0) & \
                       models.Q(price__lte=1000000),
                       name='bboard_rubric_price_constraint'),
            )
    ```

    Начиная с Django 3.0, в *имени условия* можно применять заменители `%(app_label)s` и `%(class)s`, обозначающие соответственно псевдоним приложения и имя класса модели. Пример:

    ```
    class Bb(models.Model):
        . . .
    ```

```
    class Meta:
        constraints = (
            models.CheckConstraint( . . . ,
                   name='%(app_label)s_%(class)s_price_constraint'),
        )
```

Если заносимые в запись значения не удовлетворяют заданным критериям, то при попытке сохранить запись будет возбуждено исключение `IntegrityError` из модуля `django.db`, которое следует обработать программно. Никаких сообщений об ошибках ввода при этом на веб-страницу выведено не будет;

- `UniqueConstraint` — набор полей, которые должны хранить комбинации значений, уникальные в пределах таблицы. Конструктор класса вызывается в таком формате:

```
UniqueConstraint(fields=<набор полей>, name=<имя условия>[,
                                       condition=None])
```

Набор полей задается в виде списка или кортежа, содержащего строки с именами полей. *Имя условия* указывается в виде строки и должно быть уникальным в пределах проекта. Начиная с Django 3.0, в нем можно применять заменители `%(app_label)s` и `%(class)s`, обозначающие соответственно псевдоним приложения и имя класса модели.

Пример условия, требующего уникальности комбинаций из названия товара и его цены:

```
class Bb(models.Model):
    . . .

    class Meta:
        constraints = (
            models.UniqueConstraint(fields=('title', 'price'),
            name='%(app_label)s_%(class)s_title_price_constraint'),
        )
```

Параметр `condition` задает дополнительный критерий, записанный с применением экземпляров класса `Q` (см. *главу 7*). Если он указан, то заданное условие будет применяться только к записям, удовлетворяющим этому критерию.

Пример условия, аналогичного приведенному ранее, но распространяющегося лишь на транспортные средства:

```
class Bb(models.Model):
    . . .

    class Meta:
        constraints = (
            models.UniqueConstraint(fields=('title', 'price'),
            name='%(app_label)s_%(class)s_title_price_constraint',
                  condition=models.Q(price__gte=100000)),
        )
```

Поведение модели при нарушении заданного условия различается в зависимости от того, был ли указан параметр `condition`. Если он не был задан, на странице появится соответствующее сообщение об ошибке. В противном случае сообщение не появится, а будет возбуждено исключение `IntegrityError` из модуля `django.db`, которое следует обработать программно.

В параметре `constraint` может быть указано произвольное число условий, заданных разными классами.

Условия, задаваемые в параметре `constraints`, обрабатываются на уровне СУБД (а не Django), для чего в базе данных создаются соответствующие правила.

4.5. Интернет-адрес модели и его формирование

Django позволяет сформировать интернет-адрес, указывающий на конкретную запись модели, — *интернет-адрес модели*. При переходе по нему может открываться страница с содержимым этой записи, списком связанных записей и др.

Сформировать интернет-адрес модели можно двумя способами: декларативным и императивным.

Декларативный способ заключается в описании формата интернет-адреса в настройках проекта. Набор таких адресов оформляется в виде словаря Python и записывается в параметре `ABSOLUTE_URL_OVERRIDES` модуля **settings.py** пакета конфигурации.

Ключи элементов этого словаря должны иметь вид *<псевдоним приложения>.<имя класса модели>*. Значениями элементов станут функции, в качестве единственного параметра принимающие объект записи модели и возвращающие строку с готовым интернет-адресом. Здесь удобно использовать лямбда-функции Python.

Вот пример объявления словаря, который на основе рубрики (экземпляра класса модели `Rubric`) сформирует адрес вида **bboard/*<ключ рубрики>*/**, ведущий на страницу со списком объявлений, относящихся к этой рубрике:

```
ABSOLUTE_URL_OVERRIDES = {
    'bboard.rubric': lambda rec: "/bboard/%s/" % rec.pk,
}
```

Теперь, чтобы поместить в код шаблона интернет-адрес модели, достаточно вставить туда вызов метода `get_absolute_url()`, унаследованного всеми моделями от базового класса `Model`:

```
<a href="{{ rubric.get_absolute_url }}">{{ rubric.name }}</a>
```

Точно таким же образом можно получить интернет-адрес модели где-либо еще — например, в коде контроллера.

Императивный способ заключается в непосредственном переопределении метода `get_absolute_url(self)` в классе модели. Вот пример:

```
class Rubric(models.Model):
    . . .
    def get_absolute_url(self):
        return "/bboard/%s/" % self.pk
```

Разумеется, в параметре ABSOLUTE_URL_OVERRIDES настроек проекта в таком случае нет нужды.

4.6. Методы модели

Помимо атрибутов класса, представляющих поля модели, и вложенного класса Meta, где объявляются параметры модели, в классе модели можно объявить дополнительные методы:

- __str__(self) — возвращает строковое представление записи модели. Оно будет выводиться, если в коде шаблона указать вывод непосредственно объекта записи, а не значения его поля или результата, возвращенного его методом:

  ```
  {{ rubric }}
  ```

  Пример переопределения этого метода можно найти в *разд. 2.2*;

- save(self, *args, **kwargs) — сохраняет запись. При определении этого метода обязательно следует вставить в нужное место кода вызов метода, унаследованного от базового класса. Вот пример:

  ```
  def save(self, *args, **kwargs):
      # Выполняем какие-либо действия перед сохранением
      super().save(*args, **kwargs)  # Сохраняем запись, вызвав
                                     # унаследованный метод
      # Выполняем какие-либо действия после сохранения
  ```

  В зависимости от выполнения или невыполнения какого-то условия, можно отменить сохранение записи, для чего достаточно просто не вызывать унаследованный метод save(). Пример:

  ```
  def save(self, *args, **kwargs):
      # Выполняем сохранение записи, только если метод is_model_correct()
      # вернет True
      if self.is_model_correct():
          super().save(*args, **kwargs)
  ```

- delete(self, *args, **kwargs) — удаляет запись. Этот метод также переопределяется для добавления какой-либо логики, которая должна выполняться перед удалением и (или) после него. Пример:

  ```
  def delete(self, *args, **kwargs):
      # Выполняем какие-либо действия перед удалением
      super().delete(*args, **kwargs)  # Удаляем запись, вызвав
                                       # унаследованный метод
      # Выполняем какие-либо действия после удаления
  ```

И точно таким же образом в случае необходимости можно предотвратить удаление записи:

```
def delete(self, *args, **kwargs):
    # Удаляем запись, только если метод need_to_delete() вернет True
    if self.need_to_delete():
        super().delete(*args, **kwargs)
```

В модели можно объявить дополнительное поле, значение которого вычисляется на основе каких-то других данных и которое доступно только для чтения (*функциональное поле*). Для этого достаточно объявить метод, не принимающий параметров и возвращающий нужное значение. Имя этого метода станет именем функционального поля.

В качестве примера создадим в модели `Bb` функциональное поле `title_and_price`, объявив одноименный метод:

```
class Bb(models.Model):
    . . .
    def title_and_price(self):
        if self.price:
            return '%s (%.2f)' % (self.title, self.price)
        else:
            return self.title
```

Вот так значение функционального поля выводится в шаблоне:

```
<h2>{{ bb.title_and_price }}</h2>
```

Для функционального поля допускается указать название, которое будет выводиться на веб-страницах. Строку с названием нужно присвоить атрибуту `short_description` объекта метода, который реализует это поле:

```
class Bb(models.Model):
    . . .
    def title_and_price(self):
        . . .
    title_and_price.short_description = 'Название и цена'
```

4.7. Валидация модели. Валидаторы

Валидацией называется проверка на корректность данных, занесенных в поля модели. Валидацию можно реализовать непосредственно в модели или же в форме, которая используется для занесения в нее данных (об этом мы поговорим в *главе 13*).

4.7.1. Стандартные валидаторы Django

Валидацию значений, заносимых в отдельные поля модели, выполняют *валидаторы*, реализованные в виде функций или классов. Некоторые типы полей уже используют определенные валидаторы — так, строковое поле `CharField` задействует

валидатор `MaxLengthValidator`, проверяющий, не превышает ли длина заносимого строкового значения указанную максимальную длину.

Помимо этого, можно указать для любого поля другие валидаторы, предоставляемые Django. Реализующие их классы объявлены в модуле `django.core.validators`. А указываются они в параметре `validators` конструктора класса поля. Пример:

```
from django.core import validators

class Bb(models.Model):
    title = models.CharField(max_length=50,
            validators=[validators.RegexValidator(regex='^.{4,}$')])
```

Здесь использован валидатор, представляемый классом `RegexValidator`. Он проверяет заносимое в поле значение на соответствие заданному регулярному выражению.

Если значение не проходит проверку валидатором, он возбуждает исключение `ValidationError` из модуля `django.core.exceptions`.

В составе Django поставляются следующие классы валидаторов:

- `MinLengthValidator` — проверяет, не меньше ли длина заносимой строки, чем минимум, заданный в первом параметре. Формат конструктора:

  ```
  MinLengthValidator(<минимальная длина>[, message=None])
  ```

  Параметр `message` задает сообщение об ошибке — если он не указан, то выводится стандартное сообщение. Код ошибки: `"min_length"`.

  Начиная с Django 2.2, в качестве первого параметра конструктора можно указать функцию, не принимающую параметров и возвращающую минимальную длину значения в виде целого числа. Пример:

  ```
  def get_min_length():
      # Вычисляем минимальную длину и заносим в переменную min_length
      return min_length

  class Bb(models.Model):
      title = models.CharField(max_length=50, verbose_name='Товар',
              validators=[validators.MinLengthValidator(get_min_length)])
  ```

- `MaxLengthValidator` — проверяет, не превышает ли длина заносимой строки заданный в первом параметре максимум. Используется полем типа `CharField`. Формат конструктора:

  ```
  MaxLengthValidator(<максимальная длина>[, message=None])
  ```

  Параметр `message` задает сообщение об ошибке — если он не указан, используется стандартное. Код ошибки: `"max_length"`.

  Начиная с Django 2.2, в качестве первого параметра конструктора можно указать функцию, не принимающую параметров и возвращающую максимальную длину значения в виде целого числа;

- `RegexValidator` — проверяет значение на соответствие заданному регулярному выражению. Конструктор класса:

```
RegexValidator(regex=None[, message=None][, code=None][,
                          inverse_match=None][, flags=0])
```

Он принимает следующие параметры:

- `regex` — само регулярное выражение. Может быть указано в виде строки или объекта типа `regex`, встроенного в Python;
- `message` — строка с сообщением об ошибке. Если параметр не указан, то выдается стандартное сообщение;
- `code` — код ошибки. Если не указан, используется код по умолчанию: `"invalid"`;
- `inverse_match` — если `False`, значение должно соответствовать регулярному выражению (поведение по умолчанию). Если `True`, то значение, напротив, не должно соответствовать регулярному выражению;
- `flag` — флаги регулярного выражения. Используется, только если таковое задано в виде строки;

- `EmailValidator` — проверяет на корректность заносимый в поле адрес электронной почты. Используется полем типа `EmailField`. Конструктор класса:

```
EmailValidator([message=None][,][code=None][,][, whitelist=None])
```

Параметры:

- `message` — строка с сообщением об ошибке. Если не указан, то выводится стандартное сообщение;
- `code` — код ошибки. Если не указан, используется код по умолчанию: `"invalid"`;
- `whitelist` — последовательность доменов (представленных в виде строк), которые не будут проверяться валидатором. Если параметр не указан, то в эту последовательность входит только адрес локального хоста **localhost**;

- `URLValidator` — проверяет на корректность заносимый в поле интернет-адрес. Используется полем типа `URLField`. Конструктор класса:

```
URLValidator([schemes=None][,][regex=None][,][message=None][,]
                                             [code=None])
```

Параметры:

- `schemes` — последовательность обозначений протоколов, в отношении которых будет выполняться валидация, в виде строк. Если параметр не указан, то используется последовательность `['http', 'https', 'ftp', 'ftps']`;
- `regex` — регулярное выражение, с которым должен совпадать интернет-адрес. Может быть указано в виде строки или объекта типа `regex`, встроенного в Python. Если отсутствует, то такого рода проверка не проводится;

- `message` — строка с сообщением об ошибке. Если не указан, то выводится стандартное сообщение;
- `code` — код ошибки. Если не указан, используется код по умолчанию: `"invalid"`;

❐ `ProhibitNullCharactersValidator` — проверяет, не содержит ли заносимая строка нулевой символ: `\x00`. Формат конструктора:

```
ProhibitNullCharactersValidator([message=None][,][code=None])
```

Он принимает следующие параметры:

- `message` — строка с сообщением об ошибке. Если не указан, то выдается стандартное сообщение;
- `code` — код ошибки. Если не указан, используется код по умолчанию: `"null_characters_not_allowed"`;

❐ `MinValueValidator` — проверяет, не меньше ли заносимое число заданного в первом параметре минимума. Формат конструктора:

```
MinValueValidator(<минимальное значение>[, message=None])
```

Параметр `message` задает сообщение об ошибке; если он не указан, выводится стандартное. Код ошибки: `"min_value"`.

Начиная с Django 2.2, в качестве первого параметра конструктора можно указать функцию, не принимающую параметров и возвращающую минимальное значение в виде целого числа;

❐ `MaxValueValidator` — проверяет, не превышает ли заносимое число заданный в первом параметре максимум. Формат конструктора:

```
MaxValueValidator(<максимальное значение>[, message=None])
```

Параметр `message` задает сообщение об ошибке — если он не указан, выдается стандартное. Код ошибки: `"max_value"`.

Начиная с Django 2.2, в качестве первого параметра конструктора можно указать функцию, не принимающую параметров и возвращающую максимальное значение в виде целого числа;

❐ `DecimalValidator` — проверяет заносимое вещественное число фиксированной точности, представленное объектом типа `Decimal` из модуля `decimal` Python. Формат конструктора:

```
DecimalValidator(<максимальное количество цифр в числе>,
                 <количество цифр в дробной части>)
```

Коды ошибок:

- `"max_digits"` — если общее количество цифр в числе больше заданного;
- `"max_decimal_places"` — если количество цифр в дробной части больше заданного;

- "max_whole_digits" — если количество цифр в целой части числа больше разности между общим количеством и количеством цифр в дробной части.

Часть валидаторов реализована в виде функций:

- validate_ipv46_address() — проверяет на корректность интернет-адреса протоколов IPv4 и IPv6;
- validate_ipv4_address() — проверяет на корректность интернет-адреса только протокола IPv4;
- validate_ipv6_address() — проверяет на корректность интернет-адреса только протокола IPv6.

  Эти три валидатора используются полем типа GenericIPAddressField;
- int_list_validator() — возвращает экземпляр класса RegexValidator, настроенный на проверку последовательностей целых чисел, которые разделяются указанным символом-разделителем. Формат вызова:

```
int_list_validator([sep=','][,][message=None][,][code='invalid'][,]
                                [allow_negative=False])
```

  Параметры:

  - sep — строка с символом-разделителем. Если не указан, то в качестве разделителя используется запятая;
  - message — строка с сообщением об ошибке. Если не указан, то выдается стандартное сообщение;
  - code — код ошибки. Если не указан, используется код по умолчанию: "invalid";
  - allow_negative — если True, допускаются отрицательные числа, если False — не допускаются (поведение по умолчанию).

Помимо классов и функций, модуль django.core.validators объявляет ряд переменных. Каждая из них хранит готовый объект валидатора, настроенный для определенного применения:

- validate_email — экземпляр класса EmailValidator с настройками по умолчанию;
- validate_slug — экземпляр класса RegexValidator, настроенный на проверку слагов. Допускает наличие в слагах только латинских букв, цифр, символов "минус" и подчеркивания;
- validate_unicode_slug — экземпляр класса RegexValidator, настроенный на проверку слагов. Допускает наличие в слагах только букв в кодировке Unicode, цифр, символов "минус" и подчеркивания;

  Эти два валидатора используются полем типа SlugField;
- validate_comma_separated_integer_list — экземпляр класса RegexValidator, настроенный на проверку последовательностей целых чисел, которые разделены запятыми.

4.7.2. Вывод собственных сообщений об ошибках

Во многих случаях стандартные сообщения об ошибках, выводимые валидаторами, вполне понятны. Но временами возникает необходимость вывести посетителю сообщение, более подходящее ситуации.

Собственные сообщения об ошибках указываются в параметре `error_messages` конструктора класса поля. Значением этого параметра должен быть словарь Python, у которого ключи элементов должны совпадать с кодами ошибок, а значения — задавать сами тексты сообщений.

Вот пример указания для поля `title` модели `Bb` собственного сообщения об ошибке:

```
from django.core import validators

class Bb(models.Model):
    title = models.CharField(max_length=50, verbose_name='Товар',
            validators=[validators.RegexValidator(regex='^.{4,}$')],
            error_messages={'invalid': 'Неправильное название товара'})
    . . .
```

Доступные для указания коды ошибок:

- `"null"` — поле таблицы не может хранить значение `null`, т. е. его следует заполнить;
- `"blank"` — в элемент управления должно быть занесено значение;
- `"invalid"` — неверный формат значения;
- `"invalid_choice"` — в поле со списком заносится значение, не указанное в списке;
- `"unique"` — в поле заносится неуникальное значение, что недопустимо;
- `"unique_for_date"` — в поле заносится значение, неуникальное в пределах даты, что недопустимо;
- `"invalid_date"` — значение даты хоть и введено правильно, но некорректное (например, 35.14.2020);
- `"invalid_time"` — значение времени хоть и введено правильно, но некорректное (например, 25:73:80);
- `"invalid_datetime"` — значение даты и времени хоть и введено правильно, но некорректное (например, 35.14.2020 25:73:80);
- `"min_length"` — длина сохраняемой в поле строки меньше указанного минимума;
- `"max_length"` — длина сохраняемой в поле строки больше указанного максимума;
- `"null_characters_not_allowed"` — сохраняемая строка содержит нулевые символы `\x00`;

- "min_value" — сохраняемое в поле число меньше указанного минимума;
- "max_value" — сохраняемое в поле число больше указанного максимума;
- "max_digits" — общее количество цифр в сохраняемом числе типа Decimal больше заданного;
- "max_decimal_places" — количество цифр в дробной части сохраняемого числа типа Decimal больше заданного;
- "max_whole_digits" — количество цифр в целой части сохраняемого числа типа Decimal больше разности между максимальным общим количеством цифр и количеством цифр в дробной части.

4.7.3. Написание своих валидаторов

Если нужный валидатор отсутствует в стандартном наборе, мы можем написать его самостоятельно, реализовав его в виде функции или класса.

Валидатор, выполненный в виде функции, должен принимать один параметр — значение, которое следует проверить. Если значение некорректно, то функция должна возбудить исключение ValidationError из модуля django.core.exceptions. Возвращать результат она не должна.

Для вызова конструктора класса исключения ValidationError предусмотрен следующий формат:

```
ValidationError(<описание ошибки>[, code=None][, params=None])
```

Первым, позиционным, параметром передается строка с текстовым описанием ошибки, допущенной посетителем при вводе значения. Если в этот текст нужно вставить какое-либо значение, следует использовать заменитель вида %(<*ключ элемента словаря, переданного параметром params*>s).

В параметре code указывается код ошибки. Можно указать подходящий код из числа приведенных в *разд. 4.7.2* или придумать свой собственный.

Разработчики Django настоятельно рекомендуют задавать код ошибки. Однако в этом случае нужно помнить, что текст сообщения об ошибке, для которой был указан код, может быть изменен формой, привязанной к этой модели, или самим разработчиком посредством параметра error_messages конструктора поля. Поэтому, если вы хотите, чтобы заданный вами в валидаторе текст сообщения об ошибке всегда выводился как есть, не указывайте для ошибки код.

В параметре params задается словарь со значениями, которые нужно поместить в текст сообщения об ошибке (он передается первым параметром) вместо заменителей.

Листинг 4.3 показывает код валидатора, реализованного в виде функции validate_even(). Он проверяет, является ли число четным.

Листинг 4.3. Пример валидатора-функции

```
from django.core.exceptions import ValidationError

def validate_even(val):
    if val % 2 != 0:
        raise ValidationError('Число %(value)s нечетное', code='odd',
                              params={'value': val})
```

Этот валидатор указывается для поля точно так же, как и стандартный:

```
class Bb(models.Model):
    . . .
    price = models.FloatField(validators=[validate_even])
```

Если валидатору при создании следует передавать какие-либо параметры, задающие режим его работы, то этот валидатор нужно реализовать в виде класса. Параметры валидатору будут передаваться через конструктор класса, а сама валидация станет выполняться в переопределенном методе `__call__()`. Последний должен принимать с параметром проверяемое значение и возбуждать исключение `ValidationError`, если оно окажется некорректным.

Листинг 4.4 демонстрирует код класса `MinMaxValueValidator`, проверяющего, находится ли заносимое в поле числовое значение в заданном диапазоне. Нижняя и верхняя границы этого диапазона передаются через параметры конструктора класса.

Листинг 4.4. Пример валидатора-класса

```
from django.core.exceptions import ValidationError

class MinMaxValueValidator:
    def __init__(self, min_value, max_value):
        self.min_value = min_value
        self.max_value = max_value

    def __call__(self, val):
        if val < self.min_value or val > self.max_value:
            raise ValidationError('Введенное число должно ' +\
                  'находиться в диапазоне от %(min)s до %(max)s',
                  code='out_of_range',
                  params={'min': self.min_value, 'max': self.max_value})
```

4.7.4. Валидация модели

Может возникнуть необходимость проверить на корректность не значение одного поля, а всю модель (выполнить *валидацию модели*). Для этого достаточно переопределить в классе модели метод `clean(self)`.

Метод не должен принимать параметры и возвращать результат. Единственное, что он обязан сделать, — в случае необходимости возбудить исключение `ValidationError`.

Поскольку некорректные значения могут быть занесены сразу в несколько полей, валидатор должен формировать список ошибок. В этом случае пригодится второй формат конструктора класса `ValidationError`:

```
ValidationError(<список ошибок>)
```

Список ошибок удобнее всего представлять в виде словаря. В качестве ключей элементов указываются имена полей модели, в которые были занесены некорректные значения. В качестве значений этих элементов должны выступать последовательности из экземпляров класса `ValidationError`, каждый из которых представляет одну из ошибок.

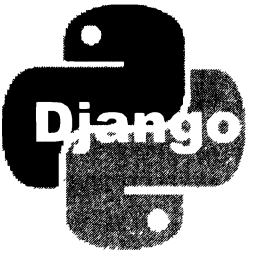
Для примера сделаем так, чтобы занесение описания продаваемого товара было обязательным, и предотвратим ввод отрицательного значения цены. Вот код метода `clean()`, который реализует всё это:

```
class Bb(models.Model):
    . . .
    def clean(self):
        errors = {}
        if not self.content:
            errors['content'] = ValidationError('Укажите описание ' +\
                                                'продаваемого товара')
        if self.price and self.price < 0:
            errors['price'] = ValidationError('Укажите ' +\
                              'неотрицательное значение цены')
        if errors:
            raise ValidationError(errors)
```

Если нужно вывести какое-либо сообщение об ошибке, относящейся не к определенному полю модели, а ко всей модели, то следует использовать в качестве ключа словаря, хранящего список ошибок, значение переменной `NON_FIELD_ERRORS` из модуля `django.core.exceptions`. Пример:

```
from django.core.exceptions import NON_FIELDS_ERRORS
. . .
errors[NON_FIELDS_ERRORS] = ValidationError('Ошибка в модели!')
```

Конструктор класса `ValidationError` поддерживает и прием списка ошибок в виде собственно списка или кортежа Python, однако в этом случае мы не сможем указать Django, к какому полю модели относятся те или иные ошибки, и форма, связанная с моделью, не сможет вывести их напротив соответствующего элемента управления.



ГЛАВА 5

Миграции

Миграция — это программа, создающая в базе данных все необходимые для работы модели структуры (таблицу, поля, индексы, правила и связи) или модифицирующая их, если модель, на основе которой она была сгенерирована, изменилась.

При выполнении миграция генерирует SQL-код, формирующий эти структуры и "понимаемый" СУБД, указанной в настройках проекта (см. *разд. 3.3.2*).

Практически всегда миграции генерируются самим Django по запросу разработчика. Писать свои собственные миграции приходится крайне редко. На случай, если это все же понадобится, — вот интернет-адрес страницы с описанием всех необходимых программных инструментов:

https://docs.djangoproject.com/en/3.0/ref/migration-operations/.

5.1. Генерирование миграций

Для генерирования миграций служит команда `makemigrations` утилиты manage.py:

```
manage.py makemigrations [<список псевдонимов приложений, разделенных пробелами>] [--name|-n <имя миграции>] [--noinput] [--no-input]
[--dry-run] [--check] [--merge] [--empty] [--no-header]
```

Если *псевдонимы приложений* не указаны, то будут обработаны модели из всех приложений проекта. Если указать *псевдоним приложения* или несколько *псевдонимов*, разделив их пробелами, будут обработаны только миграции из указанных приложений.

Команда `makemigrations` поддерживает следующие ключи:

- `--name` или `-n` — указывает имя формируемой миграции, которое будет добавлено к порядковому номеру для получения полного имени файла с кодом миграции. Если оно отсутствует, то миграция получит имя по умолчанию;
- `--noinput` или `--no-input` — отключает вывод на экран сведений о формируемой миграции;

- `--dry-run` — выводит на экран сведения о формируемой миграции, но не формирует ее;
- `--check` — выводит сведения о том, изменились ли модели после последнего формирования миграций, но не формирует саму миграцию;
- `--merge` — используется для устранения конфликтов между миграциями;
- `--empty` — создает "пустую" миграцию для программирования ее вручную. "Пустая" миграция может пригодиться, например, для добавления какого-либо расширения в базу данных PostgreSQL (о расширениях и вообще о работе с базами данных этого формата будет рассказано в *главе 18*);
- `--no-header` (начиная с Django 2.2) — указывает не вставлять в начало модуля миграции комментарий с указанием версии Django и временно́й отметки генерирования миграции.

Результатом выполнения команды станет единственный файл миграции, выполняющий над базой данных все необходимые действия: создание, изменение и удаление таблиц, полей, индексов, связей и правил.

> ***Внимание!***
>
> Django отслеживает любые изменения в коде моделей, даже те, которые не затрагивают структуры базы данных напрямую. Так, если мы укажем для поля название (параметр `verbose_name` конструктора поля), фреймворк все равно создаст в миграции код, который изменит параметры нижележащего поля таблицы в базе данных. Поэтому крайне желательно продумывать структуру моделей заранее и впоследствии, по возможности, не менять ее.
>
> Для отслеживания, какие миграции уже были выполнены, а какие — еще нет, Django создает в базе данных по умолчанию таблицу `django_migrations`. Править вручную как ее структуру, так и хранящиеся в ней записи настоятельно не рекомендуется.

5.2. Файлы миграций

Все программные модули миграций сохраняются в пакете `migrations`, расположенном в пакете приложения. По умолчанию они получают имена формата *<последовательно увеличивающиеся порядковые номера>_<имя миграции>*.py. *Порядковые номера* состоят из четырех цифр и просто помечают очередность, в которой формировались миграции. *Имя миграции* задается в ключе `--name` (`-n`) команды `makemigretions` — если этот ключ не указан, то фреймворк сам сгенерирует имя следующего вида:

- initial — если это *начальная миграция*, т. е. создающая самые первые версии всех необходимых структур в базе данных;
- auto_*<отметка даты и времени формирования миграции>* — если это миграции, сформированные после начальной и дополняющие, удаляющие или изменяющие созданные ранее структуры.

Так, при выполнении упражнений, приведенных в *главах 1* и *2*, у автора книги были сформированы миграции 0001_initial.py и 0002_auto_20191122_1843.py.

Миграции можно переименовывать, но только в том случае, если они до этого еще ни разу не выполнялись. Дело в том, что имена модулей миграций сохраняются в таблице `django_migrations`, и если мы переименуем уже выполненную миграцию, то Django не сможет проверить, была ли она уже выполнена, и выполнит ее снова.

5.3. Выполнение миграций

Выполнение миграций запускается командой `migrate` утилиты manage.py:

```
manage.py migrate [<псевдоним приложения> [<имя миграции>]]
[--fake-initial] [--noinput] [--no-input] [--fake] [--plan]
```

Если не указать ни *псевдоним приложения*, ни *имя миграции*, то будут выполнены все не выполненные к настоящему моменту миграции во всех приложениях проекта. Если указать только *псевдоним приложения*, будут выполнены все миграции в этом приложении, а если дополнительно задать *имя миграции*, то будет выполнена только эта миграция.

Задавать имя модуля миграции полностью нет необходимости — достаточно записать только порядковый номер, находящийся в начале ее имени:

```
manage.py migrate bboard 0001
```

В команде можно применить такие дополнительные ключи:

- `--fake-initial` — пропускает выполнение начальной миграции. Применяется, если в базе данных на момент первого выполнения миграций уже присутствуют все необходимые структуры, и их нужно просто модифицировать;
- `--noinput` или `--no-input` — отключает вывод на экран сведений о применении миграций;
- `--fake` — помечает миграции как выполненные, но не вносит никаких изменений в базу данных. Может пригодиться, если все необходимые изменения в базу были внесены вручную;
- `--plan` (начиная с Django 2.2) — выводит *план* миграций — список, перечисляющий миграции в порядке их выполнения.

На каждую выполненную миграцию в таблицу `django_migrations` базы данных добавляется отдельная запись, хранящая имя модуля миграции, имя приложения, в котором она была создана, дату и время выполнения миграции.

5.4. Слияние миграций

Если в модели неоднократно вносились изменения, после чего на их основе генерировались миграции, то таковых может накопиться довольно много. Чтобы уменьшить количество миграций и заодно ускорить их выполнение на "свежей" базе данных, рекомендуется осуществить *слияние миграций* — объединение их в одну.

Для этого достаточно отдать команду `squashmigrations` утилиты manage.py:

```
manage.py squashmigrations <псевдоним приложения> [<имя первой миграции>]
<имя последней миграции> [--squashed_name <имя результирующей миграции>]
[--no-optimize] [--noinput] [--no-input] [--no-header]
```

Обязательными для указания являются только *псевдоним приложения* и *имя последней миграции* из числа подвергаемых слиянию. В этом случае будут обработаны все миграции, начиная с самой первой из сформированных (обычно это начальная миграция) и заканчивая указанной в команде. Пример:

```
manage.py squashmigrations bboard 0004
```

Если задать *имя первой миграции* из подвергаемых слиянию, то будут обработаны миграции, начиная с нее. Более ранние миграции будут пропущены. Пример:

```
manage.py squashmigrations testapp 0002 0004
```

Рассмотрим ключи, поддерживаемые командой:

- `--squashed_name` — задает имя миграции, которая будет получена в результате слияния. Если оно отсутствует, то модуль результирующей миграции получит имя вида *<имя первой миграции>*_squashed_*<имя последней миграции>*.py;
- `--no-optimize` — отменяет оптимизацию кода миграции, что применяется для уменьшения его объема и повышения быстродействия. Рекомендуется указывать этот ключ, если слияние миграций выполнить не удалось или если результирующая миграция оказалась неработоспособной;
- `--noinput` или `--no-input` — отключает вывод на экран сведений о слиянии миграций;
- `--no-header` (начиная с Django 2.2) — предписывает не вставлять в начало модуля миграции, получаемой в результате слияния, комментарий с указанием версии Django и временно́й отметки слияния.

5.5. Вывод списка миграций

Чтобы просмотреть список всех миграций, имеющихся в проекте, следует отдать команду `showmigrations` утилиты manage.py:

```
manage.py showmigrations [<список псевдонимов приложений, разделенных ↵
пробелами>] [--plan] [-p]
```

Если не указать *псевдоним приложения*, будут выведены все имеющиеся в проекте миграции с разбиением по приложениям. Если указать *псевдоним приложения*, то будут выведены только миграции из этого приложения. При задании *списка псевдонимов приложений, разделенных пробелами*, выводятся только миграции из этих приложений, опять же, с разбиением по отдельным приложениям.

Список миграций при выводе сортируется в алфавитном порядке. Левее имени каждой миграции выводится значок `[X]`, если миграция была выполнена, и `[ ]` — в противном случае.

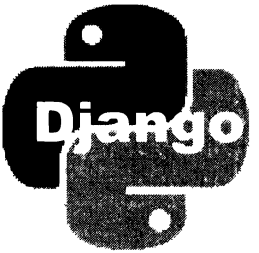
Команда `showmigrations` поддерживает ключи `--plan` и `-p`, указывающие команде вместо списка вывести план миграций. План представляет собой список, отсортированный в последовательности, в которой Django будет выполнять миграции.

5.6. Отмена всех миграций

Наконец, Django позволяет нам отменить все миграции в приложении, тем самым удалив все созданные ими в базе данных структуры. Для этого достаточно выполнить команду `migrate` утилиты manage.py, указав в ней имя нужного приложения и `zero` в качестве имени миграции:

```
manage.py migrate testapp zero
```

К сожалению, отменить отдельную, произвольно выбранную миграцию невозможно.



ГЛАВА 6

Запись данных

Модели призваны упростить работу с данными, хранящимися в информационной базе. В том числе облегчить запись данных в базу.

6.1. Правка записей

Проще всего исправить уже имеющуюся в модели запись. Для этого нужно извлечь ее каким-либо образом (начала чтения данных из моделей мы постигли в *разд. 1.10*):

```
>>> from bboard.models import Bb
>>> b = Bb.objects.get(pk=6)
>>> b
<Bb: Bb object (6)>
```

Здесь мы в консоли Django извлекаем объявление с ключом `6` (это объявление автор создал в процессе отладки сайта, написанного в *главах 1* и *2*).

Занести в поля извлеченной записи новые значения можно, просто присвоив их атрибутам класса модели, представляющим эти поля:

```
>>> b.title = 'Земельный участок'
>>> b.content = 'Большой'
>>> b.price = 100000
```

После этого останется выполнить сохранение записи, вызвав метод `save()` модели:

```
>>> b.save()
```

Поскольку эта запись имеет ключ (ее ключевое поле заполнено), Django сразу узнает, что она уже была ранее сохранена в базе, и выполнит обновление, отправив СУБД SQL-команду `UPDATE`.

6.2. Создание записей

Создать новую запись в модели можно тремя способами:

- создать новый экземпляр класса модели, вызвав конструктор без параметров, занести в поля нужные значения и сохранить запись, вызвав у нее метод `save()`:

```
>>> from bboard.models import Rubric
>>> r = Rubric()
>>> r.name = 'Бытовая техника'
>>> r.save()
```

- создать новый экземпляр класса модели, указав значения полей в вызове конструктора — через одноименные параметры, и сохранить запись:

```
>>> r = Rubric(name='Сельхозинвентарь')
>>> r.save()
```

- все классы моделей поддерживают атрибут `objects`, в котором хранится *диспетчер записей* — объект, представляющий все хранящиеся в модели записи и являющийся экземпляром класса `Manager` из модуля `django.db.models`.

  Класс `Manager` поддерживает метод `create()`, который принимает с именованными параметрами значения полей создаваемой записи, создает эту запись, сразу же сохраняет и возвращает в качестве результата. Вот пример использования этого метода:

```
>>> r = Rubric.objects.create(name='Мебель')
>>> r.pk
5
```

  Удостовериться в том, сохранена ли запись, можно, запросив значение ее ключевого поля (оно всегда доступно через универсальный атрибут класса `pk`). Если оно хранит значение, значит, запись была сохранена.

При создании новой записи любым из описанных ранее способов Django проверяет значение ее ключевого поля. Если таковое хранит пустую строку или `None` (т. е. ключ отсутствует), фреймворк вполне резонно предполагает, что запись нужно добавить в базу, и выполняет ее добавление посылкой СУБД SQL-команды `INSERT`.

Если уж зашла речь о диспетчере записей `Manager`, то нужно рассказать еще о паре полезных методов, которые он поддерживает:

- `get_or_create(`*<набор фильтров>*`[, defaults=None])` — ищет запись на основе заданного *набора фильтров* (о них будет рассказано в *главе 7*). Если подходящая запись не будет найдена, метод создаст и сохранит ее, использовав *набор фильтров* для указания значений полей новой записи.

  Необязательному параметру `defaults` можно присвоить словарь, указывающий значения для остальных полей создаваемой записи (подразумевается, что модель не содержит поля с именем `defaults`. Если же такое поле есть и по нему нужно выполнить поиск, то следует использовать фильтр вида `defaults__exact`).

В качестве результата метод возвращает кортеж из двух значений:

- записи модели, найденной в базе или созданной только что;
- `True`, если эта запись была создана, или `False`, если она была найдена в базе.

Пример:

```
>>> Rubric.objects.get_or_create(name='Мебель')
(<Rubric: Мебель>, False)
>>> Rubric.objects.get_or_create(name='Сантехника')
(<Rubric: Сантехника>, True)
```

> ***Внимание!***
>
> **Метод** `get_or_create()` **способен вернуть только одну запись, удовлетворяющую заданным критериям поиска. Если таких записей в модели окажется более одной, то будет возбуждено исключение** `MultipleObjectsReturned` **из модуля** `django.core.exceptions`.

- `update_or_create(`*<набор фильтров>*`[, defaults=None])` — **аналогичен методу** `get_or_create()`, **но в случае, если запись найдена, заносит в ее поля новые значения, заданные в словаре, который указан в параметре** `defaults`. **Пример:**

```
>>> Rubric.objects.update_or_create(name='Цветы')
(<Rubric: Цветы>, True)
>>> Rubric.objects.update_or_create(name='Цветы',
                                    defaults={'name': 'Растения'})
(<Rubric: Растения>, False)
```

6.3. Занесение значений в поля со списком

В поле со списком (см. *разд. 4.2.3*), в котором перечень доступных значений представлен последовательностью Python, следует заносить внутреннее значение, предназначенное для записи в поле:

```
>>> # Это будет объявление о покупке земельного участка
>>> b.kind = 'b'
>>> b.save()
```

Если же перечень значений представлен перечислением, то в поле нужно заносить элемент перечисления:

```
>>> # Создаем новую запись в модели Measure
>>> m = Measure()
>>> # В качестве единицы измерения указываем футы
>>> m.measurement = Measure.Measurements.FEET
```

6.4. Метод *save()*

Формат вызова метода `save()` модели, сохраняющего запись:

```
save([update_fields=None][,][force_insert=False][,][force_update=False])
```

Необязательный параметр `update_fields` указывает последовательность имен полей модели, которые нужно обновить. Его имеет смысл задавать только при обновлении записи, если были изменены значения не всех, а одного или двух полей, и если поля, не подвергшиеся обновлению, хранят объемистые данные (например, большой текст в поле текстового типа). Пример:

```
>>> b = Bb.objects.get(pk=6)
>>> b.title = 'Земельный участок'
>>> b.save(update_fields=['title'])
```

Если параметр `update_fields` не задан, будут обновлены все поля модели.

Если параметрам `force_insert` и `force_update` задать значение `False`, то Django сам будет принимать решение, создать новую запись или обновить имеющуюся (на основе чего он принимает такое решение, мы уже знаем). Однако мы можем явно указать ему создать или изменить запись, присвоив значение `True` параметру `force_insert` или `force_update` соответственно.

Такое может пригодиться, например, в случае, если какая-то таблица содержит ключевое поле не целочисленного автоинкрементного, а какого-то иного типа, например, строкового. Значение в такое поле при создании записи придется заносить вручную, но при сохранении записи, выяснив, что ключевое поле содержит значение, Django решит, что эта запись уже была сохранена ранее, и попытается обновить ее, что приведет к ошибке. Чтобы исключить такую ситуацию, при вызове метода `save()` следует задать параметр `force_insert` со значением `True`.

Здесь нужно иметь в виду две особенности. Во-первых, задание списка обновляемых полей в параметре `update_fields` автоматически дает параметру `force_update` значение `True` (т. е. явно указывает обновить запись). Во-вторых, указание `True` для обоих описанных ранее параметров вызовет ошибку.

6.5. Удаление записей

Для удаления записи достаточно вызвать у нее метод `delete()`:

```
>>> b = Bb.objects.get(pk=7)
>>> b
<Bb: Bb object (7)>
>>> b.delete()
(1, {'bboard.Bb': 1})
```

Метод `delete()` возвращает в качестве результата кортеж. Его первым элементом станет количество удаленных записей во всех моделях, имеющихся в проекте. Вторым элементом является словарь, в котором ключи элементов представляют отдельные модели, а их значения — количество удаленных из них записей. Особой практической ценности этот результат не представляет.

6.6. Обработка связанных записей

Django предоставляет ряд инструментов для удобной работы со связанными записями: создания, установления и удаления связи.

6.6.1. Обработка связи "один-со-многими"

Связать запись вторичной модели с записью первичной модели можно, присвоив полю внешнего ключа в записи вторичной модели нужный объект-запись первичной модели, например:

```
>>> # Ищем рубрику "Мебель"
>>> r = Rubric.objects.get(name='Мебель')
>>> r
<Rubric: Мебель>
>>> # Создаем объявление о продаже дивана
>>> b = Bb()
>>> b.title = 'Диван'
>>> b.content = 'Продавленный'
>>> b.price = 100
>>> # Указываем у него найденную ранее рубрику "Мебель"
>>> b.rubric = r
>>> b.save()
```

Указать объект-запись первичной модели можно и в вызове метода `create()` диспетчера записей (см. *разд. 6.2*):

```
>>> b = Bb.objects.create(title='Раковина', content='Сильно битая',
                                            price=50, rubric=r)
```

Таким же образом выполняется связывание записи вторичной модели с другой записью первичной таблицы:

```
>>> r2 = Rubric.objects.get(name='Сантехника')
>>> b.rubric = r2
>>> b.save()
>>> b.rubric
<Rubric: Сантехника>
```

Модель, представляющая запись первичной таблицы, получает атрибут с именем вида *<имя связанной вторичной модели>*`_set`. Он хранит экземпляр класса `RelatedManager` из модуля `django.db.models.fields.related`, представляющий набор связанных записей вторичной таблицы и называемый *диспетчером обратной связи*.

> ***Внимание!***
>
> Описанный ранее атрибут класса получает имя *<имя связанной вторичной модели>*`_set` по умолчанию. Однако это имя можно изменить при объявлении поля внешнего ключа, указав его в параметре `related_name` конструктора класса поля (см. *разд. 4.3.1*).

Класс `RelatedManager` поддерживает два очень полезных метода:

- `add(<связываемая запись 1>, <связываемая запись 2> . . . <связываемая запись n> [, bulk=True])` — связывает с текущей записью первичной модели записи вторичной модели, переданные в качестве параметров.

  Если значение параметра `bulk` равно `True`, то записи будут связаны непосредственно отдачей СУБД SQL-команды, без манипуляций с объектами моделей, представляющих связываемые записи. Это поведение по умолчанию, и оно позволяет увеличить производительность.

  Если значение параметра `bulk` равно `False`, то записи будут связываться посредством манипуляций объектами модели, представляющих связываемые записи. Это может пригодиться, если класс модели содержит переопределенные методы `save()` и `delete()`.

  К моменту вызова метода `add()` текущая запись первичной модели должна быть сохранена. Не забываем, что в поле внешнего ключа записи вторичной модели сохраняется ключ записи первичной модели, а он может быть получен только после сохранения записи (если в модели используется стандартное ключевое поле целочисленного автоинкрементного типа).

  Пример:

```
>>> r = Rubric.objects.get(name='Сельхозинвентарь')
>>> b = Bb.objects.create(title='Мотыга', content='Ржавая', price=20)
>>> r.bb_set.add(b)
>>> b.rubric
<Rubric: Сельхозинвентарь>
```

- `create()` — метод унаследован от класса `Manager` и, помимо создания записи вторичной модели, также выполняет ее связывание с текущей записью первичной модели:

```
>>> b2 = r.bb_set.create(title='Лопата', content='Почти новая',
                         price=1000)
>>> b2.rubric
<Rubric: Сельхозинвентарь>
```

6.6.2. Обработка связи "один-с-одним"

Связь такого рода очень проста, соответственно, программных инструментов для ее установления Django предоставляет немного.

Связать записи вторичной и первичной модели можно, присвоив запись первичной модели полю внешнего ключа записи вторичной модели. Вот пример создания записи вторичной модели `AdvUser` (см. листинг 4.1) и связывания ее с записью первичной модели `User`, представляющей пользователя `admin`:

```
>>> from django.contrib.auth.models import User
>>> from testapp.models import AdvUser
>>> u = User.objects.get(username='admin')
```

```
>>> au = AdvUser.objects.create(user=u)
>>> au.user
<User: admin>
>>> u.advuser
<AdvUser: AdvUser object (1)>
```

Первичная модель при этом получит атрибут, хранящий связанную запись вторичной модели. Имя этого атрибута совпадет с именем вторичной модели. Следовательно, связать записи первичной и вторичной модели можно, присвоив запись вторичной модели описанному ранее атрибуту. Вот пример связывания записи модели `User` с другой записью модели `AdvUser`:

```
>>> au2 = AdvUser.objects.get(pk=2)
>>> u.advuser = au2
>>> u.save()
>>> u.advuser
<AdvUser: AdvUser object (2)>
```

6.6.3. Обработка связи "многие-со-многими"

Если между двумя моделями была установлена связь такого рода, то перед собственно связыванием записей нам обязательно нужно их сохранить.

В случае связей "один-со-многими" и "один-с-одним" поле внешнего ключа, объявленное во вторичной модели, всегда хранит непосредственно объект первичной модели, представляющий связанную запись. Но в случае связи "многие-со-многими" это не так — атрибут, представляющий поле, хранит экземпляр класса `RelatedManager` — диспетчер обратной связи.

Для установления связей между записями нужно пользоваться следующими методами этого класса, первые два из которых описывались в *разд. 6.6.1*:

- `add()` — для добавления указанных записей в число связанных с текущей записью:

```
>>> from testapp.models import Spare, Machine
>>> s1 = Spare.objects.create(name='Болт')
>>> s2 = Spare.objects.create(name='Гайка')
>>> s3 = Spare.objects.create(name='Шайба')
>>> s4 = Spare.objects.create(name='Шпилька')
>>> m1 = Machine.objects.create(name='Самосвал')
>>> m2 = Machine.objects.create(name='Тепловоз')
>>> m1.spares.add(s1, s2)
>>> m1.spares.all()
<QuerySet [<Spare: Spare object (1)>, <Spare: Spare object (2)>]>
>>> s1.machine_set.all()
<QuerySet [<Machine: Machine object (1)>]>
>>> m1.spares.add(s4)
>>> m1.spares.all()
<QuerySet [<Spare: Spare object (1)>, <Spare: Spare object (2)>],
<Spare: Spare object (4)>>
```

- `create()` — для создания новых записей связанной модели и одновременного связывания их с текущей записью:

```
>>> m1.spares.create(name='Винт')
<Spare: Spare object (5)>
>>> m1.spares.all()
<QuerySet [<Spare: Spare object (1)>, <Spare: Spare object (2)>,
<Spare: Spare object (4)>], <Spare: Spare object (5)>]>
```

- `set(`*<последовательность связываемых записей>*`[, bulk=True][, clear=False])` — то же самое, что `add()`, но не добавляет указанные записи в число связанных с текущей записью, а заменяет ими те, что были связаны с ней ранее.

  Если значение параметра `bulk` равно `True`, то записи будут связаны непосредственно отдачей СУБД SQL-команды, без манипуляций с объектами моделей, представляющих связываемые записи. Это поведение по умолчанию, и оно позволяет увеличить производительность.

  Если значение параметра `bulk` равно `False`, то записи будут связываться посредством манипуляций объектами модели, представляющих связываемые записи. Это может пригодиться, если класс модели содержит переопределенные методы `save()` и `delete()`.

  Если значение параметра `clear` равно `True`, то Django сначала очистит список связанных записей, а потом свяжет заданные в методе записи с текущей записью. Если же его значение равно `False`, то указанные записи, отсутствующие в списке связанных, будут добавлены в него, а связанные записи, отсутствующие в последовательности указанных в вызове метода, — удалены из списка связанных (поведение по умолчанию).

  Пример:

```
>>> s5 = Spare.objects.get(pk=5)
>>> m1.spares.set([s2, s4, s5])
>>> m1.spares.all()
<QuerySet [<Spare: Spare object (2)>, <Spare: Spare object (4)>,
<Spare: Spare object (5)>]>
```

- `remove(`*<удаляемая запись 1>, <удаляемая запись 2> . . . <удаляемая запись n>*`)` — удаляет указанные записи из списка связанных с текущей записью:

```
>>> m1.spares.remove(s4)
>>> m1.spares.all()
<QuerySet [<Spare: Spare object (2)>, <Spare: Spare object (5)>]>
```

- `clear()` — полностью очищает список записей, связанных с текущей:

```
>>> m2.spares.set([s1, s2, s3, s4, s5])
>>> m2.spares.all()
<QuerySet [<Spare: Spare object (1)>, <Spare: Spare object (2)>,
<Spare: Spare object (3)>, <Spare: Spare object (4)>,
<Spare: Spare object (5)>]>
```

```
>>> m2.spares.clear()
>>> m2.spares.all()
<QuerySet []>
```

6.7. Произвольное переупорядочивание записей

Если у вторичной модели был указан параметр `order_with_respect_to`, то ее записи, связанные с какой-либо записью первичной модели, могут быть произвольно переупорядочены (подробности — в *разд. 4.4*). Для этого следует получить запись первичной модели и вызвать у нее нужный метод:

- `get_<имя вторичной модели>_order()` — возвращает список ключей записей вторичной модели, связанных с текущей записью первичной модели:

```
class Bb(models.Model):
    . . .
    rubric = models.ForeignKey('Rubric')

    class Meta:
        order_with_respect_to = 'rubric'
. . .
>>> r = Rubric.objects.get(name='Мебель')
>>> r.get_bb_order()
[33, 34, 37]
```

- `set_<имя вторичной модели>_order(<список ключей записей вторичной модели>)` — задает новый порядок следования записей вторичной модели. В качестве параметра указывается список ключей записей, в котором ключи должны быть выстроены в нужном порядке. Пример:

```
>>> r.set_bb_order([37, 34, 33])
```

6.8. Массовые добавление, правка и удаление записей

Если возникает необходимость создать, исправить или удалить сразу большое количество записей, то удобнее использовать средства Django для *массовой записи данных*. Это следующие методы класса `Manager` (они также поддерживаются производным от него классом `RelatedManager`):

- `bulk_create(<последовательность добавляемых записей>[, batch_size=None][, ignore_conflicts=False])` — добавляет в модель записи, указанные в *последовательности*.

  Параметр `batch_size` задает количество записей, которые будут добавлены в одной SQL-команде. Если он не указан, все заданные записи будут добавлены в одной команде (поведение по умолчанию).

Параметр `ignore_conflicts` появился в Django 2.2. Если ему задать значение `False`, то при попытке добавить записи, нарушающие заданные в таблице условия (например, содержащие неуникальные значения), будет возбуждено исключение `IntegrityError` (поведение по умолчанию). Если же параметр получит значение `True`, исключения генерироваться не будут, а записи, нарушающие условия, просто не добавятся в таблицу.

> ***Внимание!***
>
> **Некоторые СУБД, в частности Oracle, не поддерживают игнорирование нарушений условий, записанных в таблице, при добавлении записей. В таком случае при попытке вызвать метод `bulk_create` с указанием у параметра `ignore_conflicts` значения `True` будет возбуждено исключение `NotSupportedError` из модуля `django.db.utils`.**

В качестве результата возвращается набор добавленных в модель записей, представленный экземпляром класса `QuerySet`.

> ***Внимание!***
>
> **Метод `bulk_create()` не создает объекты модели, а непосредственно отправляет СУБД SQL-команду, создающую записи. Поэтому в записях, возвращенных им, ключевое поле не заполнено.**
>
> **Исходя из этого, нужно помнить, что метод `save()` у добавляемых таким образом записей не вызывается. Если он переопределен в модели, чтобы осуществить при сохранении какие-либо дополнительные действия, то эти действия выполнены не будут.**

Пример:

```
>>> r = Rubric.objects.get(name='Бытовая техника')
>>> Bb.objects.bulk_create([
        Bb(title='Пылесос', content='Хороший, мощный', price=1000,
                                                       rubric=r),
        Bb(title='Стиральная машина', content='Автоматическая',
                                      price=3000, rubric=r)
    ])
[<Bb: Bb object (None)>, <Bb: Bb object (None)>]
```

- `update(<новые значения полей>)` — **исправляет все записи в наборе, задавая для них** *новые значения полей*. **Эти** *значения* **задаются в параметрах метода, одноименных с нужными полями модели.**

  В качестве результата возвращается количество исправленных записей.

  Метод `save()` у исправляемых записей не вызывается, что может быть критично, если последний переопределен и выполняет какие-либо дополнительные действия.

  Запишем в объявления, в которых указана цена меньше 40 руб., цену 40 руб.:

```
>>> Bb.objects.filter(price__lt=40).update(price=40)
1
```

- `bulk_update(<записи>, <поля>[, batch_size=None])` **(начиная с Django 2.2) — позволяет исправить произвольное число записей.**

Исправляемые записи выбираются из модели, после чего в их поля заносятся новые значения путем присваивания соответствующим атрибутам класса модели (см. *разд. 6.1*). Последовательность исправленных таким образом записей передается методу первым параметром. Вторым параметром указывается последовательность из имен полей, значения которых были исправлены.

Параметр `batch_size` задает число записей, которые будут исправлены в одной SQL-команде. Если он не указан, все заданные записи будут исправлены в одной команде (поведение по умолчанию).

Метод `save()` у исправляемых записей в этом случае также не вызывается.

Исправим цены дачи и дивана на 1 000 000 и 200 соответственно:

```
>>> b1 = Bb.objects.get(title='Дача')
>>> b2 = Bb.objects.get(title='Диван')
>>> b1.price = 1000000
>>> b2.price = 200
>>> Bb.objects.bulk_update((b1, b2), ('price',))
>>> b1.price
1000000
>>> b2.price
2000000
```

- `delete()` — удаляет все записи в наборе. В качестве результата возвращает словарь, аналогичный таковому, возвращаемому методом `delete()` модели (см. *разд. 6.5*).

  Метод `delete()` у удаляемых записей не вызывается, что может быть критично, если последний переопределен.

  Удалим все объявления, в которых не было указано описание товара:

```
>>> Bb.objects.filter(content=None).delete()
(2, {'bboard.Bb': 2})
```

Методы для массовой записи данных работают быстрее, чем программные инструменты моделей, поскольку напрямую "общаются" с базой данных. Однако не забываем, что при их использовании дополнительные операции, выполняемые моделями (в частности, автоматическое получение ключей записей и выполнение методов `save()` и `delete()`), не работают.

6.9. Выполнение валидации модели

Валидация непосредственно модели выполняется редко — обычно это делается на уровне формы, связанной с ней. Но на всякий случай выясним, как провести ее.

Валидацию модели запускает метод `full_clean()`:

```
full_clean([exclude=None][,][validate_unique=True])
```

Параметр `exclude` задает последовательность имен полей, значения которых проверяться не будут. Если он опущен, будут проверяться все поля.

Если параметру `validate_unique` присвоить значение `True`, то при наличии в модели уникальных полей также будет проверяться уникальность заносимых в них значений (поведение по умолчанию). Если значение этого параметра — `False`, такая проверка проводиться не будет.

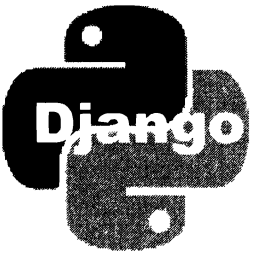
Метод не возвращает никакого результата. Если в модель занесены некорректные данные, то она возбуждает исключение `ValidationError` из модуля `django.core.exceptions`.

В последнем случае в атрибуте `message_dict` модели будет храниться словарь с сообщениями об ошибках. Ключи элементов будут соответствовать полям модели, а значениями элементов станут списки с сообщениями об ошибках.

Примеры:

```
>>> # Извлекаем из модели запись с заведомо правильными данными
>>> # и проверяем ее на корректность. Запись корректна
>>> b = Bb.objects.get(pk=1)
>>> b.full_clean()

>>> # Создаем новую "пустую" запись и выполняем ее проверку. Запись
>>> # некорректна, т. к. в обязательные поля не занесены значения
>>> b = Bb()
>>> b.full_clean()
Traceback (most recent call last):
    raise ValidationError(errors)
django.core.exceptions.ValidationError: {
  'title': ['This field cannot be blank.'],
  'rubric': ['This field cannot be blank.'],
  'content': ['Укажите описание продаваемого товара']
}
```



ГЛАВА 7

Выборка данных

Механизм моделей Django поддерживает развитые средства для выборки данных. Бóльшую их часть мы рассмотрим в этой главе.

7.1. Извлечение значений из полей записи

Получить значения полей записи можно из атрибутов класса модели, представляющих эти поля:

```
>>> from bboard.models import Bb
>>> b = Bb.objects.get(pk=1)
>>> b.title
'Дача'
>>> b.content
'Общество "Двухэтажники". Два этажа, кирпич, свет, газ, канализация'
>>> b.price
1000000.0
```

Атрибут класса `pk` хранит значение ключа текущей записи:

```
>>> b.pk
1
```

Им удобно пользоваться, когда в модели есть явно созданное ключевое поле с именем, отличным от стандартного `id`, — нам не придется вспоминать, как называется это поле.

7.1.1. Получение значений из полей со списком

Если просто обратиться к какому-либо полю со списком, мы получим значение, которое непосредственно хранится в поле, а не то, которое должно выводиться на экран:

```
>>> b = Bb.objects.get(pk=1)
>>> b.kind
's'
```

Чтобы получить "экранное" значение, следует вызвать у модели метод с именем вида `get_<имя поля>_display()`, который это значение и вернет:

```
>>> b.get_kind_display()
'Продам'
```

Это же касается полей, у которых перечень допустимых значений задан в виде объекта последовательности:

```
>>> from bboard.models import Measure
>>> m = Measure.objects.first()
>>> m.measurement
0.3048
>>> m.get_measurement_display()
'Футы'
```

7.2. Доступ к связанным записям

Средства, предназначенные для доступа к связанным записям и создаваемые самим фреймворком, различаются для разных типов связей.

Для связи *"один-со-многими"* из вторичной модели можно получить связанную запись первичной модели посредством атрибута класса, представляющего поле внешнего ключа:

```
>>> b.rubric
<Rubric: Недвижимость>
```

Можно получить значение любого поля связанной записи:

```
>>> b.rubric.name
'Недвижимость'
>>> b.rubric.pk
1
```

В классе первичной модели будет создан атрибут с именем вида *<имя связанной вторичной модели>*`_set`. Он хранит диспетчер обратной связи, представленный экземпляром класса `RelatedManager`, который является производным от класса диспетчера записей `Manager` и, таким образом, поддерживает все его методы.

Диспетчер обратной связи, в отличие от диспетчера записей, манипулирует только записями, связанными с текущей записью первичной модели.

Посмотрим, чем торгуют в рубрике "Недвижимость":

```
>>> from bboard.models import Rubric
>>> r = Rubric.objects.get(name='Недвижимость')
>>> for bb in r.bb_set.all(): print(bb.title)
...
Земельный участок
Дом
Дача
```

Выясним, есть ли там что-нибудь дешевле 10 000 руб.:

```
>>> for bb in r.bb_set.filter(price__lte=10000): print(bb.title)
...
```

Похоже, что ничего...

> ***На заметку***
>
> Имеется возможность задать другое имя для атрибута класса первичной модели, хранящего диспетчер обратной связи. Имя указывается в параметре `related_name` конструктора класса поля:
>
> ```
> class Bb(models.Model):
>     rubric = models.ForeignKey(Rubric, on_delete=models.PROTECT,
>                                related_name='entries')
> . . .
> ```
>
> Теперь мы можем получить доступ к диспетчеру обратной связи по заданному имени:
>
> ```
> >>> for bb in r.entries.all(): print(bb.title)
> ```

В случае связи *"один-с-одним"* всё гораздо проще. Из вторичной модели можно получить доступ к связанной записи первичной модели через атрибут класса, представляющий поле внешнего ключа:

```
>>> from testapp.models import AdvUser
>>> au = AdvUser.objects.first()
>>> au.user
<User: admin>
>>> au.user.username
'admin'
```

Из первичной модели можно получить доступ к связанной записи вторичной модели через атрибут класса, имя которого совпадает с именем вторичной модели:

```
>>> from django.contrib.auth import User
>>> u = User.objects.first()
>>> u.advuser
<AdvUser: AdvUser object (1)>
```

В случае связи *"многие-со-многими"* через атрибут класса ведущей модели, представляющий поле внешнего ключа, доступен диспетчер обратной связи, представляющий набор связанных записей ведомой модели:

```
>>> from testapp.models import Machine
>>> m = Machine.objects.get(pk=1)
>>> m.name
'Самосвал'
>>> for s in m.spares.all(): print(s.name)
...
Гайка
Винт
```

В ведомой модели будет присутствовать атрибут класса *<имя связанной ведущей модели>*`_set`. Его можно использовать для доступа к записям связанной ведущей модели. Пример:

```
>>> from testapp.models import Spare
>>> s = Spare.objects.get(name='Гайка')
>>> for m in s.machine_set.all(): print(m.name)
...
Самосвал
```

7.3. Выборка записей

Теперь выясним, как выполнить выборку из модели записей — как всех, так и лишь тех, которые удовлетворяют определенным условиям.

7.3.1. Выборка всех записей

Все модели поддерживают атрибут класса `objects`. Он хранит диспетчер записей (экземпляр класса `Manager`), который позволяет манипулировать всеми записями, хранящимися в модели.

Метод `all()`, поддерживаемый классом `Manager`, возвращает набор из всех записей модели в виде экземпляра класса `QuerySet`. Последний обладает функциональностью последовательности и поддерживает итерационный протокол. Так что мы можем просто перебрать записи набора и выполнить над ними какие-либо действия в обычном цикле `for...in`. Пример:

```
>>> for r in Rubric.objects.all(): print(r.name, end=' ')
...
Бытовая техника Мебель Недвижимость Растения Сантехника Сельхозинвентарь
Транспорт
```

Класс `RelatedManager` является производным от класса `Manager`, следовательно, тоже поддерживает метод `all()`. Только в этом случае возвращаемый им набор будет содержать лишь связанные записи. Пример:

```
>>> r = Rubric.objects.get(name='Недвижимость')
>>> for bb in r.bb_set.all(): print(bb.title)
...
Земельный участок
Дом
Дача
```

7.3.2. Извлечение одной записи

Ряд методов позволяют извлечь из модели всего одну запись:

- `first()` — возвращает первую запись набора или `None`, если набор пуст:

```
>>> b = Bb.objects.first()
>>> b.title
'Стиральная машина'
```

- `last()` — возвращает последнюю запись набора или `None`, если набор пуст:

```
>>> b = Bb.objects.last()
>>> b.title
'Дача'
```

Оба эти метода учитывают сортировку набора записей, заданную либо вызовом метода `order_by()`, либо в параметре `ordering` модели (см. *разд. 4.4*);

- `earliest([`*<имя поля 1>, <имя поля 2> . . . <имя поля n>*`])` — возвращает запись, у которой значение даты и времени, записанное в полях с указанными именами, является наиболее ранним.

Предварительно выполняется временная сортировка записей по указанным полям. По умолчанию записи сортируются по возрастанию значений этих полей. Чтобы задать сортировку по убыванию, имя поля нужно предварить символом "минус".

Сначала проверяется значение, записанное в поле, имя которого указано первым. Если это значение одинаково у нескольких записей, то проверяется значение следующего поля и т. д.

Если в модели указан параметр `get_latest_by`, задающий поля для просмотра (см. *разд. 4.4*), то метод можно вызвать без параметров.

Если ни одной подходящей записи не нашлось, возбуждается исключение `DoesNotExist`.

Ищем самое раннее из оставленных на сайте объявлений:

```
>>> b = Bb.objects.earliest('published')
>>> b.title
'Дача'
```

А теперь найдем самое позднее, для чего укажем сортировку по убыванию:

```
>>> b = Bb.objects.earliest('-published')
>>> b.title
'Стиральная машина'
```

- `latest([`*<имя поля 1>, <имя поля 2> . . . <имя поля n>*`])` — то же самое, что `earliest()`, но ищет запись с наиболее поздним значением даты и времени:

```
>>> b = Bb.objects.latest('published')
>>> b.title
'Стиральная машина'
```

Все эти методы поддерживаются классами `Manager`, `RelatedManager` и `QuerySet`. Следовательно, мы можем вызывать их также у набора связанных записей:

```
>>> # Найдем самое раннее объявление о продаже транспорта
>>> r = Rubric.objects.get(name='Транспорт')
>>> b = r.bb_set.earliest('published')
>>> b.title
'Мотоцикл'
```

```
>>> # Извлечем самое первое объявление с ценой не менее 10 000 руб.
>>> b = Bb.objects.filter(price__gte=10000).first()
>>> b.title
'Земельный участок'
```

> ***Внимание!***
>
> **Все методы, которые мы рассмотрим далее, также поддерживаются классами `Manager`, `RelatedManager` и `QuerySet`, вследствие чего могут быть вызваны не только у диспетчера записей, но и у диспетчера обратной связи и набора записей.**

7.3.3. Получение числа записей в наборе

Два следующих метода позволят получить число записей, имеющихся в наборе, а также проверить, есть ли там записи:

- `exists()` — возвращает `True`, если в наборе есть записи, и `False`, если набор записей пуст:

```
>>> # Проверяем, продают ли у нас сантехнику
>>> r = Rubric.objects.get(name='Сантехника')
>>> Bb.objects.filter(rubric=r).exists()
True
>>> # А растения?
>>> r = Rubric.objects.get(name='Растения')
>>> Bb.objects.filter(rubric=r).exists()
False
```

- `count()` — возвращает число записей, имеющихся в наборе:

```
>>> # Сколько у нас всего объявлений?..
>>> Bb.objects.count()
11
```

Эти методы выполняются очень быстро, поэтому для проведения простых проверок рекомендуется применять именно их.

7.3.4. Поиск одной записи

Для *поиска* записи по заданным *условиям* служит метод `get(<условия поиска>)`. *Условия поиска* записываются в виде именованных параметров, каждый из которых представляет одноименное поле. Значение, присвоенное такому параметру, задает искомое значение для поля.

Если совпадающая с заданными *условиями* запись нашлась, она будет возвращена в качестве результата. Если ни одной подходящей записи не было найдено, то будет возбуждено исключение `DoesNotExist`, класс которого является вложенным в класс модели, чья запись не была найдена. Если же подходящих записей оказалось несколько, возбуждается исключение `MultipleObjectsReturned` из модуля `django.core.exceptions`.

Пара типичных примеров:

- найдем рубрику "Растения":

```
>>> r = Rubric.objects.get(name='Растения')
>>> r.pk
7
```

- найдем рубрику с ключом 5:

```
>>> r = Rubric.objects.get(pk=5)
>>> r
<Rubric: Мебель>
```

Если в методе `get()` указать сразу несколько условий поиска, то они будут объединяться по правилам логического И. Для примера найдем рубрику с ключом 5 И названием "Сантехника":

```
>>> r = Rubric.objects.get(pk=5, name='Сантехника')
. . .
bboard.models.DoesNotExist: Rubric matching query does not exist.
```

Такой записи нет, и мы получим исключение `DoesNotExist`.

Если в модели есть хотя бы одно поле типа `DateField` или `DateTimeField`, то модель получает поддержку методов с именами вида `get_next_by_`*<имя поля>*`()` и `get_previous_by_`*<имя поля>*`()`. Формат вызова у обоих методов одинаковый:

`get_next_by_`*<имя поля>* | `get_previous_by_`*<имя поля>*`([`*<условия поиска>*`])`

Первый метод возвращает запись, чье поле с указанным *именем* хранит следующее в порядке увеличения значение даты, второй метод — запись с предыдущим значением. Если указаны *условия поиска*, то они также принимаются во внимание. Примеры:

```
>>> b = Bb.objects.get(pk=1)
>>> b.title
'Дача'
>>> # Найдем следующее в хронологическом порядке объявление
>>> b2 = b.get_next_by_published()
>>> b2.title
'Дом'
>>> # Найдем следующее объявление, в котором заявленная цена меньше 1 000 руб.
>>> b3 = b.get_next_by_published(price__lt=1000)
>>> b3.title
'Диван'
```

Если текущая модель является вторичной, и у нее было задано произвольное переупорядочивание записей, связанных с одной и той же записью первичной модели (т. е. был указан параметр `order_with_respect_to`, описанный в *разд. 4.4*), то эта вторичная модель получает поддержку методов `get_next_in_order()` и `get_previous_in_order()`. Они вызываются у какой-либо записи вторичной модели: пер-

вый метод возвращает следующую в установленном порядке запись, а второй — предыдущую. Пример:

```
>>> r = Rubric.objects.get(name='Мебель')
>>> bb2 = r.bb_set.get(pk=34)
>>> bb2.pk
34
>>> bb1 = bb2.get_previous_in_order()
>>> bb1.pk
37
>>> bb3 = bb2.get_next_in_order()
>>> bb3.pk
33
```

7.3.5. Фильтрация записей

Для *фильтрации* записей Django предусматривает два следующих метода — диаметральные противоположности друг друга:

- `filter(<условия фильтрации>)` — отбирает из текущего набора только записи, удовлетворяющие заданным *условиям фильтрации*. *Условия фильтрации* задаются точно в таком же формате, что и условия поиска в вызове метода `get()` (см. *разд. 7.3.4*). Пример:

```
>>> # Отбираем только объявления с ценой не менее 10 000 руб.
>>> for b in Bb.objects.filter(price__gte=10000):
        print(b.title, end=' ')
...
Земельный участок Велосипед Мотоцикл Дом Дача
```

- `exclude(<условия фильтрации>)` — то же самое, что `filter()`, но, наоборот, отбирает записи, *не* удовлетворяющие заданным условиям фильтрации:

```
>>> # Отбираем все объявления, кроме тех, в которых указана цена
>>> # не менее 10 000 руб.
>>> for b in Bb.objects.exclude(price__gte=10000):
        print(b.title, end=' ')
...
Стиральная машина Пылесос Лопата Мотыга Софа Диван
```

Поскольку оба эти метода поддерживаются классом `QuerySet`, мы можем "сцеплять" их вызовы друг с другом. Для примера найдем все объявления о продаже недвижимости с ценой менее 1 000 000 руб.:

```
>>> r = Rubric.objects.get(name='Недвижимость')
>>> for b in Bb.objects.filter(rubric=r).filter(price__lt=1000000):
        print(b.title, end=' ')
...
Земельный участок
```

Впрочем, такой запрос на фильтрацию можно записать и в одном вызове метода `filter()`:

```
>>> for b in Bb.objects.filter(rubric=r, price__lt=1000000):
        print(b.title, end=' ')
...
Земельный участок
```

7.3.6. Написание условий фильтрации

Если в вызове метода `filter()` или `exclude()` записать условие фильтрации в формате *<имя поля>=<значение поля>*, то Django будет отбирать записи, у которых значение заданного *поля* точно совпадает с указанной величиной *значения*, причем в случае строковых и текстовых полей сравнение выполняется с учетом регистра.

Но что делать, если нужно выполнить сравнение без учета регистра или отобрать записи, у которых значение поля больше или меньше заданной величины? Использовать *модификаторы*. Такой модификатор добавляется к имени поля и отделяется от него двойным символом подчеркивания: *<имя поля>__<модификатор>*. Все поддерживаемые Django модификаторы приведены в табл. 7.1.

***Таблица 7.1.** Модификаторы*

Модификатор	Описание
`exact`	Точное совпадение значения с учетом регистра символов. Получаемый результат аналогичен записи *<имя поля>=<значение поля>*. Применяется в случаях, если имя какого-либо поля совпадает с ключевым словом Python: `class__exact='superclass'`
`iexact`	Точное совпадение значения без учета регистра символов
`contains`	Заданное значение должно присутствовать в значении, хранящемся в поле. Регистр символов учитывается
`icontains`	Заданное значение должно присутствовать в значении, хранящемся в поле. Регистр символов не учитывается
`startswith`	Заданное значение должно присутствовать в начале значения, хранящегося в поле. Регистр символов учитывается
`istartswith`	Заданное значение должно присутствовать в начале значения, хранящегося в поле. Регистр символов не учитывается
`endswith`	Заданное значение должно присутствовать в конце значения, хранящегося в поле. Регистр символов учитывается
`iendswith`	Заданное значение должно присутствовать в конце значения, хранящегося в поле. Регистр символов не учитывается
`lt`	Значение, хранящееся в поле, должно быть меньше заданного
`lte`	Значение, хранящееся в поле, должно быть меньше заданного или равно ему
`gt`	Значение, хранящееся в поле, должно быть больше заданного
`gte`	Значение, хранящееся в поле, должно быть больше заданного или равно ему

Таблица 7.1 (продолжение)

Модификатор	Описание
range	Значение, хранящееся в поле, должно находиться внутри заданного диапазона, включая его начальное и конечное значения. Диапазон значений задается в виде кортежа, первым элементом которого записывается начальное значение, вторым — конечное
date	Значение, хранящееся в поле, рассматривается как дата. Пример: `published__date=datetime.date(2018, 6, 1)`

year	Из значения даты, хранящегося в поле, извлекается год, и дальнейшее сравнение выполняется с ним. Примеры:
	`published__year=2018`
	`published__year__lte=2017`

iso_year	Из значения даты, хранящегося в поле, извлекается год в формате ISO 8601, и дальнейшее сравнение выполняется с ним. Поддерживается, начиная с Django 2.2.
month	Из значения даты, хранящегося в поле, извлекается номер месяца, и дальнейшее сравнение выполняется с ним
day	Из значения даты, хранящегося в поле, извлекается число, и дальнейшее сравнение выполняется с ним
week	Из значения даты, хранящегося в поле, извлекается номер недели, и дальнейшее сравнение выполняется с ним
week_day	Из значения даты, хранящегося в поле, извлекается номер дня недели, и дальнейшее сравнение выполняется с ним
quarter	Из значения даты, хранящегося в поле, извлекается номер квартала года (от 1 до 4), и дальнейшее сравнение выполняется с ним
time	Значение, хранящееся в поле, рассматривается как время. Пример: `published__time=datetime.time(12, 0)`

hour	Из значения времени, хранящегося в поле, извлекаются часы, и дальнейшее сравнение выполняется с ними. Примеры:
	`published__hour=12`
	`published__hour__gte=13`

minute	Из значения времени, хранящегося в поле, извлекаются минуты, и дальнейшее сравнение выполняется с ними
second	Из значения времени, хранящегося в поле, извлекаются секунды, и дальнейшее сравнение выполняется с ними
isnull	Если `True`, то указанное поле должно хранить значение `null` (быть пустым). Если `False`, то поле должно хранить значение, отличное от `null` (быть заполненным). Пример: `content__isnull=False`
in	Значение, хранящееся в поле, должно присутствовать в указанном списке, кортеже или наборе записей `QuerySet`. Пример: `pk__in=(1, 2, 3, 4)`
regex	Значение, хранящееся в поле, должно совпадать с заданным регулярным выражением. Регистр символов учитывается. Пример: `content__regex='газ|вода'`

Таблица 7.1 (окончание)

Модификатор	Описание
iregex	Значение, хранящееся в поле, должно совпадать с заданным регулярным выражением. Регистр символов не учитывается

7.3.7. Фильтрация по значениям полей связанных записей

Чтобы выполнить фильтрацию записей вторичной модели по значениям полей из первичной модели, условие фильтрации записывается в формате *<имя поля внешнего ключа>__<имя поля первичной модели>*. Для примера выберем все объявления о продаже транспорта:

```
>>> for b in Bb.objects.filter(rubric__name='Транспорт'):
        print(b.title, end=' ')
...
Велосипед Мотоцикл
```

Для выполнения фильтрации записей первичной модели по значениям из полей вторичной модели следует записать условие вида *<имя вторичной модели>__<имя поля вторичной модели>*. В качестве примера выберем все рубрики, в которых есть объявления о продаже с заявленной ценой более 10 000 руб.:

```
>>> for r in Rubric.objects.filter(bb__price__gt=10000):
        print(r.name, end=' ')
...
Недвижимость Недвижимость Недвижимость Транспорт Транспорт
```

Мы получили три одинаковые записи "Недвижимость" и две — "Транспорт", поскольку в этих рубриках хранятся, соответственно, три и два объявления, удовлетворяющие заявленным условиям фильтрации. О способе выводить только уникальные записи мы узнаем позже.

Как видим, в подобного рода условиях фильтрации мы можем применять и модификаторы.

На заметку

Имеется возможность назначить другой фильтр, который будет применяться вместо имени вторичной модели в условиях такого рода. Он указывается в параметре `related_query_name` **конструктора класса поля:**

```
class Bb(models.Model):
    rubric = models.ForeignKey(Rubric, on_delete=models.PROTECT,
                               related_query_name='entry')
```

После этого мы можем записать рассмотренное ранее выражение в таком виде:

```
>>> for r in Rubric.objects.filter(entry__price__gt=10000):
        print(r.name, end=' ')
```

Все это касалось связей "один-со-многими" и "один-с-одним". А что же связи "многие-со-многими"? В них действуют те же самые правила. Убедимся сами:

```
>>> # Получаем все машины, в состав которых входят гайки
>>> for m in Machine.objects.filter(spares__name='Гайка'):
        print(m.name, end=' ')
...
Самосвал
>>> # Получаем все детали, входящие в состав самосвала
>>> for s in Spare.objects.filter(machine__name='Самосвал'):
        print(s.name, end=' ')
...
Гайка Винт
```

7.3.8. Сравнение со значениями других полей

До этого момента при написании условий фильтрации мы сравнивали значения, хранящиеся в полях, с константами. Но иногда бывает необходимо сравнить значение из одного поля записи со значением другого ее поля.

Для этого предназначен класс с именем `F`, объявленный в модуле `django.db.models`. Вот формат его конструктора:

`F(`*<имя поля модели, с которым должно выполняться сравнение>*`)`

Имя поля модели записывается в виде строки.

Получив экземпляр этого класса, мы можем использовать его в правой части любого условия.

Вот пример извлечения объявлений, в которых название товара встречается в тексте его описания:

```
>>> from django.db.models import F
>>> f = F('title')
>>> for b in Bb.objects.filter(content__icontains=f):
        print(b.title, end=' ')
```

Экземпляры класса `F` можно использовать не только при фильтрации, но и для занесения нового значения в поля модели. Например, так можно уменьшить цены во всех объявлениях вдвое:

```
>>> f = F('price')
>>> for b in Bb.objects.all():
        b.price = f / 2
        b.save()
```

7.3.9. Сложные условия фильтрации

Класс `Q` из модуля `django.db.models` позволяет создавать более сложные условия фильтрации. Его конструктор записывается в следующем формате:

`Q(`*<условие фильтрации>*`)`

Условие фильтрации, одно-единственное, записывается в таком же виде, как и в вызовах методов `filter()` и `exclude()`.

Два экземпляра класса `Q`, хранящие разные условия, можно объединять посредством операторов `&` и `|`, которые обозначают соответственно логическое И и ИЛИ. Для выполнения логического НЕ применяется оператор `~`. Все эти три оператора в качестве результата возвращают новый экземпляр класса `Q`.

Пример выборки объявлений о продаже ИЛИ недвижимости, ИЛИ бытовой техники:

```
>>> from django.db.models import Q
>>> q = Q(rubric__name='Недвижимость') | \
        Q(rubric__name='Бытовая техника')
>>> for b in Bb.objects.filter(q): print(b.title, end=' ')
...
Пылесос Стиральная машина Земельный участок Дом Дача
```

Пример выборки объявлений о продаже транспорта, в которых цена НЕ больше 45 000 руб.:

```
>>> q = Q(rubric__name='Транспорт') & ~Q(price__gt=45000)
>>> for b in Bb.objects.filter(q): print(b.title, end=' ')
...
Велосипед
```

7.3.10. Выборка уникальных записей

В *разд. 7.3.7* мы рассматривали пример фильтрации записей первичной модели по значениям полей из вторичной модели и получили в результате набор из пяти записей, три из которых повторялись.

Для вывода только уникальных записей служит метод `distinct()`:

```
distinct([<имя поля 1>, <имя поля 2> . . . <имя поля n>])
```

При использовании СУБД PostgreSQL в вызове метода можно перечислить *имена полей*, значения которых определят уникальность записей. Если не задавать параметров, то уникальность каждой записи будет определяться значениями всех ее полей.

Перепишем пример из *разд. 7.3.7*, чтобы он выводил только уникальные записи:

```
>>> for r in Rubric.objects.filter(bb__price__gt=10000).distinct():
        print(r.name, end=' ')
...
Недвижимость Транспорт
```

7.3.11. Выборка указанного числа записей

Для извлечения указанного числа записей применяется оператор взятия среза `[]` Python, записываемый точно так же, как и в случае использования обычных последовательностей. Единственное исключение — не поддерживаются отрицательные индексы.

Вот три примера:

```
>>> # Извлекаем первые три рубрики
>>> Rubric.objects.all()[:3]
<QuerySet [<Rubric: Бытовая техника>, <Rubric: Мебель>,
<Rubric: Недвижимость>]>

>>> # Извлекаем все рубрики, начиная с шестой
>>> Rubric.objects.all()[5:]
<QuerySet [<Rubric: Сельхозинвентарь>, <Rubric: Транспорт>]>

>>> # Извлекаем третью и четвертую рубрики
>>> Rubric.objects.all()[2:4]
<QuerySet [<Rubric: Недвижимость>, <Rubric: Растения>]>
```

7.4. Сортировка записей

Для сортировки записей в наборе применяется метод `order_by()`:

```
order_by([<имя поля 1>, <имя поля 2> . . . <имя поля n>])
```

В качестве параметров указываются *имена полей* в виде строк. Сначала сортировка выполняется по значению первого поля. Если у каких-то записей оно хранит одно и то же значение, проводится сортировка по второму полю и т. д.

По умолчанию сортировка выполняется по возрастанию значения поля. Чтобы отсортировать по убыванию значения, следует предварить имя поля знаком "минус".

Пара примеров:

```
>>> # Сортируем рубрики по названиям
>>> for r in Rubric.objects.order_by('name'): print(r.name, end=' ')
...
Бытовая техника Мебель Недвижимость Растения Сантехника Сельхозинвентарь
Транспорт

>>> # Сортируем объявления сначала по названиям рубрик, а потом
>>> # по убыванию цены
>>> for b in Bb.objects.order_by('rubric__name', '-price'):
        print(b.title, end=' ')
...
Стиральная машина Пылесос Диван Дом Дача Земельный участок Софа Лопата Мотыга
Мотоцикл Велосипед
```

Каждый вызов метода `order_by()` отменяет параметры сортировки, заданные в его предыдущем вызове или в параметрах модели (атрибут `ordering` вложенного класса `Meta`). Поэтому, если записать:

```
Bb.objects.order_by('rubric__name').order_by('-price')
```

объявления будут отсортированы только по убыванию цены.

Если передать методу `order_by()` в качестве единственного параметра строку `'?'`, то записи будут выстроены в случайном порядке. Однако это может отнять много времени.

Вызов метода `reverse()` меняет порядок сортировки записей на противоположный:

```
>>> for r in Rubric.objects.order_by('name').reverse():
        print(r.name, end=' ')
...
Транспорт Сельхозинвентарь Сантехника Растения Недвижимость Мебель
Бытовая техника
```

Чтобы отменить сортировку, заданную предыдущим вызовом метода `order_by()` или в самой модели, следует вызвать метод `order_by()` без параметров.

7.5. Агрегатные вычисления

Агрегатные вычисления затрагивают значения определенного поля всех записей, имеющихся в модели, или групп записей, удовлетворяющих какому-либо условию. К такого рода действиям относится вычисление числа объявлений, среднего арифметического цены, наименьшего и наибольшего значения цены и т. п.

Каждое из возможных действий, выполняемых при агрегатных вычислениях, представляется определенной *агрегатной функцией*. Так, существуют агрегатные функции для подсчета числа записей, среднего арифметического, минимума, максимума и др.

7.5.1. Вычисления по всем записям модели

Если нужно провести агрегатное вычисление по всем записям модели, нет ничего лучше метода `aggregate()`:

```
aggregate(<агрегатная функция 1>, <агрегатная функция 2> . . .
                                  <агрегатная функция n>)
```

Сразу отметим два момента. Во-первых, *агрегатные функции* представляются экземплярами особых классов, которые объявлены в модуле `django.db.models` и которые мы рассмотрим чуть позже. Во-вторых, возвращаемый методом результат — словарь Python, в котором отдельные элементы представляют результаты выполнения соответствующих им агрегатных функций.

Агрегатные функции можно указать в виде как позиционных, так и именованных параметров:

- если агрегатная функция указана в виде *позиционного* параметра, то в результирующем словаре будет создан элемент с ключом вида *<имя поля, по которому выполняется вычисление>__<имя класса агрегатной функции>*, хранящий результат выполнения этой функции. Для примера определим наименьшее значение цены, указанное в объявлениях:

```
>>> from django.db.models import Min
>>> Bb.objects.aggregate(Min('price'))
{'price__min': 40.0}
```

- если агрегатная функция указана в виде *именованного* параметра, то ключ элемента, создаваемого в словаре, будет совпадать с именем этого параметра. Выясним наибольшее значение цены в объявлениях:

```
>>> from django.db.models import Max
>>> Bb.objects.aggregate(max_price=Max('price'))
{'max_price': 50000000.0}
```

В вызове метода `aggregate()` допускается задавать произвольное число агрегатных функций:

```
>>> result = Bb.objects.aggregate(Min('price'), Max('price'))
>>> result['price__min'], result['price__max']
(40.0, 50000000.0)
```

А использовав для указания именованный параметр (с позиционным такой номер не пройдет), — выполнять вычисления над результатами агрегатных функций:

```
>>> result = Bb.objects.aggregate(diff=Max('price')-Min('price'))
>>> result['diff']
49999960.0
```

7.5.2. Вычисления по группам записей

Если нужно провести агрегатное вычисление по группам записей, сформированным согласно определенному критерию (например, узнать, сколько объявлений находится в каждой рубрике), то следует применить метод `annotate()`:

```
annotate(<агрегатная функция 1>, <агрегатная функция 2> . . .
                                 <агрегатная функция n>)
```

Вызывается он так же, как и `aggregate()` (см. *разд. 7.5.1*), но с двумя отличиями:

- в качестве результата возвращается новый набор записей;
- каждая запись из возвращенного набора содержит атрибут, имя которого генерируется по тем же правилам, что и ключ элемента в словаре, возвращенном методом `aggregate()`. Этот атрибут хранит результат выполнения агрегатной функции.

Пример подсчета числа объявлений, оставленных в каждой из рубрик (агрегатная функция указана в позиционном параметре):

```
>>> from django.db.models import Count
>>> for r in Rubric.objects.annotate(Count('bb')):
        print(r.name, ': ', r.bb__count, sep='')
...
Бытовая техника: 2
Мебель: 1
```

```
Недвижимость: 3
Растения: 0
Сантехника: 1
Сельхозинвентарь: 2
Транспорт: 2
```

То же самое, но теперь агрегатная функция указана в именованном параметре:

```
>>> for r in Rubric.objects.annotate(cnt=Count('bb')):
...     print(r.name, ': ', r.cnt, sep='')
```

Посчитаем для каждой из рубрик минимальную цену, указанную в объявлении:

```
>>> for r in Rubric.objects.annotate(min=Min('bb__price')):
        print(r.name, ': ', r.min, sep='')
...
Бытовая техника: 1000.0
Мебель: 200.0
Недвижимость: 100000.0
Растения: None
Сантехника: 50.0
Сельхозинвентарь: 40.0
Транспорт: 40000.0
```

У рубрики "Растения", не содержащей объявлений, значение минимальной цены равно `None`.

Использовав именованный параметр, мы фактически создаем в наборе записей новое поле (более подробно об этом приеме разговор пойдет позже). Следовательно, мы можем фильтровать записи по значению этого поля. Давайте же уберем из полученного ранее результата рубрики, в которых нет объявлений:

```
>>> for r in Rubric.objects.annotate(cnt=Count('bb'),
                                     min=Min('bb__price')).filter(cnt__gt=0):
        print(r.name, ': ', r.min, sep='')
...
Бытовая техника: 1000.0
Мебель: 200.0
Недвижимость: 100000.0
Сантехника: 50.0
Сельхозинвентарь: 40.0
Транспорт: 40000.0
```

7.5.3. Агрегатные функции

Все агрегатные функции, поддерживаемые Django, представляются классами из модуля `django.db.models`.

Конструкторы этих классов принимают ряд необязательных параметров. Указывать их можно лишь в том случае, если сама агрегатная функция задана с помощью именованного параметра, — иначе мы получим сообщение об ошибке.

- `Count` — вычисляет число записей. Формат конструктора:

```
Count(<имя поля>[, distinct=False][, filter=None])
```

Первым параметром указывается *имя поля*, имеющегося в записях, число которых должно быть подсчитано. Если нужно узнать число записей вторичной модели, связанных с записью первичной модели, то следует указать имя вторичной модели.

Если значение параметра `distinct` равно `True`, то будут подсчитываться только уникальные записи, если `False` — все записи (поведение по умолчанию). Параметр `filter` указывает условие для фильтрации записей в виде экземпляра класса `Q` (см. *разд. 7.3.9*); если он не задан, фильтрация не выполняется.

Пример подсчета объявлений, в которых указана цена более 100 000 руб., по рубрикам:

```
>>> for r in Rubric.objects.annotate(cnt=Count('bb',
                                     filter=Q(bb__price__gt=100000))):
        print(r.name, ': ', r.cnt, sep='')
...
Бытовая техника: 0
Мебель: 0
Недвижимость: 2
Растения: 0
Сантехника: 0
Сельхозинвентарь: 0
Транспорт: 0
```

- `Sum` — вычисляет сумму значений, хранящихся в поле. Формат конструктора:

```
Sum(<имя поля или выражение>[, output_field=None][, filter=None][,
                                                   distinct=False])
```

Первым параметром указывается *имя поля*, сумма значений которых будет вычисляться, или *выражение*, представленное экземпляром класса `F` (см. *разд. 7.3.8*). Параметр `output_field` задает тип результирующего значения в виде экземпляра класса, представляющего поле нужного типа (см. *разд. 4.2.2*). По умолчанию тип результата совпадает с типом поля. Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q` (см. *разд. 7.3.9*); если он не задан, то фильтрация не выполняется.

Параметр `distinct` поддерживается, начиная с Django 3.0. Если он равен `False` (по умолчанию), то будут суммироваться все значения заданного *поля или выражения*, если `True` — только уникальные значения.

Пример подсчета суммарной цены всех объектов недвижимости и возврата ее в виде целого числа:

```
>>> from django.db.models import Sum, IntegerField
>>> Bb.objects.aggregate(sum=Sum('price', output_field=IntegerField(),
                         filter=Q(rubric__name='Недвижимость')))
{'sum': 51100000}
```

- `Min` — вычисляет наименьшее значение из хранящихся в заданном поле. Формат конструктора:

  ```
  Min(<имя поля или выражение>[, output_field=None][, filter=None])
  ```

  Первым параметром указывается *имя поля* или *выражение*, представленное экземпляром класса `F`. Параметр `output_field` задает тип результирующего значения в виде экземпляра класса, представляющего поле нужного типа. По умолчанию тип результата совпадает с типом поля. Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q`; если он не указан, то фильтрация не выполняется.

- `Max` — вычисляет наибольшее значение из хранящихся в заданном поле. Формат конструктора:

  ```
  Max(<имя поля или выражение>[, output_field=None][, filter=None])
  ```

  Первым параметром указывается *имя поля* или *выражение* в виде экземпляра класса `F`. Параметр `output_field` задает тип результирующего значения в виде экземпляра класса, представляющего поле нужного типа. По умолчанию тип результата совпадает с типом поля. Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q`; если он не задан, то фильтрация не выполняется.

- `Avg` — вычисляет среднее арифметическое. Формат конструктора:

  ```
  Avg(<имя поля или выражение>[, output_field=None][, filter=None][,
                                                     distinct=False])
  ```

  Первым параметром указывается *имя поля*, по содержимому которого будет вычисляться среднее арифметическое, или *выражение* в виде экземпляра класса `F`. Параметр `output_field` задает тип результирующего значения в виде экземпляра класса, представляющего поле нужного типа. По умолчанию это объект типа `Decimal` из модуля `decimal` Python, если заданное *поле* принадлежит типу `DecimalField` (см. *разд. 4.2.2*), и величина вещественного типа — в противном случае. Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q`. По умолчанию фильтрация не выполняется.

  Параметр `distinct` поддерживается, начиная с Django 3.0. Если ему дано значение `False` (по умолчанию), то среднее арифметическое будет рассчитано на основе всех значений заданного *поля или выражения*, если `True` — только уникальных значений.

- `StdDev` — вычисляет стандартное отклонение:

  ```
  StdDev(<имя поля или выражение>[, sample=False][, output_field=None][,
                                                   filter=None])
  ```

  Первым параметром указывается *имя поля*, по содержимому которого будет вычисляться стандартное отклонение, или *выражение* в виде экземпляра класса `F`. Если значение параметра `sample` равно `True`, то вычисляется стандартное отклонение выборки, если `False` — собственно стандартное отклонение (поведение по

умолчанию). Параметр `output_field` задает тип результирующего значения в виде экземпляра класса, представляющего поле нужного типа. По умолчанию это объект типа `Decimal` из модуля `decimal` Python, если заданное *поле* принадлежит типу `DecimalField` (см. *разд. 4.2.2*), и величина вещественного типа — в противном случае. Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q`. По умолчанию фильтрация не выполняется.

- `Variance` — вычисляет дисперсию:

```
Variance(<имя поля или выражение>[, sample=False][,
                                  output_field=None][, filter=None])
```

Первым параметром указывается *имя поля*, по содержимому которого будет вычисляться дисперсия, или *выражение* в виде экземпляра класса `F`. Если значение параметра `sample` равно `True`, то вычисляется стандартная дисперсия образца, если `False` — собственно дисперсия (поведение по умолчанию). Параметр `output_field` задает тип результирующего значения в виде экземпляра класса, представляющего поле нужного типа. По умолчанию это объект типа `Decimal` из модуля `decimal` Python, если заданное *поле* принадлежит типу `DecimalField` (см. *разд. 4.2.2*), и величина вещественного типа — в противном случае. Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q`. По умолчанию фильтрация не выполняется.

7.6. Вычисляемые поля

Значения *вычисляемых полей* не берутся из базы данных, а вычисляются СУБД на основе других данных. В этом вычисляемые поля подобны функциональным (см. *разд. 4.6*), с тем исключением, что значения последних рассчитываются самим Django.

7.6.1. Простейшие вычисляемые поля

В самом простом виде вычисляемое поле создается с помощью метода `annotate()`, который нам уже знаком. В нем, с применением именованного параметра, указывается выражение, вычисляющее значение поля.

- Для указания значения какого-либо поля применяется экземпляр класса `F` (см. *разд. 7.3.8*).
- Константы целочисленного, вещественного и логического типов можно указывать как есть.
- Для записи константы строкового типа используется экземпляр класса `Value` из модуля `django.db.models`, конструктор которого вызывается в формате:

```
Value(<значение константы>[, output_field=None])
```

Само *значение константы* записывается в первом параметре конструктора этого класса.

В необязательном параметре `output_field` можно задать тип константы в виде экземпляра класса, представляющего поле нужного типа. Если параметр отсутствует, тип значения будет определен автоматически.

- Для вычислений применяются операторы `+`, `-`, `*`, `/` и `//`, записываемые как есть. Они выполняют соответствующие действия над переданными им экземплярами классов `F` и `Value` и возвращают результат этих действий также в виде экземпляра класса `F`.

  Оператор `-` позволяет изменить знак значения, представленного экземпляром класса `F` или `Value`.

В качестве примера вычислим для каждого объявления половину указанной в нем цены:

```
>>> from django.db.models import F
>>> for b in Bb.objects.annotate(half_price=F('price')/2):
        print(b.title, b.half_price)
...
Стиральная машина 1500.0
Пылесос 500.0
# Остальной вывод пропущен
```

Константа `2` здесь указана как есть, поскольку она принадлежит целочисленному типу, и Django благополучно обработает ее.

Выведем записанные в объявлениях названия товаров и, в скобках, названия рубрик:

```
>>> from django.db.models import Value
>>> from django.db.models.functions import Concat
>>> for b in Bb.objects.annotate(full_name=Concat(F('title'),
        Value(' ('), F('rubric__name'), Value(')'))): print(b.full_name)
...
Стиральная машина (Бытовая техника)
Пылесос (Бытовая техника)
Лопата (Сельхозинвентарь)
# Остальной вывод пропущен
```

В некоторых случаях может понадобиться указать для выражения, записанного в виде экземпляра класса `F`, тип возвращаемого им результата. Непосредственно в конструкторе класса `F` это сделать не получится. Положение спасет класс `ExpressionWrapper` из модуля `django.db.models`. Вот формат его конструктора:

```
ExpressionWrapper(<выражение>, <тип результата>)
```

Выражение представляется экземпляром класса `F`, а тип данных — экземпляром класса поля нужного типа. Пример:

```
>>> from django.db.models import ExpressionWrapper, IntegerField
>>> for b in Bb.objects.annotate(
            half_price=ExpressionWrapper(F('price') / 2, IntegerField())):
        print(b.title, b.half_price)
...
```

```
Стиральная машина 1500
Пылесос 500
# Остальной вывод пропущен
```

7.6.2. Функции СУБД

Функция СУБД обрабатывается не Django и не Python, а СУБД. Django просто предоставляет для этих функций удобный объектный интерфейс.

Функции СУБД используются в вызовах метода `annotate` и представляются следующими классами из модуля `django.db.models.functions`:

- `Coalesce` — возвращает первое переданное ему значение, отличное от `null` (даже если это пустая строка или 0). Конструктор класса вызывается в формате:

  `Coalesce(`*<значение 1>*`, `*<значение 2>*` . . . `*<значение n>*`)`

  Значения представляются строковыми именами полей, экземплярами классов `F` и `Value`. Все они должны иметь одинаковый тип (например, только строковый или только числовой), в противном случае мы получим ошибку.

  Пример:

  ```
  Coalesce('content', 'addendum', Value('--пусто--'))
  ```

  Если значение поля `content` отлично от `null`, то будет возвращено оно. В противном случае будет проверено значение поля `addendum` и, если оно не равно `null`, функция вернет его. Если же и значение поля `addendum` равно `null`, то будет возвращена константа `'--пусто--'`;

- `Greatest` — возвращает наибольшее значение из переданных ему:

  `Greatest(`*<значение 1>*`, `*<значение 2>*` . . . `*<значение n>*`)`

  Значения представляются строковыми именами полей, экземплярами классов `F` и `Value`. Все они должны иметь одинаковый тип (например, только строковый или только числовой), в противном случае мы получим ошибку.

  Пример:

  ```
  >>> from django.db.models.functions import Greatest
  >>> for b in Bb.objects.annotate(gp=Greatest('price', 1000)):
          print(b.title, ': ', b.gp)
  ...
  Стиральная машина :  3000.0
  Пылесос :  1000.0
  Лопата :  1000.0
  Мотыга :  1000.0
  Софа :  1000.0
  Диван :  1000.0
  Земельный участок :  100000.0
  Велосипед :  40000.0
  Мотоцикл :  50000.0
  ```

```
Дом :  50000000.0
Дача :  1000000.0
```

- `Least` — возвращает наименьшее значение из переданных ему:

  `Least(<значение 1>, <значение 2> . . . <значение n>)`

  Значения представляются строковыми именами полей, экземплярами классов `F` и `Value`. Все они должны иметь одинаковый тип;

- `Cast` — принудительно преобразует заданное *значение* к указанному *типу* и возвращает результат преобразования:

  `Cast(<значение>, <тип>)`

  Значение представляется строкой с именем поля или экземпляром класса `F`. *Тип* должен указываться в виде экземпляра класса, представляющего поле соответствующего типа;

- `Concat` — объединяет переданные ему *значения* в одну строку, которая и возвращается в качестве результата:

  `Concat(<значение 1>, <значение 2> . . . <значение n>)`

  Значения представляются строковыми именами полей, экземплярами классов `F` и `Value`. Все они должны иметь строковый или текстовый тип;

- `Lower` — преобразует символы строки к нижнему регистру и возвращает преобразованную строку в качестве результата:

  `Lower(<значение>)`

  Значение представляется строкой с именем поля или экземпляром класса `F`. Оно должно иметь строковый или текстовый тип;

- `Upper` — преобразует символы строки к верхнему регистру и возвращает преобразованную строку в качестве результата:

  `Upper(<значение>)`

  Значение представляется строкой с именем поля или экземпляром класса `F`. Оно должно иметь строковый или текстовый тип;

- `Length` — возвращает длину полученного *значения* в символах:

  `Length(<значение>)`

  Значение представляется строкой с именем поля или экземпляром класса `F`. Оно должно иметь строковый или текстовый тип. Если *значение* равно `null`, то возвращается `None`;

- `StrIndex` — возвращает номер вхождения указанной *подстроки* в строковое *значение*. Нумерация символов в строке начинается с 1. Если *подстрока* отсутствует в *значении*, возвращается 0. Формат конструктора:

  `StrIndex(<значение>, <подстрока>)`

  Значение и *подстрока* представляются строками с именами поля или экземплярами классов `F` или `Value`. Они должны иметь строковый или текстовый тип;

- `Substr` — извлекает из *значения подстроку* с указанными *позицией* первого символа и *длиной* и возвращает в качестве результата:

  ```
  Substr(<значение>, <позиция>[, <длина>])
  ```

  Значение представляется строкой с именем поля или экземпляром класса `F`. Оно должно иметь строковый или текстовый тип.

  При указании *позиции* и *длины* извлекаемой подстроки нужно иметь в виду, что нумерация символов в строке начинается с 1. Если *длина* не задана, извлекается вся оставшаяся часть *значения*;

- `Left` — возвращает подстроку, начинающуюся с начала заданного *значения* и имеющую заданную *длину*:

  ```
  Left(<значение>, <длина>)
  ```

- `Right` — возвращает подстроку, заканчивающуюся в конце заданного *значения* и имеющую заданную *длину*:

  ```
  Right(<значение>, <длина>)
  ```

- `Replace` — возвращает *значение*, в котором вместо всех вхождений *заменяемой подстроки* подставлена *заменяющая подстрока*:

  ```
  Replace(<значение>, <заменяемая подстрока>[, <заменяющая подстрока>])
  ```

  Если *заменяющая подстрока* не указана, используется пустая строка (т. е. функция фактически будет удалять все вхождения *заменяемой подстроки*). Поиск *заменяемой подстроки* выполняется с учетом регистра символов;

- `Repeat` — возвращает заданное *значение*, повторенное заданное *число* раз:

  ```
  Repeat(<значение>, <число повторов>)
  ```

- `LPad` — возвращает заданное *значение*, дополненное слева *символами-заполнителями* таким образом, чтобы достичь указанной *длины*:

  ```
  LPad(<значение>, <длина>[, <символ-заполнитель>])
  ```

  Если *символ-заполнитель* не указан, то используется пробел;

- `RPad` — возвращает заданное *значение*, дополненное справа *символами-заполнителями* таким образом, чтобы достичь указанной *длины*:

  ```
  RPad(<значение>, <длина>[, <символ-заполнитель>])
  ```

  Если *символ-заполнитель* не указан, то используется пробел;

- `Trim` — возвращает указанное *значение* с удаленными начальными и конечными пробелами:

  ```
  Trim(<значение>)
  ```

- `LTrim` — возвращает указанное *значение* с удаленными начальными пробелами:

  ```
  LTrim(<значение>)
  ```

- `RTrim` — возвращает указанное *значение* с удаленными конечными пробелами:

  ```
  RTrim(<значение>)
  ```

- `Chr` — возвращает символ с указанным целочисленным *кодом*:

  ```
  Chr(<код символа>)
  ```

- `Ord` — возвращает целочисленный код первого символа заданного *значения*:

  ```
  Ord(<значение>)
  ```

- `Now()` — возвращает текущие дату и время;

- `Extract` — извлекает часть значения даты и времени и возвращает его в виде числа:

  ```
  Extract(<значение даты и (или) времени>,
          <извлекаемая часть значения>[, tzinfo=None])
  ```

  Значение представляется экземпляром класса `F` и должно принадлежать типу даты, времени или временно́й отметки.

  Извлекаемая часть даты и времени представляется в виде строки `"year"` (год), `"quarter"` (номер квартала), `"month"` (номер месяца), `"day"` (число), `"week"` (порядковый номер недели), `"week_day"` (номер дня недели), `"hour"` (часы), `"minute"` (минуты) или `"second"` (секунды).

  Параметр `tzinfo` задает временну́ю зону. На практике он указывается крайне редко.

  Пример:

  ```
  Extract('published', 'year')
  ```

  Класс `Extract` можно заменить более простыми в применении классами: `ExtractYear` (извлекает год), `ExtractIsoYear` (год в формате ISO-8601, начиная с Django 2.2), `ExtractQuarter` (номер квартала), `ExtractMonth` (номер месяца), `ExtractDay` (число), `ExtractWeek` (порядковый номер недели), `ExtractWeekDay` (номер дня недели), `ExtractHour` (часы), `ExtractMinute` (минуты) и `ExtractSecond` (секунды). Формат вызова конструкторов этих классов:

  ```
  <класс>(<значение даты и (или) времени>[, tzinfo=None])
  ```

  Пример:

  ```
  ExtractYear('published')
  ```

- `Trunc` — сбрасывает в ноль *значение даты и (или) времени* до указанной *конечной части*, если считать справа:

  ```
  Trunc(<значение даты и (или) времени>, <конечная часть>[,
        output_field=None][, tzinfo=None][, is_dst=None])
  ```

  Значение представляется экземпляром класса `F` и должно принадлежать типу даты, времени или временно́й отметки.

Конечная часть представляется в виде строки `"year"` (год), `"quarter"` (номер квартала), `"month"` (номер месяца), `"week"` (неделя), `"day"` (число), `"hour"` (часы), `"minute"` (минуты) или `"second"` (секунды).

Параметр `output_field` указывает тип возвращаемого значения даты и (или) времени. Его значение должно представлять собой экземпляр класса `DateField`, `TimeField` или `DateTimeField` (они описаны в *разд. 4.2.2*). Если он не указан, то тип возвращаемого результата будет совпадать с типом изначального значения.

Параметр `tzinfo` задает временну́ю зону. На практике он указывается крайне редко.

Параметр `is_dst` поддерживается, начиная с Django 3.0. Если ему задать значение `None` (по умолчанию) или `False`, то коррекция летнего времени проводиться не будет, что в некоторых случаях может вызвать возникновение ошибок и возбуждение исключений `AmbiguousTimeError` из модуля `pytz.exceptions`. Если задано значение `True`, то коррекция летнего времени будет проводиться.

Примеры:

```
# Предположим, в поле published хранится временна́я отметка
# 03.12.2019 14:33:28.366947
Trunc('published', 'year')      #  01.01.2019 00:00:00
Trunc('published', 'quarter')   #  01.10.2019 00:00:00
Trunc('published', 'month')     #  01.12.2019 00:00:00
Trunc('published', 'week')      #  02.12.2019 00:00:00
Trunc('published', 'day')       #  03.12.2019 00:00:00
Trunc('published', 'hour')      #  03.12.2019 14:00:00
Trunc('published', 'minute')    #  03.12.2019 14:33:00
Trunc('published', 'second')    #  03.12.2019 14:33:28
```

Вместо класса `Trunc` можно использовать более простые в применении классы: `TruncYear` (сбрасывает до года), `TruncQuarter` (до квартала), `TruncMonth` (до месяца), `TruncWeek` (до полуночи понедельника текущей недели), `TruncDay` (до числа), `TruncHour` (до часов), `TruncMinute` (до минут), `TruncSecond` (до секунд), `TruncDate` (извлекает значение даты) и `TruncTime` (извлекает значение времени). Формат вызова конструкторов этих классов:

```
<класс>(<значение даты и (или) времени>[, output_field=None][, tzinfo=None])
```

Пример:

```
TruncYear('published')
```

Поддержка следующих функций появилась в Django 2.2:

- `Reverse` — возвращает заданное *значение*, в котором символы выстроены в обратном порядке:

```
Reverse(<значение>)
```

Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть строкового или текстового типа.

Пример:

```
>>> from django.db.models.functions import Reverse
>>> b = Bb.objects.filter(title='Велосипед').annotate(
                          rev_title=Reverse('title'))
>>> b[0].rev_title
'деписолеВ'
```

- `NullIf` — возвращает `None`, если заданные *значения* равны, и *значение 1* — в противном случае:

  `NullIf(<значение 1>, <значение 2>)`

  Значения представляются строковыми именами полей, экземплярами классов `F` или `Value`;

- `Sqrt` — возвращает квадратный корень *значения*:

  `Sqrt(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа;

- `Mod` — возвращает остаток от целочисленного деления *значения 1* на *значение 2*:

  `Mod(<значение 1>, <значение 2>)`

  Значения представляются строковыми именами полей или экземплярами класса `F`. Они должны быть целочисленного или вещественного типа;

- `Power` — возвращает результат возведения в степень:

  `Power(<основание>, <показатель>)`

  Основание и *показатель* представляются строковыми именами полей или экземплярами класса `F`. Они должны быть целочисленного или вещественного типа;

- `Round` — округляет заданное *значение* до ближайшего целого и возвращает результат:

  `Round(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа;

- `Floor` — округляет заданное *значение* до ближайшего меньшего целого и возвращает результат:

  `Floor(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа;

- `Ceil` — округляет заданное *значение* до ближайшего большего целого и возвращает результат:

  `Ceil(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа;

- `Pi()` — возвращает значение числа π. Вызывается без параметров;
- `Abs` — возвращает абсолютное значение, вычисленное от заданного *значения*:

  `Abs(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа;
- `Sin` — возвращает синус *значения*, заданного в радианах:

  `Sin(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа;
- `Cos` — возвращает косинус *значения*, заданного в радианах:

  `Cos(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа;
- `Tan` — возвращает тангенс *значения*, заданного в радианах:

  `Tan(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа;
- `Cot` — возвращает котангенс *значения*, заданного в радианах:

  `Cot(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа;
- `ASin` — возвращает арксинус *значения*:

  `ASin(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа и находиться в диапазоне от –1 до 1;
- `ACos` — возвращает арккосинус *значения*:

  `ACos(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа и находиться в диапазоне от –1 до 1;
- `ATan` — возвращает арктангенс *значения*:

  `ATan(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа;

- `ATan2` — возвращает арктангенс от частного от деления *значения 1* на *значение 2*:

  `ATan2(<значение 1>, <значение 2>)`

  Значения представляются строковыми именами полей или экземплярами класса `F`. Они должны быть целочисленного или вещественного типа;

- `Radians` — преобразует заданное значение из градусов в радианы и возвращает результат:

  `Padians(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа;

- `Degrees` — преобразует заданное значение из радианов в градусы и возвращает результат:

  `Degrees(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа;

- `Exp` — возвращает результат вычисления экспоненты от *значения*:

  `Exp(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа;

- `Log` — возвращает логарифм от *значения* по заданному *основанию*:

  `Log(<основание>, <значение>)`

  Основание и *значение* представляются строковыми именами полей или экземплярами класса `F`. Они должны быть целочисленного или вещественного типа;

- `Ln` — возвращает натуральный логарифм от *значения*:

  `Ln(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа.

Поддержка следующих функций появилась в Django 3.0:

- `Sign` — возвращает –1, если *значение* отрицательное, 0, если равно нулю, и 1, если положительное:

  `Sign(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть целочисленного или вещественного типа;

- `MD5` — возвращает хэш значения, вычисленный по алгоритму MD5:

  `MD5(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть строкового или текстового типа.

Пример:

```
>>> from django.db.models.functions import MD5
>>> b = Bb.objects.annotate(hash=MD5('title')).first()
>>> b.title
'Стиральная машина'
>>> b.hash
'4c5602d936847378b9604e3fbd80a98f'
```

- `SHA1` — возвращает хэш значения, вычисленный по алгоритму SHA1:

  `SHA1(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть строкового или текстового типа;

- `SHA244` — возвращает хэш значения, вычисленный по алгоритму SHA244:

  `SHA244(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть строкового или текстового типа.

  Внимание!

  Oracle не поддерживает функцию `SHA244`.

- `SHA256` — возвращает хэш значения, вычисленный по алгоритму SHA256:

  `SHA256(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть строкового или текстового типа;

- `SHA384` — возвращает хэш значения, вычисленный по алгоритму SHA384:

  `SHA384(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть строкового или текстового типа;

- `SHA512` — возвращает хэш значения, вычисленный по алгоритму SHA512:

  `SHA512(<значение>)`

  Значение представляется строковым именем поля или экземпляром класса `F`. Оно должно быть строкового или текстового типа.

  Внимание!

  Для использования функций `SHA1`, `SHA244`, `SHA256`, `SHA384` и `SHA512` в PostgreSQL следует установить расширение pgcrypto. Процесс его установки описан в документации по этой СУБД.

7.6.3. Условные выражения СУБД

Такие условные выражения обрабатываются непосредственно СУБД.

Для записи условного выражения применяется класс `Case` из модуля `django.db.models`. Формат его конструктора:

```
Cast(<условие 1>, <условие 2> . . . <условие n>[, default=None][,
                                                output_field=None])
```

Каждое *условие* записывается в виде экземпляра класса `When` из модуля `django.db.models`, конструктор которого имеет следующий формат:

```
When(<условие>, then=None)
```

Условие можно записать в формате, рассмотренном в *разд. 7.3.4*, или же в виде экземпляра класса `Q` (см. *разд. 7.3.9*). Параметр `then` указывает значение, которое будет возвращено при выполнении *условия*.

Вернемся к классу `Case`. Указанные в нем *условия* проверяются в порядке, в котором они записаны. Если какое-либо из *условий* выполняется, возвращается результат, заданный в параметре `then` этого *условия*, при этом остальные *условия* из присутствующих в конструкторе класса `Case` не проверяются. Если ни одно из *условий* не выполнилось, то возвращается значение, заданное параметром `default`, или `None`, если таковой отсутствует.

Параметр `output_field` задает тип возвращаемого результата в виде экземпляра класса поля подходящего типа. Хотя в документации по Django он помечен как необязательный, по опыту автора, лучше его указывать.

Выведем список рубрик и, против каждой из них, пометку, говорящую о том, сколько объявлений оставлено в этой рубрике:

```
>>> from django.db.models import Case, When, Value, Count, CharField
>>> for r in Rubric.objects.annotate(cnt=Count('bb'),
            cnt_s=Case(When(cnt__gte=5, then=Value('Много')),
                       When(cnt__gte=3, then=Value('Средне')),
                       When(cnt__gte=1, then=Value('Мало')),
                       default=Value('Вообще нет'),
                       output_field=CharField())):
        print('%s: %s' % (r.name, r.cnt_s))
...
Бытовая техника: Мало
Мебель: Мало
Недвижимость: Средне
Растения: Вообще нет
Сантехника: Мало
Сельхозинвентарь: Мало
Транспорт: Мало
```

7.6.4. Вложенные запросы

Вложенные запросы могут использоваться в условиях фильтрации или для расчета результатов у вычисляемых полей. Django позволяет создавать вложенные запросы двух видов.

Запросы первого вида — полнофункциональные — возвращают какой-либо результат. Они создаются с применением класса `Subquery` из модуля `django.db.models`. Формат конструктора этого класса таков:

```
Subquery(<вложенный набор записей>[, output_field=None])
```

Вложенный набор записей формируется с применением описанных ранее инструментов. Необязательный параметр `output_field` задает тип возвращаемого вложенным запросом результата, если этим результатом является единичное значение (как извлечь из набора записей значение единственного поля, мы узнаем очень скоро).

Если во *вложенном наборе записей* необходимо ссылаться на поле "внешнего" набора записей, то ссылка на это поле, записываемая в условиях фильтрации *вложенного набора записей*, оформляется как экземпляр класса `OuterRef` из того же модуля `django.db.models`:

```
OuterRef(<поле "внешнего" набора записей>)
```

В качестве примера извлечем список рубрик и для каждой выведем дату и время публикации самого "свежего" объявления:

```
>>> from django.db.models import Subquery, OuterRef, DateTimeField
>>> sq = Subquery(Bb.objects.filter(
         rubric=OuterRef('pk')).order_by('-published').values(
                                                        'published')[:1])
>>> for r in Rubric.objects.annotate(last_bb_date=sq):
        print(r.name, r.last_bb_date)
...
Бытовая техника 2019-12-03 09:59:08.835913+00:00
Мебель 2019-12-03 09:48:17.614623+00:00
Недвижимость 2019-12-03 09:32:51.414377+00:00
Растения None
Сантехника 2019-12-03 09:50:07.111273+00:00
Сельхозинвентарь 2019-12-03 09:55:23.869020+00:00
Транспорт 2019-12-03 09:25:57.441447+00:00
```

Во вложенном запросе необходимо сравнить значение поля `rubric` объявления со значением ключевого поля `pk` рубрики, которое присутствует во "внешнем" запросе. Ссылку на это поле мы оформили как экземпляр класса `OuterRef`.

Вложенные запросы второго вида лишь позволяют проверить, присутствуют ли в таком запросе записи. Они создаются с применением класса `Exists` из модуля `django.db.models`:

```
Exists(<вложенный набор записей>)
```

Выведем список только тех рубрик, в которых присутствуют объявления с заявленной ценой более 100 000 руб.:

```
>>> from django.db.models import Exists
>>> subquery = Exists(Bb.objects.filter(rubric=OuterRef('pk'),
                                        price__gt=100000))
```

```
>>> for r in Rubric.objects.annotate(is_expensive=subquery).filter(
                                    is_expensive=True): print(r.name)
...
Недвижимость
```

7.7. Объединение наборов записей

Для объединения нескольких наборов записей в один применяется метод `union()`:

```
union(<набор записей 1>, <набор записей 2> . . . <набор записей n>[,
                                                  all=False])
```

Все заданные *наборы записей* будут объединены с текущим, и получившийся набор будет возвращен в качестве результата.

Если значение необязательного параметра `all` равно `True`, то в результирующем наборе будут присутствовать все записи, в том числе и одинаковые. Если же для параметра указать значение `False`, результирующий набор будет содержать только уникальные записи (поведение по умолчанию).

> ***Внимание!***
>
> **У наборов записей, предназначенных к объединению, не следует задавать сортировку. Если же таковая все же была указана, нужно убрать ее, вызвав метод `order_by()` без параметров.**

Для примера сформируем набор из объявлений с заявленной ценой более 100 000 руб. и объявлений по продаже бытовой техники:

```
>>> bbs1 = Bb.objects.filter(price__gte=100000).order_by()
>>> bbs2 = Bb.objects.filter(rubric__name='Бытовая техника').order_by()
>>> for b in bbs1.union(bbs2): print(b.title, sep=' ')
...
Дача
Дом
Земельный участок
Пылесос
Стиральная машина
```

Django поддерживает два более специализированных метода для объединения наборов записей:

- `intersection(<набор записей 1>, <набор записей 2> . . . <набор записей n>)` — возвращает набор, содержащий только записи, которые имеются во всех объединяемых наборах;
- `difference(<набор записей 1>, <набор записей 2> . . . <набор записей n>)` — возвращает набор, содержащий только записи, которые имеются лишь в каком-либо одном из объединяемых наборов, но не в двух или более сразу.

7.8. Извлечение значений только из заданных полей

Каждый из ранее описанных методов возвращает в качестве результата набор записей — последовательность объектов, представляющих записи. Такие структуры данных удобны в работе над целыми записями, однако отнимают много системных ресурсов.

Если необходимо извлечь из модели только значения определенного поля (полей) хранящихся там записей, удобнее применить следующие методы:

- `values([<поле 1>, <поле 2> . . . <поле n>])` — извлекает из модели значения только указанных *полей*. Возвращает набор записей (экземпляр класса `QuerySet`), элементами которого являются словари. Ключи элементов таких словарей совпадают с именами заданных полей, а значения элементов — это и есть значения полей.

  Поле может быть задано:

  - позиционным параметром — в виде строки со своим именем;
  - именованным параметром — в виде экземпляра класса `F`. Имя параметра станет именем поля.

  Если в числе полей, указанных в вызове метода, присутствует поле внешнего ключа, то элемент результирующего словаря, соответствующий этому полю, будет хранить значение ключа связанной записи, а не саму запись.

  Примеры:

```
>>> Bb.objects.values('title', 'price', 'rubric')
<QuerySet [
    {'title': 'Стиральная машина', 'price': 3000.0, 'rubric': 3},
    {'title': 'Пылесос', 'price': 1000.0, 'rubric': 3},
    # Часть вывода пропущена
]>

>>> Bb.objects.values('title', 'price', rub=F('rubric'))
<QuerySet [
    {'title': 'Стиральная машина', 'price': 3000.0, 'rub': 3},
    {'title': 'Пылесос', 'price': 1000.0, 'rub': 3},
    # Часть вывода пропущена
]>
```

  Если метод `values()` вызван без параметров, то он вернет набор словарей со всеми полями модели. При этом вместо полей модели он будет содержать поля таблицы из базы данных, обрабатываемой этой моделью. Вот пример (обратим внимание на имена полей — это поля таблицы, а не модели):

```
>>> Bb.objects.values()
<QuerySet [
    {'id': 23, 'title': 'Стиральная машина',
```

```
    'content': 'Автоматическая', 'price': 3000.0,
    'published': datetime.datetime(2019, 12, 3, 9, 59, 8, 835913,
    tzinfo=<UTC>), 'rubric_id': 3},
    # Часть вывода пропущена
]>
```

- `values_list([`*`<поле 1>`*`, `*`<поле 2>`*` . . . `*`<поле n>`*`][,][flat=False][,][named=False])` — то же самое, что `values()`, но возвращенный им набор записей будет содержать кортежи:

```
>>> Bb.objects.values_list('title', 'price', 'rubric')
<QuerySet [
    ('Стиральная машина', 3000.0, 3),
    ('Пылесос', 1000.0, 3),
    # Часть вывода пропущена
]>
```

Параметр `flat` имеет смысл указывать только в случае, если возвращается значение одного поля. Если его значение — `False`, то значения этого поля будут оформлены как кортежи из одного элемента (поведение по умолчанию). Если же задать для него значение `True`, возвращенный набор записей будет содержать значения поля непосредственно. Пример:

```
>>> Bb.objects.values_list('title')
<QuerySet [('Стиральная машина',), ('Пылесос',), ('Лопата',),
('Мотыга',), ('Софа',), ('Диван',), ('Земельный участок',),
('Велосипед',), ('Мотоцикл',), ('Дом',), ('Дача',)]>
>>> Bb.objects.values_list('title', flat=True)
<QuerySet ['Стиральная машина', 'Пылесос', 'Лопата', 'Мотыга', 'Софа',
'Диван', 'Земельный участок', 'Велосипед', 'Мотоцикл', 'Дом', 'Дача']>
```

Если параметру `named` дать значение `True`, то набор записей будет содержать не обычные кортежи, а именованные;

- `dates(`*`<имя поля>`*`, `*`<часть даты>`*`[, order='ASC'])` — возвращает набор записей (экземпляр класса `QuerySet`) с уникальными значениями даты, которые присутствуют в поле с заданным *именем* и урезаны до заданной *части*. В качестве *части даты* можно указать `"year"` (год), `"month"` (месяц) или `"day"` (число, т. е. дата не будет урезаться). Если параметру `order` дать значение `"ASC"`, то значения в наборе записей будут отсортированы по возрастанию (поведение по умолчанию), если `"DESC"` — по убыванию. Примеры:

```
>>> Bb.objects.dates('published', 'day')
<QuerySet [datetime.date(2019, 12, 3), datetime.date(2019, 11, 21)]>
>>> Bb.objects.dates('published', 'month')
<QuerySet [datetime.date(2019, 12, 1), datetime.date(2019, 11, 1)]>
```

- `datetimes(`*`<имя поля>`*`, `*`<часть даты и времени>`*`[, order='ASC'][, tzinfo=None])` — то же самое, что `dates()`, но манипулирует значениями временны́х отметок. В качестве *части даты и времени* можно указать `"year"` (год), `"month"` (месяц), `"day"` (число), `"hour"` (часы), `"minute"` (минуты) или `"second"` (секунды,

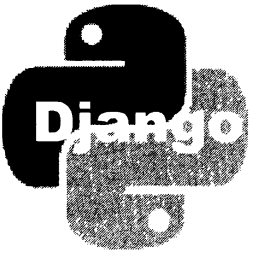
т. е. значение не будет урезаться). Параметр `tzinfo` указывает временну́ю зону. Пример:

```
>>> Bb.objects.datetimes('published', 'day')
<QuerySet [datetime.datetime(2019, 12, 3, 0, 0, tzinfo=<UTC>),
datetime.datetime(2019, 11, 21, 0, 0, tzinfo=<UTC>)]>
```

- `in_bulk(`*`<последовательность значений>`*`[, field_name='pk'])` — ищет в модели записи, у которых поле с именем, заданным параметром `field_name`, хранит значения из указанной *последовательности*. Возвращает словарь, ключами элементов которого станут значения из *последовательности*, а значениями элементов — объекты модели, представляющие найденные записи. Заданное поле должно хранить уникальные значения (параметру `unique` конструктора поля модели нужно задать значение `True`). Примеры:

```
>>> Rubric.objects.in_bulk([1, 2, 3])
{1: <Rubric: Недвижимость>, 2: <Rubric: Транспорт>,
3: <Rubric: Бытовая техника>}

>>> Rubric.objects.in_bulk(['Транспорт', 'Мебель'], field_name='name')
{'Мебель': <Rubric: Мебель>, 'Транспорт': <Rubric: Транспорт>}
```



ГЛАВА 8

Маршрутизация

Маршрутизация — это определение контроллера, который следует выполнить при получении в составе клиентского запроса интернет-адреса заданного формата. Подсистема фреймворка, выполняющая маршрутизацию, носит название *маршрутизатора*.

8.1. Как работает маршрутизатор

Маршрутизатор Django работает, основываясь на написанном разработчиком списке маршрутов. Каждый элемент такого списка — *маршрут* — устанавливает связь между путем определенного формата (шаблонным путем) и контроллером. Вместо контроллера в маршруте может быть указан вложенный список маршрутов.

Алгоритм работы маршрутизатора следующий:

1. Из полученного запроса извлекается запрашиваемый клиентом интернет-адрес.
2. Из интернет-адреса удаляются обозначение протокола, доменное имя (или IP-адрес) хоста, номер TCP-порта, набор GET-параметров и имя якоря, если они там присутствуют. В результате остается один только путь.
3. Этот путь последовательно сравнивается с шаблонными путями, записанными во всех маршрутах, имеющихся в списке.
4. Как только шаблонный путь из очередного проверяемого маршрута совпадёт с *началом* полученного пути:
   - если в маршруте указан контроллер, он выполняется;
   - если в маршруте указан вложенный список маршрутов, то из полученного пути удаляется префикс, совпадающий с шаблонным путем, и начинается просмотр маршрутов из вложенного списка.
5. Если ни один из шаблонных путей не совпал с полученным в запросе, то клиенту отправляется стандартная страница сообщения об ошибке 404 (запрошенная страница не найдена).

Обратим особое внимание на то, что маршрутизатор считает полученный путь совпавшим с шаблонным, если последний присутствует в начале первого. Это может привести к неприятной коллизии. Например, если мы имеем список маршрутов с шаблонными путями **create/** и **create/comment/** (расположенными именно в таком порядке), то при получении запроса с путем **/create/comment/** маршрутизатор посчитает, что произошло совпадение с шаблонным путем **create/**, т. к. он присутствует в начале запрашиваемого пути. Поэтому в рассматриваемом случае нужно расположить маршруты в порядке **create/comment/** и **create/**, тогда описанная здесь коллизия не возникнет.

В составе запрашиваемого пути может быть передано какое-либо значение. Чтобы получить его, достаточно поместить в нужное место шаблонного пути в соответствующем маршруте обозначение URL-параметра. Маршрутизатор извлечет значение и передаст его либо в контроллер, либо вложенному списку маршрутов (и тогда полученное значение станет доступно всем контроллерам, объявленным во вложенном списке).

По возможности маршруты должны содержать уникальные шаблонные пути. Создавать маршруты с одинаковыми шаблонными путями допускается, но не имеет смысла, т. к. в любом случае будет срабатывать самый первый маршрут из совпадающих, а последующие окажутся бесполезны.

8.1.1. Списки маршрутов уровня проекта и уровня приложения

Поскольку маршрутизатор Django поддерживает объявление вложенных списков маршрутов, все маршруты проекта описываются в виде своего рода иерархии:

- в списке маршрутов, входящем в состав пакета конфигурации, записываются маршруты, ведущие на отдельные приложения проекта.

  Каждый из этих маршрутов указывает на вложенный список, принадлежащий соответствующему приложению.

  По умолчанию список маршрутов уровня проекта записывается в модуле urls.py пакета конфигурации. Можно сохранить его и в другом модуле, указав его путь, отсчитанный от папки проекта, в параметре `ROOT_URLCONF` модуля settings.py пакета конфигурации (см. *главу 3*);

- а в списках маршрутов, принадлежащих отдельным приложениям, записываются маршруты, указывающие непосредственно на контроллеры, которые входят в состав этих приложений.

  Под хранение маршрута уровня приложения, как правило, в пакете приложения создается отдельный модуль, обычно с именем urls.py. Его придется создать вручную, поскольку команда `startapp` утилиты manage.py этого не делает.

8.2. Объявление маршрутов

Любые списки маршрутов — неважно, уровня проекта или приложения — объявляются согласно схожим правилам.

Список маршрутов оформляется как обычный список Python и присваивается переменной с именем `urlpatterns` — именно там маршрутизатор Django будет искать его. Примеры объявления списков маршрутов можно увидеть в листингах из *глав 1* и *2*.

Каждый элемент списка маршрутов должен представлять собой результат, возвращаемый функцией `path()` из модуля `django.urls`. Формат вызова этой функции таков:

```
path(<шаблонный путь>, <контроллер>|<вложенный список маршрутов>[,
    <дополнительные параметры>][, name=<имя маршрута>])
```

Шаблонный путь записывается в виде строки. В конце он должен содержать прямой слеш, а в начале таковой, напротив, не ставится. О *дополнительных параметрах* и *имени маршрута* мы поговорим позже.

В качестве *шаблонного пути* можно указать пустую строку, создав *корневой маршрут*. Он будет связан с "корнем" приложения (если задан в списке маршрутов уровня этого приложения) или всего сайта (будучи заданным в списке уровня проекта).

Для объявления в *шаблонном пути* URL-параметров (параметризованный маршрут) применяется следующий формат:

```
< [<обозначение формата>:]<имя URL-параметра> >
```

Поддерживаются следующие *обозначения форматов* для значений URL-параметров:

- `str` — любая непустая строка, не включающая слеши (формат по умолчанию);
- `int` — положительное целое число, включая 0;
- `slug` — строковый слаг, содержащий латинские буквы, цифры, знаки дефиса и подчеркивания;
- `uuid` — уникальный универсальный идентификатор. Он должен быть правильно отформатирован: должны присутствовать дефисы, а буквы следует указать в нижнем регистре;
- `path` — фрагмент пути — любая непустая строка, включающая слеши.

Имя URL-параметра задает имя для параметра контроллера, через который последний сможет получить значение, переданное в составе интернет-адреса. Это имя должно удовлетворять правилам именования переменных Python.

Вернемся к функции `path()`. *Контроллер* указывается в виде ссылки на функцию (если это контроллер-функция, речь о которых пойдет в *главе 9*) или результата, возвращенного методом `as_view()` контроллера-класса (контроллерам-классам посвящена *глава 10*).

Пример написания маршрутов, указывающих на контроллеры, можно увидеть в листинге 8.1. Первый маршрут указывает на контроллер-класс `BbCreateView`, а второй и третий — на контроллеры-функции `by_rubric()` и `index()`.

Листинг 8.1. Маршруты, указывающие на контроллеры

```
from django.urls import path
from .views import index, by_rubric, BbCreateView

urlpatterns = [
    path('add/', BbCreateView.as_view()),
    path('<int:rubric_id>/', by_rubric),
    # Корневой маршрут, указывающий на "корень" приложения bboard
    path('', index),
]
```

Вложенный список маршрутов указывается в виде результата, возвращенного функцией `include()` из того же модуля `django.urls`. Вот ее формат вызова:

```
include(<путь к модулю>|<вложенный список маршрутов>[,
        namespace=<имя пространства имен>])
```

В качестве первого параметра обычно указывается строка с *путем к модулю*, содержащему вложенный список маршрутов. О пространствах имен и их *именах* мы поговорим позже.

Пример указания маршрутов, ведущих на вложенные списки маршрутов, показан в листинге 8.2. В первом маршруте вложенный список задан в виде пути к модулю, а во втором — в виде готового списка маршрутов (он хранится в свойстве `urls` экземпляра класса `AdminSite`, содержащегося в переменной `site` модуля `django.contrib.admin`).

Листинг 8.2. Маршруты, указывающие на вложенные списки маршрутов

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    # Корневой маршрут, указывающий на "корень" самого веб-сайта
    path('', include('bboard.urls')),
    path('admin/', admin.site.urls),
]
```

В первом параметре функции `include()` можно указать непосредственно *вложенный список маршрутов*. Это может пригодиться при программировании простого сайта — отпадет нужда создавать отдельные модули под списки маршрутов уровня приложений.

Пример списка маршрутов уровня проекта, непосредственно включающего вложенный список уровня приложения, приведен в листинге 8.3. В таком случае нужда в отдельном модуле urls.py, хранящем список маршрутов уровня приложения, отпадает.

Листинг 8.3. Список маршрутов уровня проекта, включающий список маршрутов уровня приложения

```
from django.contrib import admin
from django.urls import path, include
from bboard.views import index, by_rubric, BbCreateView

urlpatterns = [
    path('bboard/', include([
        path('add/', BbCreateView.as_view(), name='add'),
        path('<int:rubric_id>/', by_rubric, name='by_rubric'),
        path('', index, name='index'),
    ])),
    path('admin/', admin.site.urls),
]
```

8.3. Передача данных в контроллеры

Значения URL-параметров контроллер-функция получает через параметры, имена которых совпадают с именами URL-параметров.

Так, в приведенном далее примере контроллер-функция `by_rubric()` получает значение URL-параметра `rubric_id` через параметр `rubric_id`:

```
urlpatterns = [
    path('<int:rubric_id>/', by_rubric),
]

def by_rubric(request, rubric_id):
    . . .
```

Есть еще один способ передать какие-либо данные в контроллер. Для этого нужно объявить словарь Python, создать в нем столько элементов, сколько нужно передать значений, присвоить передаваемые значения этим элементам и передать полученный словарь функции `path()` в третьем параметре. Эти значения контроллер сможет получить также через одноименные параметры.

Вот пример передачи контроллеру-функции значения `mode`:

```
vals = {'mode': 'index'}
urlpatterns = [
    path('<int:rubric_id>/', by_rubric, vals),
]

def by_rubric(request, rubric_id, mode):
    . . .
```

Внимание!

Если в маршруте присутствует URL-параметр с тем же именем, что и у дополнительного значения, передаваемого через словарь, контроллеру будет передано значение из словаря. Извлечь значение URL-параметра в таком случае не получится.

8.4. Именованные маршруты

Любому маршруту можно дать имя, указав его в необязательном параметре `name` функции `path()` (см. *разд. 8.2*) и создав тем самым именованный маршрут:

```
urlpatterns = [
    path('<int:rubric_id>/', by_rubric, name='by_rubric'),
]
```

После этого можно задействовать обратное разрешение интернет-адресов, т. е. автоматическое формирование готового адреса по заданным имени маршрута и набору параметров (если это параметризованный маршрут). Вот так оно выполняется в коде контроллера:

```
from django.urls import reverse
. . .
url = reverse('by_rubric', kwargs={'rubric_id': 2})
```

А так — в коде шаблона:

```
<a href="{% url 'by_rubric' rubric_id=2 %}">. . .</a>
```

Более подробно обратное разрешение будет описано в *главах 9* и *11*.

8.5. Имена приложений

Сложные сайты могут состоять из нескольких приложений, списки маршрутов которых могут содержать совпадающие шаблонные пути (например, в приложениях `bboard` и `otherapp` могут находиться маршруты с одинаковым шаблонным путем **index/**).

Как в таком случае дать понять Django, интернет-адрес из какого приложения нужно сформировать посредством обратного разрешения? Задать каждому приложению уникальное имя.

Имя приложения указывается в модуле со списком маршрутов уровня этого приложения. Строку с его названием нужно присвоить переменной `app_name`. Пример:

```
app_name = 'bboard'
urlpatterns = [
    . . .
]
```

Чтобы сослаться на маршрут, объявленный в нужном приложении, следует предварить имя маршрута именем этого приложения, разделив их двоеточием. Примеры:

```
# Формируем интернет-адрес страницы категории с ключом 2
# приложения bboard
url = reverse('bboard:by_rubric', kwargs={'rubric_id': 2})
# Результат: /bboard/2/

# Формируем в гиперссылке адрес главной страницы приложения bboard
<a href="{% url 'bboard:index' %}">. . .</a>
# Результат: /bboard/

# Формируем адрес главной страницы приложения admin (это административный
# веб-сайт Django)
<a href="{% url 'admin:index' %}">. . .</a>
# Результат: /admin/
```

8.6. Псевдонимы приложений

Псевдоним приложения — это своего рода его альтернативное имя. Оно может пригодиться в случае, если в проекте используются несколько экземпляров одного и того же приложения, которые манипулируют разными данными, связаны с разными путями и которые требуется как-то различать в программном коде.

Предположим, что у нас в проекте есть два экземпляра приложения `bboard`, и мы записали такой список маршрутов уровня проекта:

```
urlpatterns = [
    path('', include('bboard.urls')),
    path('bboard/', include('bboard.urls')),
]
```

Если теперь записать в коде шаблона тег:

```
<a href="{% url 'bboard:by_rubric' bb.rubric.pk %}">. . .</a>
```

будет сформирован интернет-адрес другой страницы *того же* приложения. Но как быть, если нужно указать на страницу, принадлежащую другому приложению? Дать обоим приложениям разные псевдонимы.

Псевдоним приложения указывается в параметре `namespace` функции `include()`:

```
urlpatterns = [
    path('', include('bboard.urls', namespace='default-bboard')),
    path('bboard/', include('bboard.urls', namespace='other-bboard')),
]
```

Заданный псевдоним используется вместо имени приложения при обратном разрешении маршрута:

```
# Создаем гиперссылку на главную страницу приложения с псевдонимом
# default-bboard
<a href="{% url 'defailt-bboard:index' %}">. . .</a>
```

```
# Создаем гиперссылку на страницу рубрики приложения с псевдонимом other-bboard
<a href="{% url 'other-bboard:by_rubric' bb.rubric.pk %}">. . .</a>
```

На заметку

В документации по Django для обозначения и имен, и псевдонимов приложений используется термин "пространство имен" (namespace), что, на взгляд автора, лишь порождает путаницу.

8.7. Указание шаблонных путей в виде регулярных выражений

Наконец, поддерживается указание шаблонных путей в виде регулярных выражений. Это может пригодиться, если шаблонный путь очень сложен, или при переносе кода, написанного под Django версий 1.*.

Для записи шаблонного пути в виде регулярного выражения применяется функция `re_path()` из модуля `django.urls`:

```
re_path(<регулярное выражение>,
        <контроллер>|<вложенный список маршрутов>[,
        <дополнительные параметры>][, name=<имя маршрута>])
```

Регулярное выражение должно быть представлено в виде строки и записано в формате, "понимаемом" модулем `re` языка Python. Остальные параметры точно такие же, как и у функции `path()`.

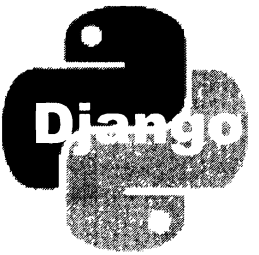
Пример указания шаблонных путей в виде регулярных выражений приведен в листинге 8.4.

Листинг 8.4. Указание шаблонных путей в виде регулярных выражений

```
from django.urls import re_path
from .views import index, by_rubric, BbCreateView

urlpatterns = [
    re_path(r'^add/$', BbCreateView.as_view(), name='add'),
    re_path(r'^(?P<rubric_id>[0-9]*)/$', by_rubric, name='by_rubric'),
    re_path(r'^$', index, name='index'),
]
```

Запись шаблонных интернет-адресов в виде регулярных выражений имеет одно преимущество: мы сами указываем, как будет выполняться сравнение. Записав в начале регулярного выражения обозначение начала строки, мы дадим понять Django, что шаблонный путь должен присутствовать в начале полученного пути (обычный режим сравнения). А дополнительно поставив в конец регулярного выражения обозначение конца строки, мы укажем, что полученный путь должен полностью совпадать с шаблонным. Иногда это может пригодиться.



ГЛАВА 9

Контроллеры-функции

Контроллер запускается при получении клиентского запроса по определенному интернет-адресу и в ответ генерирует соответствующую веб-страницу. Контроллеры реализуют основную часть логики сайта.

9.1. Введение в контроллеры-функции

Контроллеры-функции реализуются в виде обычных функций Python. Такая функция обязана принимать следующие параметры (указаны в порядке очередности их объявления):

- экземпляр класса `HttpRequest` (объявлен в модуле `django.http`), хранящий сведения о полученном клиентском запросе. Традиционно этот параметр носит имя `request`;
- набор именованных параметров, имена которых совпадают с именами URL-параметров, объявленных в связанном с контроллером маршруте.

Контроллер-функция должен возвращать в качестве результата экземпляр класса `HttpResponse`, также объявленного в модуле `django.http`, или какого-либо из его подклассов. Этот экземпляр класса представляет ответ, отсылаемый клиенту (веб-страница, обычный текстовый документ, файл, данные в формате JSON, перенаправление или сообщение об ошибке).

При создании нового приложения утилита manage.py записывает в пакет приложения модуль views.py, в котором, как предполагают разработчики фреймворка, и будет находиться код контроллеров. Однако ничто не мешает сохранить код контроллеров в других модулях с произвольными именами.

9.2. Как пишутся контроллеры-функции

Принципы написания контроллеров-функций достаточно просты. Однако здесь имеется один подводный камень, обусловленный самой природой Django.

9.2.1. Контроллеры, выполняющие одну задачу

Если контроллер-функция должен выполнять всего одну задачу — скажем, вывод веб-страницы, то все очень просто. Для примера рассмотрим код из листинга 2.2 — он объявляет контроллер-функцию `by_rubric()`, которая выводит страницу с объявлениями, относящимися к выбранной посетителем рубрике.

Если нужно выводить на экран страницу для добавления объявления и потом сохранять это объявление в базе данных, понадобятся два контроллера такого рода.

Напишем контроллер-функцию, который создаст форму и выведет на экран страницу добавления объявления (листинг 9.1).

Листинг 9.1. Контроллер-функция add()

```
def add(request):
    bbf = BbForm()
    context = {'form': bbf}
    return render(request, 'bboard/create.html', context)
```

Далее напишем контроллер-функцию `add_save()`, который сохранит введенное объявление в базе (листинг 9.2).

Листинг 9.2. Контроллер-функция add_save()

```
def add_save(request):
    bbf = BbForm(request.POST)
    if bbf.is_valid():
        bbf.save()
        return HttpResponseRedirect(reverse('by_rubric',
               kwargs={'rubric_id': bbf.cleaned_data['rubric'].pk}))
    else:
        context = {'form': bbf}
        return render(request, 'bboard/create.html', context)
```

Объявим маршруты, ведущие на эти контроллеры:

```
from .views import add, add_save

urlpatterns = [
    path('add/save/', add_save, name='add_save'),
    path('add/', add, name='add'),
    . . .
]
```

После этого останется открыть шаблон bboard\create.html, написанный в *главе 2*, и добавить в имеющийся в его коде тег `<form>` атрибут `action` с интернет-адресом, указывающим на контроллер `add_save()`:

```
<form action="{% url 'add_save' %}" method="post">
```

9.2.2. Контроллеры, выполняющие несколько задач

На практике для обработки данных, вводимых посетителями в формы, обычно применяют не два контроллера, а один. Он и выведет страницу с формой, и сохранит занесенные в нее данные. Код такого контроллера-функции `add_and_save()` приведен в листинге 9.3.

Листинг 9.3. Контроллер-функция add_and_save()

```
def add_and_save(request):
    if request.method == 'POST':
        bbf = BbForm(request.POST)
        if bbf.is_valid():
            bbf.save()
            return HttpResponseRedirect(reverse('bboard:by_rubric',
                   kwargs={'rubric_id': bbf.cleaned_data['rubric'].pk}))
        else:
            context = {'form': bbf}
            return render(request, 'bboard/create.html', context)
    else:
        bbf = BbForm()
        context = {'form': bbf}
        return render(request, 'bboard/create.html', context)
```

Сначала проверяется, с применением какого HTTP-метода был отправлен запрос. Если это метод GET, значит, посетитель хочет зайти на страницу добавления нового объявления, и эту страницу, с пустой формой, нужно вывести на экран.

Если же запрос был выполнен методом POST, значит, осуществляется отсылка введенных в форму данных, и их надо сохранить в базе. Занесенные в форму данные проверяются на корректность и, если проверка прошла успешно, сохраняются в базе, в противном случае страница с формой выводится на экран повторно. После успешного сохранения осуществляется перенаправление на страницу со списком объявлений из категории, заданной при вводе нового объявления.

Поскольку мы обходимся только одним контроллером, нам понадобится лишь один маршрут:

```
from .views import add_and_save

urlpatterns = [
    path('add/', add_and_save, name='add'),
    . . .
]
```

В коде шаблона bboard\create.html — в теге `<form>`, создающем форму, — уже не нужно указывать интернет-адрес для отправки занесенных в форму данных:

```
<form method="post">
```

В этом случае данные будут отправлены по тому же интернет-адресу, с которого была загружена текущая страница.

9.3. Формирование ответа

Основная задача контроллера — сформировать ответ, который будет отправлен посетителю. Обычно такой ответ содержит веб-страницу.

9.3.1. Низкоуровневые средства для формирования ответа

Формированием ответа на самом низком уровне занимается класс `HttpResponse` из модуля `django.http`. Его конструктор вызывается в следующем формате:

```
HttpResponse([<содержимое>][,][content_type=None][,][status=200][,]
                                                    [reason=None])
```

Содержимое должно быть указано в виде строки или последовательности строк.

Необязательный параметр `content_type` задает MIME-тип ответа и его кодировку. Если он отсутствует, ответ получит MIME-тип `text/html` и кодировку из параметра `DEFAULT_CHARSET`, указанного в настройках проекта (см. *разд. 3.3.1*).

Параметр `status` указывает целочисленный код статуса ответа (по умолчанию — `200`, т. е. файл успешно отправлен), а параметр `reason` — строковый статус (по умолчанию — `"OK"`).

Класс ответа поддерживает атрибуты:

- `content` — содержимое ответа в виде объекта типа `bytes`;
- `charset` — обозначение кодировки;
- `status_code` — целочисленный код статуса;
- `reason_phrase` — строковое обозначение статуса;
- `streaming` — если `True`, это потоковый ответ, если `False` — обычный. Будучи вызванным у экземпляра класса `HttpResponse`, метод всегда возвращает `False`;
- `closed` — `True`, если ответ закрыт, и `False`, если еще нет.

Также поддерживаются следующие методы:

- `write(<строка>)` — добавляет *строку* в ответ;
- `writelines(<последовательность строк>)` — добавляет к ответу строки из указанной *последовательности*. Разделители строк при этом не вставляются;
- `flush()` — принудительно переносит содержимое буфера записи в ответ;
- `has_header(<заголовок>)` — возвращает `True`, если указанный заголовок существует в ответе, и `False` — в противном случае;
- `setdefault(<заголовок>, <значение>)` — создает в ответе указанный *заголовок* с указанным *значением*, если таковой там отсутствует.

Класс HttpResponse поддерживает функциональность словарей, которой мы можем пользоваться, чтобы указывать и получать значения заголовков:

```
response['pragma'] = 'no-cache'
age = response['Age']
del response['Age']
```

В листинге 9.4 приведен код простого контроллера, использующего низкоуровневые средства для создания ответа и выводящего строку: **Здесь будет главная страница сайта**. Кроме того, он задает в ответе заголовок keywords со значением "Python, Django".

Листинг 9.4. Использование низкоуровневых средств формирования ответа

```
from django.http import HttpResponse

def index(request):
    resp = HttpResponse("Здесь будет",
                        content_type='text/plain; charset=utf-8')
    resp.write(' главная')
    resp.writelines((' страница', ' сайта'))
    resp['keywords'] = 'Python, Django'
    return resp
```

9.3.2. Формирование ответа на основе шаблона

Применять низкоуровневые средства для создания ответа на практике приходится крайне редко. Гораздо чаще мы имеем дело с высокоуровневыми средствами — шаблонами.

Для загрузки нужного шаблона Django предоставляет две функции, объявленные в модуле django.template.loader:

- get_template(*<путь к шаблону>*) — загружает шаблон, расположенный по указанному *пути*, и возвращает представляющий его экземпляр класса Template из модуля django.template.
- select_template(*<последовательность путей шаблонов>*) — перебирает указанную *последовательность путей шаблонов*, пытается загрузить шаблон, расположенный по очередному пути, и возвращает первый шаблон, который удалось загрузить, в виде экземпляра класса Template.

Если шаблон загрузить не получилось, то будет возбуждено исключение TemplateDoesNotExist. Если в коде шаблона встретилась ошибка, возбуждается исключение TemplateSyntaxError. Оба класса исключений объявлены в модуле django.template.

Пути шаблонов указываются относительно папки, в которой хранятся шаблоны (как указать эту папку, будет рассказано в *главе 11*).

Для получения обычной веб-страницы нужно выполнить *рендеринг* шаблона, вызвав один из следующих методов:

- `render([context=<контекст шаблона>][,][request=<запрос>])` — метод класса `Template`. Выполняет рендеринг текущего шаблона на основе заданного *контекста* (задается в виде словаря со значениями, которые должны быть доступны в шаблоне). Если указан *запрос* (экземпляр класса `Request`), то он также будет добавлен в контекст шаблона. Возвращает строку с HTML-кодом сформированной страницы. Пример использования этого метода можно увидеть в листинге 9.5.

Листинг 9.5. Применение метода `render()` для рендеринга шаблона

```
from django.http import HttpResponse
from django.template.loader import get_template

def index(request):
    bbs = Bb.objects.all()
    rubrics = Rubric.objects.all()
    context = {'bbs': bbs, 'rubrics': rubrics}
    template = get_template('bboard/index.html')
    return HttpResponse(template.render(context=context,
                                        request=request))
```

Параметры у метода `render()` можно указать не только как именованные, но и как позиционные:

```
return HttpResponse(template.render(context, request))
```

- `render_to_string(<путь к шаблону>[, context=<контекст шаблона>][, request=<запрос>])` — функция из модуля `django.template.loader`. Загружает шаблон с указанным *путем* и выполняет его рендеринг с применением заданных *контекста шаблона* и *запроса*. Возвращает строку с HTML-кодом сформированной страницы. Пример использования функции `render_to_string()` приведен в листинге 9.6.

Листинг 9.6. Применение функции `render_to_string()` для рендеринга шаблона

```
from django.http import HttpResponse
from django.template.loader import render_to_string

def index(request):
    bbs = Bb.objects.all()
    rubrics = Rubric.objects.all()
    context = {'bbs': bbs, 'rubrics': rubrics}
    return HttpResponse(render_to_string('bboard/index.html', context,
                                         request))
```

9.3.3. Класс *TemplateResponse*: отложенный рендеринг шаблонов

Помимо класса `HttpResponse`, ответ можно сформировать с помощью аналогичного класса `TemplateResponse` из модуля `django.template.response`.

Основное преимущество класса `TemplateResponse` проявляется при использовании посредников, добавляющих в контекст шаблона дополнительные данные (о посредниках разговор пойдет в *главе 21*). Этот класс поддерживает отложенный рендеринг шаблона, выполняющийся только после "прохождения" всей цепочки зарегистрированных в проекте посредников, непосредственно перед отправкой ответа клиенту. Благодаря этому посредники, собственно, и могут добавить в контекст шаблона дополнительные данные.

Кроме того, в виде экземпляров класса `TemplateResponse` генерируют ответы все высокоуровневые контроллеры-классы, о которых будет рассказано в *главе 10*.

Конструктор класса `TemplateResponse` вызывается в следующем формате:

```
TemplateResponse(<запрос>, <путь к шаблону>[, context=None][,
                         content_type=None][, status=200])
```

Запрос должен быть представлен в виде экземпляра класса `HttpRequest`. Он доступен внутри любого контроллера-функции через его первый параметр. Параметр `context` указывает контекст шаблона.

Необязательный параметр `content_type` задает MIME-тип ответа и его кодировку. Если он отсутствует, то ответ получит MIME-тип `text/html` и кодировку из параметра `DEFAULT_CHARSET`, указанного в настройках проекта (см. *разд. 3.3.1*). Параметр `status` указывает целочисленный код статуса ответа (по умолчанию — `200`, т. е. файл успешно отправлен).

Из всех атрибутов, поддерживаемых описываемым классом, наиболее интересны только `template_name` и `context_data`. Первый хранит путь к шаблону, а второй — контекст шаблона.

Листинг 9.7. Использование класса `TemplateResponse`

```
from django.template.response import TemplateResponse

def index(request):
    bbs = Bb.objects.all()
    rubrics = Rubric.objects.all()
    context = {'bbs': bbs, 'rubrics': rubrics}
    return TemplateResponse(request, 'bboard/index.html',
                                     context=context)
```

9.4. Получение сведений о запросе

Полученный от посетителя запрос представляется экземпляром класса `HttpRequest`. Он хранит разнообразные сведения о запросе, которые могут оказаться очень полезными.

Прежде всего, это ряд атрибутов, хранящих различные величины:

- `GET` — словарь со всеми GET-параметрами, полученными в составе запроса. Ключи элементов этого словаря совпадают с именами GET-параметров, а значения элементов — суть значения этих параметров;
- `POST` — словарь со всеми POST-параметрами, полученными в составе запроса. Ключи элементов этого словаря совпадают с именами POST-параметров, а значения элементов — суть значения этих параметров;
- `FILES` — словарь со всеми выгруженными файлами. Ключи элементов этого словаря совпадают с именами POST-параметров, посредством которых передается содержимое файлов, а значения элементов — сами файлы, представленные экземплярами класса `UploadedFile`;
- `method` — обозначение HTTP-метода в виде строки, набранной прописными буквами (`"GET"`, `"POST"` и т. д.);
- `scheme` — обозначение протокола (`"http"` или `"https"`);
- `path` и `path_info` — путь;
- `encoding` — обозначение кодировки, в которой был получен запрос. Если `None`, запрос закодирован в кодировке по умолчанию (она задается в параметре `DEFAULT_CHARSET` настроек проекта);
- `content_type` — обозначение MIME-типа полученного запроса, извлеченное из HTTP-заголовка `Content-Type`;
- `content_params` — словарь, содержащий дополнительные параметры MIME-типа полученного запроса, которые извлекаются из HTTP-заголовка `Content-Type`. Ключи элементов соответствуют самим параметрам, а значения элементов — значениям параметров;
- `META` — словарь, содержащий дополнительные параметры в виде следующих элементов:
  - `CONTENT_LENGTH` — длина тела запроса в символах, заданная в виде строки;
  - `CONTENT_TYPE` — MIME-тип тела запроса (может быть `"application/x-www-form-urlencoded"`, `"multipart/form-data"` или `"text/plain"`);
  - `HTTP_ACCEPT` — строка с перечнем поддерживаемых веб-обозревателем MIME-типов данных, разделенных запятыми;
  - `HTTP_ACCEPT_ENCODINGS` — строка с перечнем поддерживаемых веб-обозревателей кодировок, разделенных запятыми;

- HTTP_ACCEPT_LANGUAGES — строка с перечнем поддерживаемых веб-обозревателем языков, разделенных запятыми;
- HTTP_HOST — доменное имя (или IP-адрес) и номер TCP-порта, если он отличается от используемого по умолчанию, веб-сервера, с которого была загружена страница;
- HTTP_REFERER — интернет-адрес страницы, с которой был выполнен переход на текущую страницу (может отсутствовать, если это первая страница, открытая в веб-обозревателе);
- HTTP_USER_AGENT — строка с обозначением веб-обозревателя, запрашивающего страницу;
- QUERY_STRING — строка с необработанными GET-параметрами;
- REMOTE_ADDR — IP-адрес клиентского компьютера, запрашивающего страницу;
- REMOTE_HOST — доменное имя клиентского компьютера, запрашивающего страницу. Если доменное имя не удается определить, элемент хранит пустую строку;
- REMOTE_USER — имя пользователя, выполнившего вход на веб-сервер. Если вход на веб-сервер не был выполнен или если используется другой способ аутентификации, этот элемент будет отсутствовать;
- REQUEST_METHOD — обозначение HTTP-метода ("GET", "POST" и т. д.);
- SERVER_NAME — доменное имя сервера;
- SERVER_PORT — номер TCP-порта, через который работает веб-сервер, в виде строки.

В словаре, хранящемся в атрибуте META, могут присутствовать и другие элементы. Они формируются на основе любых нестандартных заголовков, которые имеются в клиентском запросе. Имя такого элемента создается на основе имени соответствующего заголовка, преобразованного к верхнему регистру, с подчеркиваниями вместо дефисов и с префиксом HTTP_, добавленным в начало. Так, на основе заголовка Upgrade-Insecure-Requests в словаре META будет создан элемент с ключом HTTP_UPGRADE_INSECURE_REQUESTS;

- body — "сырое" содержимое запроса в виде объекта типа bytes;
- resolver_match — экземпляр класса ResolverMatch (рассмотрен в *разд. 9.10*), описывающий сработавший маршрут (о маршрутах рассказывалось в *главе 8*);
- headers (начиная с Django 2.2) — объект, хранящий все заголовки запроса. Обладает функциональностью словаря, ключи элементов которого совпадают с именами заголовков, а значениями элементов являются значения, переданные в этих заголовках. Имена заголовков можно указывать в любом регистре. Пример:

```
print(request.headers['Accept-Encoding'])
print(request.headers['accept-encoding'])
# В обоих случаях будет выведено: gzip, deflate, br
```

Начиная с Django 3.0, в именах заголовков вместо дефисов можно записывать подчеркивания:

```
print(request.headers['accept_encoding'])
```

Методы, поддерживаемые классом `HttpRequest`:

- `get_host()` — возвращает строку с комбинацией IP-адреса (или доменного имени, если его удастся определить) и номера TCP-порта, через который работает веб-сервер;
- `get_port()` — возвращает строку с номером TCP-порта, через который работает веб-сервер;
- `get_full_path()` — возвращает полный путь к текущей странице;
- `build_absolute_uri(<путь>)` — строит полный интернет-адрес на основе доменного имени (IP-адреса) сервера и указанного *пути*:

```
print(request.build_absolute_uri('/test/url/'))
# Будет выведено: http://localhost:8000/test/uri/
```

- `is_secure()` — возвращает `True`, если обращение выполнялось по протоколу HTTPS, и `False` — если по протоколу HTTP;
- `is_ajax()` — возвращает `True`, если это AJAX-запрос, и `False` — если обычный.

  AJAX-запросы выявляются фреймворком по наличию в запросе заголовка `X_Requested_With` со значением `"XMLHttpRequest"`.

9.5. Перенаправление

Очень часто приходится выполнять *перенаправление* клиента по другому интернет-адресу. Так, после добавления объявления следует выполнить перенаправление на страницу списка объявлений, относящихся к рубрике, к которой принадлежит добавленное объявление.

Перенаправление такого рода называется *временным*. Оно просто вызывает переход на страницу с заданным интернет-адресом.

Для выполнения временного перенаправления нужно создать экземпляр класса `HttpResponseRedirect`, являющегося подклассом класса `HttpResponse` и объявленного в модуле `django.http`. Вот формат конструктора этого класса:

```
HttpResponseRedirect(<целевой интернет-адрес>[, status=302][,
                                                reason=None])
```

Целевой интернет-адрес указывается в виде строки.

Созданный таким образом экземпляр класса `HttpResponseRedirect` следует вернуть из контроллера-функции в качестве результата.

Пример выполнения временного перенаправления:

```
return HttpResponseRedirect(reverse('bboard:index'))
```

Постоянное перенаправление применяется в тех случаях, когда сайт "переезжает" на новый интернет-адрес, и при заходе по старому адресу посетителя нужно отправить по новому местоположению. Помимо перехода по новому адресу, веб-обозреватель заменяет новым старый интернет-адрес везде, где он присутствует: в списке истории, в избранном и др.

Постоянное перенаправление реализуется другим классом из модуля `django.http` — `HttpResponsePermanentRedirect`, также производным от `HttpResponse`:

```
HttpResponsePermanentRedirect(<целевой интернет-адрес>[, status=301][,
                                                        reason=None])
```

Пример:

```
return HttpResponsePermanentRedirect('http://www.new_address.ru/')
```

9.6. Обратное разрешение интернет-адресов

Механизм обратного разрешения формирует интернет-адреса на основе объявленных в списках именованных маршрутов (см. *главу 8*).

Для формирования адресов в коде контроллеров применяется функция `reverse()` из модуля `django.urls`:

```
reverse(<имя маршрута>[, args=None][, kwargs=None][, urlconf=None])
```

Имя маршрута указывается в виде строки. Если проект содержит несколько приложений с заданными пространствами имен, то первым параметром функции указывается строка вида `<пространство имен>:<имя маршрута>`.

Если указан параметризованный *маршрут*, то следует задать значения URL-параметров — одним из двух способов:

- в параметре `args` — в виде последовательности. Первый элемент такой последовательности задаст значение первого по счету URL-параметра в маршруте, второй элемент задаст значение второго URL-параметра и т. д.;
- в параметре `kwargs` — в виде словаря. Ключи его элементов соответствуют именам URL-параметров, а значения элементов зададут значения параметров.

Допускается указывать только один из этих параметров: `args` или `kwargs`. Задание обоих параметров вызовет ошибку.

Параметр `urlconf` задает путь к модулю со списком маршрутов, который будет использоваться для обратного разрешения. Если он не указан, задействуется модуль со списком маршрутов уровня проекта, заданный в его настройках (параметр `ROOT_URLCONF`. Более подробно о нем см. в *разд. 3.3.1*).

Если в параметре `urlconf` указан модуль с маршрутами уровня приложения, то записывать пространство имен в первом параметре функции `reverse()` не нужно.

Функция возвращает строку с интернет-адресом, полученным в результате обратного разрешения.

Примеры:

```
url1 = reverse('bboard:index', urlconf='bboard.urls')
url2 = reverse('bboard:by_rubric', args=(current_rubric.pk,))
url3 = reverse('bboard:by_rubric', kwargs={'rubric_id': current_rubric.pk})
```

Функция `reverse()` имеет серьезный недостаток — она работает лишь после того, как список маршрутов был загружен и обработан. Из-за этого ее можно использовать только в контроллерах.

Если же нужно указать интернет-адрес где-либо еще, например в атрибуте контроллера-класса (о них речь пойдет в *главе 10*), то следует применить функцию `reverse_lazy()` из того же модуля `django.urls`. Ее формат вызова точно такой же, как и у функции `reverse()`. Пример:

```
class BbCreateView(CreateView):
    . . .
    success_url = reverse_lazy('index')
```

Принудительно указать имя или псевдоним приложения, которое будет использоваться при обратном разрешении интернет-адресов в шаблонах, можно, занеся его в атрибут `current_app` объекта запроса:

```
def index(request):
    . . .
    request.current_app = 'other_bboard'
    . . .
```

Для обратного разрешения интернет-адресов в шаблонах применяется тег шаблонизатора `url`. Мы рассмотрим его в *главе 11*.

9.7. Выдача сообщений об ошибках и обработка особых ситуаций

Для выдачи сообщений об ошибках и уведомления клиентов об особых ситуациях (например, если страница не изменилась с момента предыдущего запроса, и ее можно загрузить из локального кэша) Django предоставляет ряд классов. Все они являются производными от класса `HttpResponse` и объявлены в модуле `django.http`.

- `HttpResponseNotFound([<содержимое>][,][content_type=None][,][status=404][,][reason=None])` — запрашиваемая страница не существует (код статуса 404):

```
def detail(request, bb_id):
    try:
        bb = Bb.objects.get(pk=bb_id)
    except Bb.DoesNotExist:
        return HttpResponseNotFound('Такое объявление не существует')
    return HttpResponse( . . . )
```

Также можно возбудить исключение `Http404` из модуля `django.http`:

```
def detail(request, bb_id):
    try:
        bb = Bb.objects.get(pk=bb_id)
    except Bb.DoesNotExist:
        raise Http404('Такое объявление не существует')
    return HttpResponse( . . . )
```

- `HttpResponseBadRequest([<содержимое>][,][content_type=None][,][status=400][,][reason=None])` — клиентский запрос некорректно сформирован (код статуса 400);
- `HttpResponseForbidden([<содержимое>][,][content_type=None][,][status=403][,][reason=None])` — доступ к запрошенной странице запрещен (код статуса 403).

  Также можно возбудить исключение `PermissionDenied` из модуля `django.core.exceptions`;
- `HttpResponseNotAllowed(<последовательность обозначений разрешенных методов>[,][content_type=None][,][status=405][,][reason=None])` — клиентский запрос был выполнен с применением недопустимого HTTP-метода (код статуса 405). Первым параметром конструктора указывается последовательность методов, которые допустимы для запрошенной страницы. Пример:

  ```
  return HttpResponseNotAllowed(['GET'])
  ```
- `HttpResponseGone([<содержимое>][,][content_type=None][,][status=410][,][reason=None])` — запрошенная страница удалена насовсем (код статуса 410);
- `HttpResponseServerError([<содержимое>][,][content_type=None][,][status=500][,][reason=None])` — ошибка в программном коде сайта (код статуса 500);
- `HttpResponseNotModified([<содержимое>][,][content_type=None][,][status=304][,][reason=None])` — запрашиваемая страница не изменилась с момента последнего запроса и может быть извлечена веб-обозревателем из локального кэша (код статуса 304).

9.8. Специальные ответы

Иногда нужно отправить посетителю данные формата, отличного от веб-страницы или простого текста. Для таких случаев Django предлагает три класса специальных ответов, объявленные в модуле `django.http`.

9.8.1. Потоковый ответ

Обычный ответ `HttpResponse` полностью формируется в оперативной памяти. Если объем ответа невелик, это вполне допустимо. Но для отправки страниц большого объема этот класс не годится, поскольку отнимет много памяти. В таких случаях применяется *потоковый ответ*, который формируется и отсылается по частям небольших размеров.

Потоковый ответ представляется классом `StreamingHttpResponse`. Формат его конструктора:

```
StreamingHttpResponse(<содержимое>[, content_type=None][, status=200][,
                                                          reason=None])
```

Содержимое задается в виде последовательности строк или итератора, на каждом проходе возвращающего строку. Остальные параметры такие же, как и у класса `HttpResponse`.

Класс поддерживает следующие атрибуты:

- `streaming_content` — итератор, на каждом проходе возвращающий фрагмент содержимого ответа в виде объекта типа `bytes`;
- `status_code` — целочисленный код статуса;
- `reason_phrase` — строковый статус;
- `streaming` — если `True`, это потоковый ответ, если `False` — обычный. Будучи вызванным у экземпляра класса `StreamingHttpResponse`, метод всегда возвращает `True`.

Пример кода, выполняющего отправку потокового ответа, приведен в листинге 9.8.

Листинг 9.8. Отправка потокового ответа

```
from django.http import StreamingHttpResponse

def index(request):
    resp_content = ('Здесь будет',' главная', ' страница', ' сайта')
    resp = StreamingHttpResponse(resp_content,
           content_type='text/plain; charset=utf-8')
    return resp
```

9.8.2. Отправка файлов

Для отправки клиентам файлов применяется класс `FileResponse` — производный от класса `StreamingHttpResponse`:

```
FileResponse(<файловый объект>[, as_attachment=False][, filename=''][,
             content_type=None][, status=200][, reason=None])
```

Пример:

```
from django.http import FileResponse
filename = r'c:/images/image.png'
return FileResponse(open(filename, 'rb'))
```

Отправленный таким образом файл будет открыт непосредственно в веб-обозревателе. Чтобы дать веб-обозревателю указание сохранить файл на локальном диске, достаточно задать в вызове конструктора класса `FileResponse` параметры:

- as_attachment со значением True;
- filename, в котором указать имя сохраняемого файла, — если заданный первым параметром *файловый объект* не содержит имени файла (например, если он был сформирован программно в оперативной памяти).

Пример:

```
filename = r'c:/archives/archive.zip'
return FileResponse(open(filename, 'rb'), as_attachment=True)
```

9.8.3. Отправка данных в формате JSON

Для отправки данных в формате JSON применяется класс JsonResponse — производный от класса HttpResponse. Формат его конструктора:

```
JsonResponse(<данные>[, safe=True][, encoder=DjangoJSONEncoder])
```

Кодируемые в JSON *данные* должны быть представлены в виде словаря Python. Если требуется закодировать и отправить что-либо отличное от словаря, нужно назначить параметру safe значение False. Параметр encoder задает кодировщик, применяемый для преобразования данных в формат JSON; если он не указан, используется стандартный кодировщик.

Пример:

```
data = {'title': 'Мотоцикл', 'content': 'Старый', 'price': 10000.0}
return JsonResponse(data)
```

9.9. Сокращения Django

Сокращение — это функция, выполняющая сразу несколько действий. Применение сокращений позволяет несколько уменьшить код и упростить программирование.

Все сокращения, доступные в Django, объявлены в модуле django.shortcuts:

- render(*<запрос>*, *<путь к шаблону>*[, context=None][, content_type=None][, status=200]) — выполняет рендеринг шаблона и отправку получившейся в результате страницы клиенту. *Запрос* должен быть представлен в виде экземпляра класса Request, *путь к шаблону* — в виде строки.

  Необязательный параметр context указывает контекст шаблона, content_type — MIME-тип и кодировку отправляемого ответа (по умолчанию text/html с кодировкой из параметра DEFAULT_CHARSET настроек проекта), а status — числовой код статуса (по умолчанию — 200).

  В качестве результата возвращается готовый ответ в виде экземпляра класса HttpResponse.

  Пример:

  ```
  return render(request, 'bboard/index.html', context)
  ```

- `redirect(<цель>[, permanent=False][, <значения URL-параметров>])` — выполняет перенаправление по заданной *цели*, в качестве которой могут быть указаны:
  - объект модели — тогда интернет-адрес для перенаправления будет получен вызовом метода `get_absolute_url()` этого объекта (см. *разд. 4.5*);
  - имя маршрута (возможно, с указанием пространства имен) и набор *значений URL-параметров* — тогда адрес для перенаправления будет сформирован с применением обратного разрешения.

    Значения URL-параметров могут быть указаны в вызове функции в виде как позиционных, так и именованных параметров;
  - непосредственно заданный интернет-адрес.

  Необязательный параметр `permanent` указывает тип перенаправления: временное (если `False` или если он опущен) или постоянное (если `True`).

  В качестве результата возвращается полностью сформированный экземпляр класса `HttpResponseRedirect`.

  Пример:

```
return redirect('bboard:by_rubric',
                rubric_id=bbf.cleaned_data['rubric'].pk)
```

- `get_object_or_404(<источник>, <условия поиска>)` — ищет запись согласно заданным *условиям поиска* и возвращает ее в качестве результата. Если запись найти не удается, возбуждает исключение `Http404`. В качестве *источника* можно указать класс модели, диспетчер записей (экземпляр класса `Manager`) или набор записей (экземпляр класса `QuerySet`).

  Если заданным условиям поиска удовлетворяют несколько записей, то возбуждается исключение `MultipleObjectsReturned`.

  Пример:

```
def detail(request, bb_id):
    bb = get_object_or_404(Bb, pk=bb_id)
    return HttpResponse( . . . )
```

- `get_list_or_404(<источник>, <условия фильтрации>)` — применяет к записям заданные *условия фильтрации* и возвращает в качестве результата полученный набор записей (экземпляр класса `QuerySet`). Если ни одной записи, удовлетворяющей *условиям*, не существует, то возбуждает исключение `Http404`. В качестве *источника* можно указать класс модели, диспетчер записей или набор записей.

  Пример:

```
def by_rubric(request, rubric_id):
    bbs = get_list_or_404(Bb, rubric=rubric_id)
    . . .
```

9.10. Программное разрешение интернет-адресов

Иногда может понадобиться программно "прогнать" какой-либо интернет-адрес через маршрутизатор, выяснить, совпадает ли он с каким-либо маршрутом, и получить сведения об этом маршруте, т. е. выполнить *программное разрешение* адреса.

Для этого предназначена функция `resolve()` из модуля `django.urls`:

```
resolve(<интернет-адрес>,[, urlconf=None])
```

Интернет-адрес указывается в виде строки.

Параметр `urlconf` задает путь к модулю со списком маршрутов, который будет использоваться для программного разрешения. Если он не указан, задействуется модуль со списком маршрутов уровня проекта, заданный в его настройках (параметр `ROOT_URLCONF`. Более подробно о нем см. в *разд. 3.3.1*).

Если заданный *адрес* совпадает с одним из перечисленных в списке маршрутов, то функция возвращает экземпляр класса `ResolverMatch`, хранящий сведения об *адресе* и совпавшем с ним маршруте. Если программное разрешение не увенчалось успехом, то возбуждается исключение `Resolver404`, производное от `Http404`, из того же модуля `django.urls`.

Класс `ResolverMatch` поддерживает следующие атрибуты:

- `func` — ссылка на контроллер (функцию или класс);
- `kwargs` — словарь со значениями URL-параметров, извлеченных из указанного *адреса*. Ключи элементов совпадают с именами URL-параметров. Если маршрут непараметризованный — пустой словарь;
- `url_name` — имя маршрута или `None`, если маршрут неименованный;
- `route` (начиная с Django 2.2) — строка с шаблонным путем;
- `view_name` — то же самое, что и `route`, но только с добавлением имени или псевдонима приложения;
- `app_name` — строка с именем приложения или `None`, если таковое не задано;
- `namespace` — строка с псевдонимом приложения. Если псевдоним не указан, то атрибут хранит имя приложения.

Пример:

```
>>> from django.urls import resolve
>>> r = resolve('/2/')
>>> r.kwargs
{'rubric_id': 2}
>>> r.url_name
'by_rubric'
>>> r.view_name
'default-bboard:by_rubric'
```

```
>>> r.route
'<int:rubric_id>/'
>>> r.app_name
'bboard'
>>> r.namespace
'default-bboard'
```

9.11. Дополнительные настройки контроллеров

Django предоставляет ряд декораторов, позволяющий задать дополнительные настройки контроллеров-функций.

Декораторы, задающие набор применимых к контроллеру HTTP-методов и объявленные в модуле `django.views.decorators.http`:

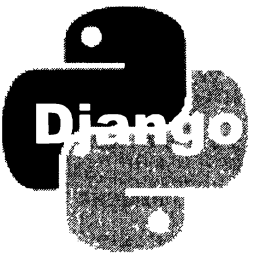
- `require_http_methods(`*<последовательность обозначений методов>*`)` — разрешает для контроллера только те HTTP-методы, обозначения которых указаны в заданной *последовательности*:

  ```
  from Django.views.decorators.http import require_http_methods
  @require_http_methods(['GET', 'POST'])
  def add(request):
      . . .
  ```

- `require_get()` — разрешает для контроллера только метод GET;
- `require_post()` — разрешает для контроллера только метод POST;
- `require_safe()` — разрешает для контроллера только методы GET и HEAD (они считаются безопасными, т. к. не изменяют внутренние данные сайта).

Если к контроллеру, помеченному одним из этих декораторов, отправить запрос с применением недопустимого HTTP-метода, то декоратор вернет экземпляр класса `HttpResponseNotAllowed` (см. *разд. 9.7*), тем самым отправляя клиенту сообщение о недопустимом методе.

Декоратор `gzip_page()` из модуля `django.views.decorators.gzip` сжимает ответ, сгенерированный помеченным контроллером, с применением алгоритма GZIP (конечно, если веб-обозреватель поддерживает такое сжатие).



ГЛАВА 10

Контроллеры-классы

Контроллер-класс, в отличие от контроллера-функции, может самостоятельно выполнять некоторые утилитарные действия (выборку из модели записи по полученному ключу, вывод страницы и др.).

Основную часть функциональности контроллеры-классы получают из *примесей* (классов, предназначенных лишь для расширения функциональности других классов), от которых наследуют. Мы рассмотрим как примеси, так и полноценные классы.

10.1. Введение в контроллеры-классы

Контроллер-класс записывается в маршруте не в виде ссылки, как контроллер-функция, а в виде результата, возвращенного методом `as_view()`, который поддерживается всеми контроллерами-классами. Вот пример указания в маршруте контроллера-класса `CreateView`:

```
path('add/', CreateView.as_view()),
```

В вызове метода `as_view()` можно задать параметры контроллера-класса. Пример указания модели и пути к шаблону (параметры `model` и `template_name` соответственно):

```
path('add/', CreateView.as_view(model=Bb, template_name='bboard/create.html')),
```

Задать параметры контроллера-класса можно и по-другому: создав производный от него класс и указав параметры в его атрибутах:

```
class BbCreateView(CreateView):
    template_name = 'bboard/create.html'
    model = Bb
. . .
path('add/', BbCreateView.as_view()),
```

Второй подход позволяет более радикально изменить поведение контроллера-класса, переопределив его методы, поэтому применяется чаще.

10.2. Базовые контроллеры-классы

Самые простые и низкоуровневые контроллеры-классы, называемые *базовыми*, объявлены в модуле `django.views.generic.base`.

10.2.1. Контроллер *View*: диспетчеризация по HTTP-методу

Контроллер-класс `View` определяет HTTP-метод, посредством которого был выполнен запрос, и исполняет код, соответствующий этому методу.

Класс поддерживает атрибут `http_method_names`, хранящий список имен допустимых HTTP-методов. По умолчанию он хранит список `['get', 'post', 'put', 'patch', 'delete', 'head', 'options', 'trace']`, включающий все методы, поддерживаемые протоколом HTTP.

Класс также содержит четыре метода (помимо уже знакомого нам `as_view()`), которые переопределяются в подклассах:

- `setup(self, request, *args, **kwargs)` (начиная с Django 2.2) — выполняется самым первым и инициализирует объект контроллера.

  Здесь и далее в параметре `request` передается объект запроса (в виде экземпляра класса `HttpRequest`), в параметре `kwargs` — словарь со значениями именованных URL-параметров.

  Параметр `args` остался в "наследство" от первых версий Django и служит для передачи списка со значениями неименованных URL-параметров. В Django 2.0 и более поздних версиях неименованные URL-параметры не поддерживаются, поэтому данный параметр не используется.

  В изначальной реализации создает в контроллере следующие атрибуты:

  - `request` — запрос, представленный экземпляром класса `Request`;
  - `kwargs` — словарь со значениями URL-параметров.

  Переопределив этот метод, можно сохранить в объекте контроллера какие-либо дополнительные данные;

- `dispatch(self, request, *args, **kwargs)` — обрабатывает полученный в параметре `request` запрос и возвращает ответ, представленный экземпляром класса `HttpResponse` или его подкласса. Выполняется после метода `setup()`.

  В изначальной реализации извлекает обозначение HTTP-метода, вызывает одноименный ему метод класса: `get()` — если запрос был выполнен HTTP-методом GET, `post()` — если запрос выполнялся методом POST, и т. п. — передавая ему все полученные параметры. Вызываемые методы должны быть объявлены в формате:

  <имя метода класса>`(self, request, *args, **kwargs)`

Если метод класса, одноименный с HTTP-методом, отсутствует, ничего не делает. Единственное исключение — HTTP-метод HEAD: при отсутствии метода `head()` вызывается метод `get()`;

- `http_method_not_allowed(self, request, *args, **kwargs)` — вызывается, если запрос был выполнен с применением неподдерживаемого HTTP-метода.

  В изначальной реализации возвращает ответ типа `HttpResponseNotAllowed` со списком допустимых методов;

- `options(self, request, *args, **kwargs)` — обрабатывает запрос, выполненный HTTP-методом OPTIONS.

  В изначальной реализации возвращает ответ с заголовком `Allow`, в котором записаны все поддерживаемые HTTP-методы.

Класс `View` используется крайне редко — обычно применяются производные от него контроллеры-классы, выполняющие более сложные действия.

10.2.2. Примесь *ContextMixin*: создание контекста шаблона

Класс-примесь `ContextMixin` добавляет контроллеру-классу средства для формирования контекста шаблона:

- `extra_context` — атрибут, задающий содержимое контекста шаблона. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст;

- `get_context_data(self, **kwargs)` — метод, должен создавать и возвращать контекст шаблона. С параметром `kwargs` передается словарь, элементы которого должны быть добавлены в контекст шаблона.

  В изначальной реализации создает пустой контекст шаблона, добавляет в него элемент `view`, хранящий ссылку на текущий экземпляр контроллера-класса, элементы из словарей `kwargs` и `extra_context`.

> ***Внимание!***
>
> Словарь, заданный в качестве значения атрибута `extra_content`, будет создан при первом обращении к контроллеру-классу и в дальнейшем останется неизменным. Если значением какого-либо элемента этого словаря является набор записей, он также останется неизменным, даже если впоследствии какие-то записи будут добавлены, исправлены или удалены.
>
> Поэтому добавлять наборы записей в контекст шаблона рекомендуется только в переопределенном методе `get_context_data()`. В этом случае набор записей будет создаваться каждый раз при выполнении этого метода и отразит самые последние изменения в модели.

10.2.3. Примесь *TemplateResponseMixin*: рендеринг шаблона

Класс-примесь `TemplateResponseMixin` добавляет наследующему классу средства для рендеринга шаблона.

> ***На заметку***
>
> Эта примесь и все наследующие от нее классы формируют ответ в виде экземпляра класса `TemplateResponse`, описанного в *разд. 9.3.3*.

Поддерживаются следующие атрибуты и методы:

- `template_name` — атрибут, задающий путь к шаблону в виде строки;
- `get_template_names(self)` — метод, должен возвращать список путей к шаблонам, заданных в виде строк.

  В изначальной реализации возвращает список из одного элемента — пути к шаблону, извлеченного из атрибута `template_name`;
- `response_class` — атрибут, задает ссылку на класс, представляющий ответ (по умолчанию: `TemplateResponse`);
- `content_type` — атрибут, задающий MIME-тип ответа и его кодировку. По умолчанию — `None` (используется MIME-тип и кодировка по умолчанию);
- `render_to_response(self, context, **response_kwargs)` — возвращает экземпляр класса, представляющего ответ. В параметре `context` передается контекст шаблона в виде словаря, а в параметре `response_kwargs` — словарь, элементы которого будут переданы конструктору класса ответа в качестве дополнительных параметров.

  В изначальной реализации создает и возвращает экземпляр класса, указанного в атрибуте `response_class`.

10.2.4. Контроллер *TemplateView*: все вместе

Контроллер-класс `TemplateView` наследует классы `View`, `ContextMixin` и `TemplateResponseMixin`. Он автоматически выполняет рендеринг шаблона и отправку ответа при получении запроса по методу GET.

В формируемый контекст шаблона добавляются все URL-параметры, которые присутствуют в маршруте, под своими изначальными именами.

Класс `TemplateView` уже можно применять в практической работе. Например, в листинге 10.1 приведен код производного от него контроллера-класса `BbByRubricView`, который выводит страницу с объявлениями из выбранной рубрики.

Листинг 10.1. Использование контроллера-класса `TemplateView`

```
from django.views.generic.base import TemplateView
from .models import Bb, Rubric
```

```
class BbByRubricView(TemplateView):
    template_name = 'bboard/by_rubric.html'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['bbs'] = Bb.objects.filter(rubric=context['rubric_id'])
        context['rubrics'] = Rubric.objects.all()
        context['current_rubric'] = Rubric.objects.get(
                                            pk=context['rubric_id'])
        return context
```

Поскольку класс `TemplateView` добавляет в контекст шаблона значения всех полученных им URL-параметров, мы можем извлечь ключ рубрики, обратившись к элементу `rubric_id` контекста в переопределенном методе `get_context_data()`. В классах, рассматриваемых далее, такой "номер" уже не пройдет, поскольку они не наследуют от `TemplateView`.

10.3. Классы, выводящие одну запись

Контроллеры-классы из модуля `django.views.generic.detail` — более высокоуровневые, чем базовые, рассмотренные в *разд. 10.2*. Они носят название *обобщенных*, поскольку выполняют типовые задачи и могут быть использованы в различных ситуациях.

10.3.1. Примесь *SingleObjectMixin*: поиск записи

Класс-примесь `SingleObjectMixin`, наследующий от `ContextMixin`, выполняет сразу три действия:

- извлекает из полученного интернет-адреса ключ или слаг записи, предназначенной к выводу на странице;
- ищет запись в заданной модели по полученному ранее ключу или слагу;
- помещает найденную запись в контекст шаблона.

Примесь поддерживает довольно много атрибутов и методов:

- `model` — атрибут, задает модель;
- `queryset` — атрибут, указывает либо диспетчер записей (`Manager`), либо набор записей (`QuerySet`), в котором будет выполняться поиск записи;
- `get_queryset(self)` — метод, должен возвращать набор записей (`QuerySet`), в котором будет выполняться поиск записи.

  В изначальной реализации возвращает значение атрибута `queryset`, если оно задано, или набор записей из модели, заданной в атрибуте `model`;
- `pk_url_kwarg` — атрибут, задает имя URL-параметра, через который контроллер-класс получит ключ записи (по умолчанию: `"pk"`);

- `slug_field` — атрибут, задает имя поля модели, в котором хранится слаг (по умолчанию: `"slug"`);
- `get_slug_field(self)` — метод, должен возвращать строку с именем поля модели, в котором хранится слаг. В реализации по умолчанию возвращает значение из атрибута `slug_field`;
- `slug_url_kwarg` — атрибут, задает имя URL-параметра, через который контроллер-класс получит слаг (по умолчанию: `"slug"`);
- `query_pk_and_slug` — атрибут, указывает, что случится, если ключ записи не был найден в интернет-адресе. Если значение атрибута равно `False`, то в этом случае будет возбуждено исключение `AttributeError`, если `True` — будет выполнена попытка найти запись по слагу (если он присутствует — иначе также будет возбуждено исключение `AttributeError`);
- `context_object_name` — атрибут, задает имя переменной контекста шаблона, в которой будет сохранена найденная запись;
- `get_context_object_name(self, obj)` — метод, должен возвращать имя переменной контекста шаблона, в которой будет сохранена найденная запись, в виде строки. В параметре `obj` передается объект записи.

  В изначальной реализации возвращает значение атрибута `context_object_name` или, если этот атрибут хранит `None`, приведенное к нижнему регистру имя модели (так, если задана модель `Rubric`, метод вернет имя `rubric`);
- `get_object(self, queryset=None)` — метод, выполняющий поиск записи по указанным критериям и возвращающий найденную запись в качестве результата. В параметре `queryset` может быть передан набор записей, в котором должен выполняться поиск.

  В изначальной реализации ищет запись в наборе из параметра `queryset` или, если он не задан, — в наборе записей, возвращенном методом `get_queryset()`. Значения ключа и слага получает из словаря, сохраненного в атрибуте экземпляра `kwargs` (его создает метод `setup()`, описанный в *разд. 10.2.1*), а имена необходимых URL-параметров — из атрибутов `pk_url_kwarg` и `slug_url_kwarg`. Если запись не найдена, метод возбуждает исключение `Http404` из модуля `django.http`;
- `get_context_data(self, **kwargs)` — переопределенный метод, создающий и возвращающий контекст шаблона.

  В изначальной реализации требует, чтобы в экземпляре текущего контроллера-класса присутствовал атрибут `object`, хранящий найденную запись или `None`, если таковая не была найдена, а также если контроллер используется для создания новой записи. В контексте шаблона создает переменную `object` и переменную с именем, возвращенным методом `get_context_object_name()`, обе переменные хранят найденную запись.

10.3.2. Примесь *SingleObjectTemplateResponseMixin*: рендеринг шаблона на основе найденной записи

Класс-примесь `SingleObjectTemplateResponseMixin`, наследующий от `TemplateResponseMixin`, выполняет рендеринг шаблона на основе записи, найденной в модели. Он требует, чтобы в контроллере-классе присутствовал атрибут `object`, в котором хранится либо найденная запись в виде объекта модели, либо `None`, если запись не была найдена, а также если контроллер используется для создания новой записи.

Вот атрибуты и методы, поддерживаемые этим классом:

- `template_name_field` — атрибут, содержащий имя поля модели, в котором хранится путь к шаблону. Если `None`, то путь к шаблону не будет извлекаться из записи модели (поведение по умолчанию);
- `template_name_suffix` — атрибут, хранящий строку с суффиксом, который будет добавлен к автоматически сгенерированному пути к шаблону (по умолчанию: `"_detail"`);
- `get_template_names(self)` — переопределенный метод, возвращающий список путей к шаблонам, заданных в виде строк.

  В изначальной реализации возвращает список:

  - либо из пути, извлеченного из унаследованного атрибута `template_name`, если этот путь указан;
  - либо из:
    - пути, извлеченного из поля модели, имя которого хранится в атрибуте `template_name_field`, если все необходимые данные (имя поля, запись модели и сам путь в поле этой записи) указаны;
    - пути вида *<псевдоним приложения>\<имя модели><суффикс>*.html (так, для модели `Bb` из приложения `bboard` будет сформирован путь bboard\bb_detail.html).

10.3.3. Контроллер *DetailView*: все вместе

Контроллер-класс `DetailView` наследует классы `View`, `SingleObjectMixin` и `SingleObjectTemplateResponseMixin`. Он ищет запись по полученным значениям ключа или слага, заносит ее в атрибут `object` (чтобы успешно работали наследуемые им примеси) и выводит на экран страницу с содержимым этой записи.

В листинге 10.2 приведен код контроллера-класса `BbDetailView`, производного от `DetailView` и выводящего страницу с объявлением, выбранным посетителем.

Листинг 10.2. Использование контроллера-класса `DetailView`

```
from django.views.generic.detail import DetailView
from .models import Bb, Rubric
```

```
class BbDetailView(DetailView):
    model = Bb

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['rubrics'] = Rubric.objects.all()
        return context
```

Компактность кода контроллера обусловлена в том числе и тем, что он следует соглашениям. Так, в нем не указан путь к шаблону — значит, класс будет искать шаблон со сформированным по умолчанию путем bboard\bb_detail.html.

Добавим в список маршрутов маршрут, связанный с этим контроллером:

```
from .views import BbDetailView
urlpatterns = [
    . . .
    path('detail/<int:pk>/', BbDetailView.as_view(), name='detail'),
    . . .
]
```

Опять же, этот маршрут написан соответственно соглашениям — у URL-параметра, содержащего ключ записи, указано имя `pk`, используемое классом `DetailView` по умолчанию.

Теперь напишем шаблон bboard\bb_detail.html. Его код приведен в листинге 10.3.

Листинг 10.3. Код шаблона, выводящего объявление

```
{% extends "layout/basic.html" %}

{% block title %}{{ bb.title }}{% endblock %}

{% block content %}
<p>Рубрика: {{ bb.rubric.name }}</p>
<h2>{{ bb.title }}</h2>
<p>{{ bb.content }}</p>
<p>Цена: {{ bb.price }}</p>
{% endblock %}
```

По умолчанию класс `BbDetailView` создаст в контексте шаблона переменную `bb`, хранящую найденную запись (эту функциональность он унаследовал от базового класса `DetailView`). В коде шаблона мы используем эту переменную.

Осталось добавить в шаблоны bboard\index.html и bboard\by_rubric.html гиперссылки, которые будут вести на страницу выбранного объявления:

```
<h2><a href="{% url 'detail' pk=bb.pk %}">{{ bb.title }}</a></h2>
```

Как видим, применяя контроллеры-классы достаточно высокого уровня и, главное, следуя заложенным в них соглашениям, можно выполнять весьма сложные действия без написания громоздкого кода.

10.4. Классы, выводящие наборы записей

Обобщенные классы из модуля `django.views.generic.list` выводят на экран целый набор записей.

10.4.1. Примесь *MultipleObjectMixin*: извлечение набора записей

Класс-примесь `MultipleObjectMixin`, наследующий от `ContextMixin`, извлекает из модели набор записей, возможно, отфильтрованный, отсортированный и разбитый на части посредством пагинатора (о пагинаторе разговор пойдет в *главе 12*). Полученный набор записей он помещает в контекст шаблона.

Номер части, которую нужно извлечь, передается в составе интернет-адреса, через URL- или GET-параметр `page`. Номер должен быть целочисленным и начинаться с 1. Если это правило нарушено, то будет возбуждено исключение `Http404`. Допустимо также указывать значение `"last"`, обозначающее последнюю часть.

Примесь поддерживает следующие атрибуты и методы:

- `model` — атрибут, задает модель;
- `queryset` — атрибут, указывает либо диспетчер записей (`Manager`), либо исходный набор записей (`QuerySet`), из которого будут извлекаться записи;
- `get_queryset(self)` — метод, должен возвращать исходный набор записей (`QuerySet`), из которого будут извлекаться записи.

  В изначальной реализации возвращает значение атрибута `queryset`, если оно задано, или набор записей из модели, которая задана в атрибуте `model`;
- `ordering` — атрибут, задающий параметры сортировки записей. Значение может быть указано в виде:
  - строки с именем поля — для сортировки только по этому полю. По умолчанию будет выполняться сортировка по возрастанию значения поля. Чтобы указать сортировку по убыванию, нужно предварить имя поля символом "минус";
  - последовательности строк с именами полей — для сортировки сразу по нескольким полям.

  Если значение атрибута не указано, станет выполняться сортировка, заданная в параметрах модели, или, когда таковое не указано, записи сортироваться не будут;
- `get_ordering(self)` — метод, должен возвращать параметры сортировки записей. В изначальной реализации возвращает значение атрибута `ordering`;

- `paginate_by` — атрибут, задающий целочисленное количество записей в одной части пагинатора. Если не указан или его значение равно `None`, набор записей не будет разбиваться на части;
- `get_paginate_by(self, queryset)` — метод, должен возвращать число записей набора, полученного в параметре `queryset`, помещающихся в одной части пагинатора. В изначальной реализации просто возвращает значение из атрибута `paginate_by`;
- `page_kwarg` — атрибут, указывающий имя URL- или GET-параметра, через который будет передаваться номер выводимой части пагинатора, в виде строки (по умолчанию: `"page"`);
- `paginate_orphans` — атрибут, задающий целочисленное минимальное число записей, которые могут присутствовать в последней части пагинатора. Если последняя часть пагинатора содержит меньше записей, то оставшиеся записи будут выведены в предыдущей части. Если задать значение 0, то в последней части может присутствовать сколько угодно записей (поведение по умолчанию);
- `get_paginate_orphans(self)` — метод, должен возвращать минимальное число записей, помещающихся в последней части пагинатора. В изначальной реализации возвращает значение атрибута `paginate_orphans`;
- `allow_empty` — атрибут. Значение `True` разрешает извлечение "пустой", т. е. не содержащей ни одной записи, части пагинатора (поведение по умолчанию). Значение `False`, напротив, предписывает при попытке извлечения "пустой" части возбуждать исключение `Http404`;
- `get_allow_empty(self)` — метод, должен возвращать `True`, если разрешено извлечение "пустой" части пагинатора, или `False`, если такое недопустимо. В изначальной реализации возвращает значение атрибута `allow_empty`;
- `paginator_class` — атрибут, указывающий класс используемого пагинатора (по умолчанию: `Paginator` из модуля `django.core.paginator`);
- `get_paginator()` — метод, должен создавать объект пагинатора и возвращать его в качестве результата. Формат объявления:

```
get_paginator(self, queryset, per_page, orphans=0,
                                        allow_empty_first_page=True)
```

  Параметр `queryset` хранит обрабатываемый пагинатором набор записей, параметр `per_page` — число записей в части, `orphans` — минимальное число записей в последней части пагинатора, а `allow_empty_first_page` указывает, разрешено извлечение "пустой" части (`True`) или нет (`False`).

  В изначальной реализации создает экземпляр класса пагинатора из атрибута `paginator_class`, передавая его конструктору все полученные им параметры;
- `paginate_queryset(self, queryset, page_size)` — метод, разбивает набор записей, полученный в параметре `queryset`, на части с указанным в параметре `page_size` числом записей в каждой части и возвращает кортеж из четырех элементов:

- объекта самого пагинатора;
- объекта его текущей части, номер которой был получен с URL- или GET-параметром;
- набора записей, входящих в текущую часть (извлекается из атрибута `object_list` объекта текущей части пагинатора);
- `True`, если извлеченный набор записей действительно был разбит на части с применением пагинатора, и `False` — в противном случае;

- `context_object_name` — атрибут, задает имя переменной контекста шаблона, в которой будет сохранен извлеченный набор записей;
- `get_context_object_name(self, object_list)` — метод, должен возвращать строку с именем переменной контекста шаблона, в которой будет сохранен набор записей, полученный в параметре `object_list`.

  В изначальной реализации возвращает имя из атрибута `context_object_name` или, если оно не указано, приведенное к нижнему регистру имя модели с добавленным суффиксом `_list`;
- `get_context_data(self, **kwargs)` — переопределенный метод, создающий и возвращающий контекст шаблона.

  В изначальной реализации извлекает набор записей из необязательного параметра `object_list` или, если этот параметр не указан, из атрибута `object_list`. После чего возвращает контекст с пятью переменными:
  - `object_list` — выводимый на странице набор записей (если используется пагинатор, это будет набор записей из его текущей части);
  - переменная с именем, возвращенным методом `get_context_object_name()`, — то же самое;
  - `is_paginated` — `True`, если применялся пагинатор, и `False` — в противном случае;
  - `paginator` — объект пагинатора или `None`, если пагинатор не применялся;
  - `page_obj` — объект текущей страницы пагинатора или `None`, если пагинатор не применялся.

10.4.2. Примесь *MultipleObjectTemplateResponseMixin*: рендеринг шаблона на основе набора записей

Класс-примесь `MultipleObjectTemplateResponseMixin`, наследующий от `TemplateResponseMixin`, выполняет рендеринг шаблона на основе извлеченного из модели набора записей. Он требует, чтобы в контроллере-классе присутствовал атрибут `object_list`, в котором хранится набор записей.

Вот список атрибутов и методов, поддерживаемых им:

- `template_name_suffix` — атрибут, хранящий строку с суффиксом, который будет добавлен к автоматически сгенерированному пути к шаблону (по умолчанию: `"_list"`);

- `get_template_names(self)` — переопределенный метод, возвращающий список путей к шаблонам, заданных в виде строк.

  В изначальной реализации возвращает список из:

  - пути, полученного из унаследованного атрибута `template_name`, если этот путь указан;
  - пути вида *<псевдоним приложения>\<имя модели><суффикс>*.html (так, для модели `Bb` из приложения `bboard` будет сформирован путь bboard\bb_list.html).

10.4.3. Контроллер *ListView*: все вместе

Контроллер-класс `ListView` наследует классы `View`, `MultipleObjectMixin` и `MultipleObjectTemplateResponseMixin`. Он извлекает из модели набор записей, записывает его в атрибут `object_list` (чтобы успешно работали наследуемые им примеси) и выводит на экран страницу со списком записей.

В листинге 10.4 приведен код контроллера-класса `BbByRubricView`, унаследованного от `ListView` и выводящего страницу с объявлениями из выбранной посетителем рубрики.

Листинг 10.4. Использование контроллера-класса `ListView`

```
from django.views.generic.list import ListView
from .models import Bb, Rubric

class BbByRubricView(ListView):
    template_name = 'bboard/by_rubric.html'
    context_object_name = 'bbs'

    def get_queryset(self):
        return Bb.objects.filter(rubric=self.kwargs['rubric_id'])

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['rubrics'] = Rubric.objects.all()
        context['current_rubric'] = Rubric.objects.get(
                                        pk=self.kwargs['rubric_id'])
        return context
```

Код этого контроллера получился более объемным, чем у ранее написанного контроллера-функции `by_rubric()` (см. листинг 2.2). Это обусловлено особенностями используемого нами шаблона bboard\by_rubric.html (см. листинг 2.3). Во-первых, требуется вывести список рубрик и текущую рубрику, следовательно, придется добавить все эти данные в контекст шаблона, переопределив метод `get_context_data()`. Во-вторых, мы используем уже имеющийся шаблон, поэтому вынуждены указать в контроллере его имя и имя переменной контекста, в которой будет храниться список объявлений.

Значение URL-параметра `rubric_id` мы получили обращением к словарю из атрибута `kwargs`, содержащему все URL-параметры, указанные в маршруте. Из контекста шаблона извлечь его мы не можем.

10.5. Классы, работающие с формами

Обобщенные контроллеры-классы из модуля `django.views.generic.edit` рассчитаны на работу с формами, как связанными с моделями, так и обычными (формы, связанные с моделями, будут описаны в *главе 13*, а обычные — в *главе 17*).

10.5.1. Классы для вывода и валидации форм

Классы самого низкого уровня "умеют" лишь вывести форму, проверить занесенные в нее данные на корректность и, в случае ошибки, вывести повторно, вместе с предупреждающими сообщениями.

10.5.1.1. Примесь *FormMixin*: создание формы

Класс-примесь `FormMixin`, производный от класса `ContextMixin`, создает форму (неважно, связанную с моделью или обычную), проверяет введенные в нее данные, выполняет перенаправление, если данные прошли проверку, или выводит форму повторно (в противном случае).

Вот набор поддерживаемых атрибутов и методов:

- `form_class` — атрибут, хранит ссылку на класс используемой формы;
- `get_form_class(self)` — метод, должен возвращать ссылку на класс используемой формы. В изначальной реализации возвращает значение атрибута `form_class`;
- `initial` — атрибут, хранящий словарь с изначальными данными для занесения в только что созданную форму. Ключи элементов этого словаря должны соответствовать полям формы, а значения элементов зададут значения полей. По умолчанию хранит пустой словарь;
- `get_initial(self)` — метод, должен возвращать словарь с изначальными данными для занесения в только что созданную форму. В изначальной реализации просто возвращает значение атрибута `initial`;
- `success_url` — атрибут, хранит интернет-адрес для перенаправления, если введенные в форму данные прошли проверку на корректность;
- `get_success_url(self)` — метод, должен возвращать интернет-адрес для перенаправления в случае, если введенные в форму данные прошли валидацию. В изначальной реализации возвращает значение атрибута `success_url`;
- `prefix` — атрибут, задает строковый префикс для имени формы, который будет присутствовать в создающем форму HTML-коде. Префикс стоит задавать только в том случае, если планируется поместить несколько однотипных форм в одном теге `<form>`. По умолчанию хранит значение `None` (отсутствие префикса);

- `get_prefix(self)` — метод, должен возвращать префикс для имени формы. В изначальной реализации возвращает значение из атрибута `prefix`;
- `get_form(self, form_class=None)` — метод, создающий и возвращающий объект формы.

  В изначальной реализации, если класс формы указан в параметре `form_class`, создает экземпляр этого класса, в противном случае — экземпляр класса, возвращенного методом `get_form_class()`. При этом конструктору класса формы передаются параметры, возвращенные методом `get_form_kwargs()`;
- `get_form_kwargs(self)` — метод, должен создавать и возвращать словарь с параметрами, которые будут переданы конструктору класса формы в методе `get_form()`.

  В изначальной реализации возвращает словарь с элементами:

  - `initial` — словарь с изначальными данными, возвращенный методом `get_initial()`;
  - `prefix` — префикс для имени формы, возвращенный методом `get_prefix()`;

  Следующие два элемента создаются только в том случае, если для отправки запроса применялись HTTP-методы POST и PUT (т. е. при проверке введенных в форму данных):

  - `data` — словарь с данными, занесенными в форму посетителем;
  - `files` — словарь с файлами, отправленными посетителем из формы;
- `get_context_data(self, **kwargs)` — переопределенный метод, создающий и возвращающий контекст шаблона.

  В изначальной реализации добавляет в контекст шаблона переменную `form`, хранящую созданную форму;
- `form_valid(self, form)` — метод, должен выполнять обработку данных, введенных в переданную через параметр `form` форму, в том случае, если они прошли валидацию.

  В изначальной реализации просто выполняет перенаправление по адресу, возвращенному методом `get_success_url()`;
- `form_invalid(self, form)` — метод, должен выполнять обработку ситуации, когда данные, введенные в переданную через параметр `form` форму, не проходят валидацию. В изначальной реализации повторно выводит страницу с формой на экран.

10.5.1.2. Контроллер *ProcessFormView*: вывод и обработка формы

Контроллер-класс `ProcessFormView`, производный от класса `View`, выводит на экран страницу с формой, принимает введенные данные и проводит их валидацию.

Он переопределяет три метода, унаследованные от базового класса:

- `get(self, request, *args, **kwargs)` — выводит страницу с формой на экран;
- `post(self, request, *args, **kwargs)` — получает введенные в форму данные и выполняет их валидацию. Если валидация прошла успешно, вызывает метод `form_valid()`, в противном случае — метод `form_invalid()` (см. *разд. 10.5.1.1*);
- `put(self, request, *args, **kwargs)` — то́ же, что и `post()`.

10.5.1.3. Контроллер-класс *FormView*: создание, вывод и обработка формы

Контроллер-класс `FormView`, производный от `FormMixin`, `ProcessFormView` и `TemplateResponseMixin`, создает форму, выводит на экран страницу с этой формой, проверяет на корректность введенные данные и, в случае отрицательного результата проверки, выводит страницу с формой повторно. Нам остается только реализовать обработку корректных данных, переопределив метод `form_valid()`.

В листинге 10.5 приведен код контроллера-класса `BbAddView`, добавляющего на виртуальную доску новое объявление.

Листинг 10.5. Использование контроллера-класса `FormView`

```
from django.views.generic.edit import FormView
from django.urls import reverse
from .models import Bb, Rubric

class BbAddView(FormView):
    template_name = 'bboard/create.html'
    form_class = BbForm
    initial = {'price': 0.0}

    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(*args, **kwargs)
        context['rubrics'] = Rubric.objects.all()
        return context

    def form_valid(self, form):
        form.save()
        return super().form_valid(form)

    def get_form(self, form_class=None):
        self.object = super().get_form(form_class)
        return self.object

    def get_success_url(self):
        return reverse('bboard:by_rubric',
            kwargs={'rubric_id': self.object.cleaned_data['rubric'].pk})
```

При написании этого класса мы столкнемся с проблемой. Чтобы после сохранения объявления сформировать интернет-адрес для перенаправления, нам нужно получить значение ключа рубрики, к которой относится добавленное объявление. Поэтому мы переопределили метод `get_form()`, в котором сохранили созданную форму в атрибуте `object`. После этого в коде метода `get_success_url()` без проблем сможем получить доступ к форме и занесенным в нее данным.

Сохранение введенных в форму данных мы выполняем в переопределенном методе `form_valid()`.

10.5.2. Классы для добавления, правки и удаления записей

Описанные далее высокоуровневые классы, помимо обработки форм, выполняют добавление, правку и удаление записей.

10.5.2.1. Примесь *ModelFormMixin*: создание формы, связанной с моделью

Класс-примесь `ModelFormMixin`, наследующий от классов `SingleObjectMixin` и `FormMixin`, полностью аналогичен последнему, но работает с формами, связанными с моделями.

Вот список поддерживаемых им атрибутов и методов:

- `model` — атрибут, задает ссылку на класс модели, на основе которой будет создана форма;
- `fields` — атрибут, указывает последовательность имен полей модели, которые должны присутствовать в форме.

  Можно указать либо модель и список ее полей в атрибутах `model` и `fields`, либо непосредственно класс формы в атрибуте `form_class`, унаследованном от класса `FormMixin`, но никак не одновременно и то, и другое.

  Если указан атрибут `model`, обязательно следует задать также и атрибут `fields`. Если этого не сделать, возникнет ошибка;
- `get_form_class(self)` — переопределенный метод, должен возвращать ссылку на класс используемой формы.

  В изначальной реализации возвращает значение атрибута `form_class`, если оно задано. В противном случае возвращается ссылка на класс формы, автоматически созданный на основе модели, которая взята из атрибута `model` или извлечена из набора записей, заданного в унаследованном атрибуте `queryset`. Для создания класса формы также используется список полей из атрибута `fields`;
- `success_url` — атрибут, хранит интернет-адрес для перенаправления, если введенные в форму данные прошли проверку на корректность.

  В отличие от одноименного атрибута базового класса `FormMixin`, он поддерживает указание непосредственно в строке с интернет-адресом специальных после-

довательностей символов вида {*<имя поля таблицы в базе данных>*}. Вместо такой последовательности будет подставлено значение поля с указанным *именем*.

Отметим, что в такие последовательности должно подставляться имя не поля модели, а таблицы базы данных, которая обрабатывается моделью. Так, для вставки в адрес ключа записи следует использовать поле `id`, а не `pk`, а для вставки внешнего ключа — поле `rubric_id`, а не `rubric`.

Примеры:

```
class BbCreateView(CreateView):
    . . .
    success_url = '/bboard/detail/{id}'

class BbCreateView(CreateView):
    . . .
    success_url = '/bboard/{rubric_id}'
```

- `get_success_url(self)` — переопределенный метод, возвращает адрес для перенаправления в случае, если введенные данные прошли валидацию.

  В изначальной реализации возвращает значение атрибута `success_url`, в котором последовательности вида {*<имя поля таблицы в базе данных>*} уже заменены значениями соответствующих полей. Если адрес там не указан, пытается получить его вызовом метода `get_absolute_url()` модели;

- `get_form_kwargs(self)` — переопределенный метод, создает и возвращает словарь с параметрами, которые будут переданы конструктору класса формы в унаследованном методе `get_form()`.

  В изначальной реализации добавляет в словарь, сформированный унаследованным методом, элемент `instance`, хранящий обрабатываемую формой запись модели (если она существует, т. е. форма используется не для добавления записи). Эта запись извлекается из атрибута `object`;

- `form_valid(self, form)` — переопределенный метод, должен выполнять обработку данных, введенных в переданную через параметр `form` форму, в том случае, если они прошли валидацию.

  В изначальной реализации сохраняет содержимое формы в модели, вызвав у нее метод `save()`, присваивает новую запись атрибуту `object`, после чего вызывает унаследованный метод `form_valid()`.

10.5.2.2. Контроллер *CreateView*: создание новой записи

Контроллер-класс `CreateView` наследует от классов `ProcessFormView`, `ModelFormMixin` и `SingleObjectTemplateResponseMixin`. Он выводит форму, проверяет введенные в нее данные и создает на их основе новую запись.

Атрибут `template_name_suffix` этого класса хранит строку с суффиксом, который будет добавлен к автоматически сгенерированному пути к шаблону (по умолчанию: `"_form"`).

Также в классе доступен атрибут `object`, в котором хранится созданная в модели запись или `None`, если таковая еще не была создана.

Пример контроллера-класса, производного от `CreateView`, можно увидеть в листинге 2.7.

10.5.2.3. Контроллер *UpdateView*: исправление записи

Контроллер-класс `UpdateView` наследует от классов `ProcessFormView`, `ModelFormMixin` и `SingleObjectTemplateResponseMixin`. Он ищет запись по полученным из URL-параметра ключу или слагу, выводит страницу с формой для ее правки, проверяет и сохраняет исправленные данные.

Атрибут `template_name_suffix` этого класса хранит строку с суффиксом, который будет добавлен к автоматически сгенерированному пути к шаблону (по умолчанию: `"_form"`).

Также в классе доступен атрибут `object`, в котором хранится исправляемая запись.

Поскольку класс `UpdateView` предварительно выполняет поиск записи, в нем необходимо указать модель (в унаследованном атрибуте `model`), набор записей (в атрибуте `queryset`) или переопределить метод `get_queryset()`.

В листинге 10.6 приведен код контроллера-класса `BbEditView`, который выполняет исправление объявления.

Листинг 10.6. Использование контроллера-класса `UpdateView`

```
from django.views.generic.edit import UpdateView
from .models import Bb, Rubric

class BbEditView(UpdateView):
    model = Bb
    form_class = BbForm
    success_url = '/'

    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(*args, **kwargs)
        context['rubrics'] = Rubric.objects.all()
        return context
```

Шаблон bboard\bb_form.html вы можете написать самостоятельно, взяв за основу уже имеющийся шаблон bboard\create.html (там всего лишь придется поменять текст **Добавление** на **Исправление**, а подпись кнопки — с **Добавить** на **Сохранить**). Также самостоятельно вы можете записать маршрут для этого контроллера и вставить в шаблоны bboard\index.html и bboard\by_rubric.html код, создающий гиперссылки на страницы исправления объявлений.

10.5.2.4. Примесь *DeletionMixin*: удаление записи

Класс-примесь `DeletionMixin` добавляет наследующему ее контроллеру инструменты для удаления записи. Он предполагает, что запись, подлежащая удалению, уже найдена и сохранена в атрибуте `object`.

Класс объявляет атрибут `success_url` и метод `get_success_url()`, аналогичные присутствующим в примеси `ModelFormMixin` (см. *разд. 10.5.2.1*).

10.5.2.5. Контроллер *DeleteView*: удаление записи с подтверждением

Контроллер-класс `DeleteView` наследует от классов `DetailView`, `DeletionMixin` и `SingleObjectTemplateResponseMixin`. Он ищет запись по полученному из URL-параметра ключу или слагу, выводит страницу подтверждения, включающую в себя форму с кнопкой удаления, и удаляет запись.

Атрибут `template_name_suffix` этого класса хранит строку с суффиксом, который будет добавлен к автоматически сгенерированному пути к шаблону (по умолчанию: `"_confirm_delete"`).

Также в классе доступен атрибут `object`, в котором хранится удаляемая запись.

Поскольку класс `DeleteView` предварительно выполняет поиск записи, в нем необходимо указать модель (в атрибуте `model`), набор записей (в атрибуте `queryset`) или переопределить метод `get_queryset()`.

В листинге 10.7 приведен код контроллера-класса `BbDeleteView`, который выполняет удаление объявления.

Листинг 10.7. Использование контроллера-класса `DeleteView`

```
from django.views.generic.edit import DeleteView
from .models import Bb, Rubric

class BbDeleteView(DeleteView):
    model = Bb
    success_url = '/'

    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(*args, **kwargs)
        context['rubrics'] = Rubric.objects.all()
        return context
```

Код шаблона bboard\bb_confirm_delete.html, выводящего страницу подтверждения и форму с кнопкой **Удалить**, приведен в листинге 10.8.

Листинг 10.8. Код шаблона, выводящего страницу удаления записи

```
{% extends "layout/basic.html" %}

{% block title %}Удаление объявления{% endblock %}
```

```
{% block content %}
<h2>Удаление объявления</h2>
<p>Рубрика: {{ bb.rubric.name }}</p>
<p>Товар: {{ bb.title }}</p>
<p>{{ bb.content }}</p>
<p>Цена: {{ bb.price }}</p>
<form method="post">
    {% csrf_token %}
    <input type="submit" value="Удалить">
</form>
{% endblock %}
```

10.6. Классы для вывода хронологических списков

Обобщенные классы из модуля `django.views.generic.dates` выводят хронологические списки: за определенный год, месяц, неделю или дату, за текущее число.

10.6.1. Вывод последних записей

Рассматриваемые далее классы выводят хронологические списки наиболее "свежих" записей.

10.6.1.1. Примесь *DateMixin*: фильтрация записей по дате

Класс-примесь `DateMixin` предоставляет наследующим контроллерам возможность фильтровать записи по значениям даты (или временно́й отметки), хранящимся в заданном поле.

Вот атрибуты и методы, поддерживаемые этим классом:

- `date_field` — атрибут, указывает имя поля модели типа `DateField` или `DateTimeField`, по которому будет выполняться фильтрация записей. Имя поля должно быть задано в виде строки;
- `get_date_field(self)` — метод, должен возвращать имя поля модели, по которому будет выполняться фильтрация записей. В изначальной реализации возвращает значение атрибута `date_field`;
- `allow_future` — атрибут. Значение `True` указывает включить в результирующий набор записи, у которых возвращенное методом `get_date_field()` поле хранит дату из будущего. Значение `False` запрещает включать такие записи в набор (поведение по умолчанию);
- `get_allow_future(self)` — метод, должен возвращать значение `True` или `False`, говорящее, следует ли включать в результирующий набор "будущие" записи. В изначальной реализации возвращает значение атрибута `allow_future`.

10.6.1.2. Контроллер *BaseDateListView*: базовый класс

Контроллер-класс `BaseDateListView` наследует от классов `View`, `MultipleObjectMixin` и `DateMixin`. Он предоставляет базовую функциональность для других, более специализированных классов, в частности, задает сортировку записей по убыванию значения поля, возвращенного методом `get_date_field()` класса `DateMixin` (см. *разд. 10.6.1.1*).

В классе объявлены такие атрибуты и методы:

- `allow_empty` — атрибут. Значение `True` разрешает извлечение "пустой", т. е. не содержащей ни одной записи, части пагинатора. Значение `False`, напротив, предписывает при попытке извлечения "пустой" части возбудить исключение `Http404` (поведение по умолчанию);
- `date_list_period` — атрибут, указывающий, по какой части следует урезать значения даты. Должен содержать значение `"year"` (год, значение по умолчанию), `"month"` (месяц) или `"day"` (число, т. е. дата не будет урезаться);
- `get_date_list_period(self)` — метод, должен возвращать обозначение части, по которой нужно урезать дату. В изначальной реализации возвращает значение атрибута `date_list_period`;
- `get_dated_items(self)` — метод, должен возвращать кортеж из трех элементов:
  - список значений дат, хранящихся в записях из полученного набора;
  - сам набор записей;
  - словарь, элементы которого будут добавлены в контекст шаблона.

  В изначальной реализации возбуждает исключение `NotImplementedError`. Предназначен для переопределения в подклассах;
- `get_dated_queryset(self, **lookup)` — метод, возвращает набор записей, отфильтрованный согласно заданным условиям, которые в виде словаря передаются в параметре `lookup`;
- `get_date_list(self, queryset, date_type=None, ordering='ASC')` — метод, возвращает список значений даты, урезанной по части, что задана в параметре `date_type` (если он отсутствует, будет использовано значение, возвращенное методом `get_date_list_period()`), для которых существуют записи в наборе, заданном в параметре `queryset`. Параметр `ordering` задает направление сортировки: `"ASC"` (по возрастанию, поведение по умолчанию) или `"DESC"` (по убыванию).

Класс добавляет в контекст шаблона два дополнительных элемента:

- `object_list` — результирующий набор записей;
- `date_list` — список урезанных значений дат, хранящихся в записях из полученного набора.

10.6.1.3. Контроллер *ArchiveIndexView*: вывод последних записей

Контроллер-класс `ArchiveIndexView` наследует от классов `BaseDateListView` и `MultipleObjectTemplateResponseMixin`. Он выводит хронологический список записей, отсортированных по убыванию значения заданного поля.

Для хранения результирующего набора выводимых записей в контексте шаблона создается переменная `latest`. В переменной `date_list` контекста шаблона хранится список значений дат, урезанных до года. К автоматически сгенерированному пути к шаблону по умолчанию добавляется суффикс `_archive`.

Листинг 10.9 содержит код контроллера-класса `BbIndexView`, основанного на классе `ArchiveIndexView`. Для вывода страницы он может использовать шаблон bboard\index.html, написанный нами в *главах 1* и *2*,

Листинг 10.9. Применение контроллера-класса `ArchiveIndexView`

```
from django.views.generic.dates import ArchiveIndexView
from .models import Bb, Rubric

class BbIndexView(ArchiveIndexView):
    model = Bb
    date_field = 'published'
    date_list_period = 'year'
    template_name = 'bboard/index.html'
    context_object_name = 'bbs'
    allow_empty = True

    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(*args, **kwargs)
        context['rubrics'] = Rubric.objects.all()
        return context
```

А вот так мы можем использовать хранящийся в переменной `date_list` контекста шаблона список дат, урезанных до года:

```
<p>
    {% for d in date_list %}
    {{ d.year }}
    {% endfor %}
</p>
```

В результате на экране появится список разделенных пробелами годов, за которые в наборе имеются записи.

10.6.2. Вывод записей по годам

Следующая пара классов выводит список записей, относящихся к определенному году.

10.6.2.1. Примесь *YearMixin*: извлечение года

Класс-примесь `YearMixin` извлекает из URL- или GET-параметра с именем `year` значение года, которое будет использовано для последующей фильтрации записей.

Этот класс поддерживает следующие атрибуты и методы:

- `year_format` — атрибут, указывает строку с форматом значения года, поддерживаемым функцией `strftime()` языка Python. Значение года будет извлекаться из URL- или GET-параметра и преобразовываться в нужный вид согласно этому формату. Значение по умолчанию: `"%Y"` (год из четырех цифр);
- `get_year_format(self)` — метод, должен возвращать строку с форматом значения года. В изначальной реализации возвращает значение атрибута `year_format`;
- `year` — атрибут, задает значение года в виде строки. Если `None`, то год будет извлекаться из URL- или GET-параметра (поведение по умолчанию);
- `get_year(self)` — метод, должен возвращать значение года в виде строки. В изначальной реализации пытается вернуть значение (если оно задано): атрибута класса `year`, URL-параметра `year`, GET-параметра `year`. Если все попытки завершились неудачами, то возбуждает исключение `Http404`;
- `get_previous_year(self, date)` — метод, возвращает значение даты, представляющей собой первый день года, который предшествует дате из параметра `date`. В зависимости от значений атрибутов класса `allow_empty` и `allow_future` может возбуждать исключение `Http404`;
- `get_next_year(self, date)` — метод, возвращает значение даты, представляющей собой первый день года, который следует за датой из параметра `date`. В зависимости от значений атрибутов класса `allow_empty` и `allow_future` может возбуждать исключение `Http404`.

10.6.2.2. Контроллер *YearArchiveView*: вывод записей за год

Контроллер-класс `YearArchiveView` наследует классы `BaseDateListView`, `YearMixin` и `MultipleObjectTemplateResponseMixin`. Он выводит хронологический список записей, относящихся к указанному году и отсортированных по возрастанию значения заданного поля.

Класс поддерживает дополнительные атрибут и метод:

- `make_object_list` — атрибут. Если `True`, то будет сформирован и добавлен в контекст шаблона набор всех записей за заданный год. Если `False`, то в контексте шаблона будет присутствовать "пустой" набор записей (поведение по умолчанию);

- `get_make_object_list(self)` — метод, должен возвращать логический признак того, формировать ли полноценный набор записей, относящихся к заданному году. В изначальной реализации возвращает значение атрибута `make_object_list`.

Набор записей, относящихся к заданному году, будет храниться в переменной контекста шаблона, имеющей имя по умолчанию: `object_list`.

Кроме того, в контексте шаблона будут созданы следующие переменные:

- `date_list` — список значений дат, за которые в наборе существуют записи, урезанных до месяца и выстроенных в порядке возрастания;
- `year` — объект типа `date`, представляющий заданный год;
- `previous_year` — объект типа `date`, представляющий предыдущий год;
- `next_year` — объект типа `date`, представляющий следующий год.

К автоматически сгенерированному пути к шаблону по умолчанию добавляется суффикс `_archive_year`.

10.6.3. Вывод записей по месяцам

Следующие два класса выводят список записей, относящихся к определенному месяцу определенного года.

10.6.3.1. Примесь *MonthMixin*: извлечение месяца

Класс-примесь `MonthMixin` извлекает из URL- или GET-параметра с именем `month` значение месяца, которое будет использовано для последующей фильтрации записей.

Поддерживаются следующие атрибуты и методы:

- `month_format` — атрибут, указывает строку с форматом значения месяца, поддерживаемым функцией `strftime()` языка Python. Значение месяца будет извлекаться из URL- или GET-параметра и преобразовываться в нужный вид согласно этому формату. Значение по умолчанию: `"%b"` (сокращенное наименование, записанное согласно текущим языковым настройкам);
- `get_month_format(self)` — метод, должен возвращать строку с форматом значения месяца. В изначальной реализации возвращает значение атрибута `month_format`;
- `month` — атрибут, задает значение месяца в виде строки. Если `None`, то месяц будет извлекаться из URL- или GET-параметра (поведение по умолчанию);
- `get_month(self)` — метод, должен возвращать значение месяца в виде строки. В изначальной реализации пытается вернуть значение (если оно задано): атрибута класса `month`, URL-параметра `month`, GET-параметра `month`. Если все попытки завершились неудачами, то возбуждает исключение `Http404`;
- `get_previous_month(self, date)` — метод, возвращает значение даты, представляющей собой первый день месяца, который предшествует дате из параметра

date. В зависимости от значений атрибутов класса allow_empty и allow_future может возбуждать исключение Http404;

- get_next_month(self, date) — метод, возвращает значение даты, представляющей собой первый день месяца, который следует за датой из параметра date. В зависимости от значений атрибутов класса allow_empty и allow_future может возбуждать исключение Http404.

10.6.3.2. Контроллер *MonthArchiveView*: вывод записей за месяц

Контроллер-класс MonthArchiveView наследует классы BaseDateListView, MonthMixin, YearMixin и MultipleObjectTemplateResponseMixin. Он выводит хронологический список записей, относящихся к указанному месяцу указанного года.

Набор записей, относящихся к заданному месяцу, будет храниться в переменной контекста шаблона, имеющей имя по умолчанию: object_list.

Кроме того, в контексте шаблона будут созданы следующие переменные:

- date_list — список значений дат, за которые в наборе существуют записи, урезанных до значения числа и выстроенных в порядке возрастания;
- month — объект типа date, представляющий заданный месяц;
- previous_month — объект типа date, представляющий предыдущий месяц;
- next_month — объект типа date, представляющий следующий месяц.

К автоматически сгенерированному пути к шаблону по умолчанию добавляется суффикс _archive_month.

Вот небольшой пример использования контроллера MonthArchiveView:

```
urlpatterns = [
    # Пример интернет-адреса: /2019/12/
    path('<int:year>/<int:month>/', BbMonthArchiveView.as_view()),
]
. . .
class BbMonthArchiveView(MonthArchiveView):
    model = Bb
    date_field = "published"
    month_format = '%m'            # Порядковый номер месяца
```

10.6.4. Вывод записей по неделям

Эти классы выводят список записей, которые относятся к неделе с заданным порядковым номером.

10.6.4.1. Примесь *WeekMixin*: извлечение номера недели

Класс-примесь WeekMixin извлекает из URL- или GET-параметра с именем week номер недели, который будет использован для фильтрации записей.

Поддерживаются такие атрибуты и методы:

- `week_format` — атрибут, указывает строку с форматом номера недели, поддерживаемым функцией `strftime()` языка Python. Значение номера недели будет извлекаться из URL- или GET-параметра и преобразовываться в нужный вид согласно этому формату. Значение по умолчанию: `"%U"` (номер недели, если первый день недели — воскресенье).

  Для обработки более привычного формата номера недели, когда первым днем является понедельник, нужно указать в качестве значения формата строку `"%W"`;

- `get_week_format(self)` — метод, должен возвращать строку с форматом значения недели. В изначальной реализации возвращает значение атрибута `week_format`;

- `week` — атрибут, задает значение недели в виде строки. Если `None`, то номер недели будет извлекаться из URL- или GET-параметра (поведение по умолчанию);

- `get_week(self)` — метод, должен возвращать номер недели в виде строки. В изначальной реализации пытается вернуть значение (если оно задано): атрибута класса `week`, URL-параметра `week`, GET-параметра `week`. Если все попытки завершились неудачами, то возбуждает исключение `Http404`;

- `get_previous_week(self, date)` — метод, возвращает значение даты, представляющей собой первый день недели, которая предшествует дате из параметра `date`. В зависимости от значений атрибутов класса `allow_empty` и `allow_future` может возбуждать исключение `Http404`;

- `get_next_week(self, date)` — метод, возвращает значение даты, представляющей собой первый день недели, которая следует за датой из параметра `date`. В зависимости от значений атрибутов класса `allow_empty` и `allow_future` может возбуждать исключение `Http404`.

10.6.4.2. Контроллер *WeekArchiveView*: вывод записей за неделю

Контроллер-класс `WeekArchiveView` наследует классы `BaseDateListView`, `WeekMixin`, `YearMixin` и `MultipleObjectTemplateResponseMixin`. Он выводит хронологический список записей, относящихся к указанной неделе указанного года.

Набор записей, относящихся к заданной неделе, будет храниться в переменной контекста шаблона, имеющей имя по умолчанию: `object_list`.

В контексте шаблона также будут созданы дополнительные переменные:

- `week` — объект типа `date`, представляющий заданную неделю;

- `previous_week` — объект типа `date`, представляющий предыдущую неделю;

- `next_week` — объект типа `date`, представляющий следующую неделю.

К автоматически сгенерированному пути к шаблону по умолчанию добавляется суффикс `_archive_week`.

Пример использования контроллера `WeekArchiveView`:

```
urlpatterns = [
    # Пример интернет-адреса: /2019/week/48/
    path('<int:year>/week/<int:week>/',
         WeekArchiveView.as_view(model=Bb, date_field="published")),
]
```

10.6.5. Вывод записей по дням

Следующие два класса выводят записи, относящиеся к определенному числу заданных месяца и года.

10.6.5.1. Примесь *DayMixin*: извлечение заданного числа

Класс-примесь `DayMixin` извлекает из URL- или GET-параметра с именем `day` число, которое будет использовано для фильтрации записей.

Атрибуты и методы, поддерживаемые классом, таковы:

- `day_format` — атрибут, указывает строку с форматом числа, поддерживаемым функцией `strftime()` языка Python. Значение числа будет извлекаться из URL- или GET-параметра и преобразовываться в нужный вид согласно этому формату. Значение по умолчанию: `"%d"` (число с начальным нулем);
- `get_day_format(self)` — метод, должен возвращать строку с форматом значения числа. В изначальной реализации возвращает значение атрибута `day_format`;
- `day` — атрибут, задает значение числа в виде строки. Если `None`, то число будет извлекаться из URL- или GET-параметра (поведение по умолчанию);
- `get_day(self)` — метод, должен возвращать значение числа в виде строки. В изначальной реализации пытается вернуть значение (если оно задано): атрибута класса `day`, URL-параметра `day`, GET-параметра `day`. Если все попытки завершились неудачами, то возбуждает исключение `Http404`;
- `get_previous_day(self, date)` — метод, возвращает значение даты, которая предшествует дате из параметра `date`. В зависимости от значений атрибутов класса `allow_empty` и `allow_future` может возбуждать исключение `Http404`;
- `get_next_day(self, date)` — метод, возвращает значение даты, которая следует за датой из параметра `date`. В зависимости от значений атрибутов класса `allow_empty` и `allow_future` может возбуждать исключение `Http404`.

10.6.5.2. Контроллер *DayArchiveView*: вывод записей за день

Контроллер-класс `DayArchiveView` наследует классы `BaseDateListView`, `DayMixin`, `MonthMixin`, `YearMixin` и `MultipleObjectTemplateResponseMixin`. Он выводит хронологический список записей, относящихся к указанному числу заданных месяца и года.

Набор записей, относящихся к заданному дню, будет храниться в переменной контекста шаблона, имеющей имя по умолчанию: `object_list`.

Дополнительные переменные, создаваемые в контексте шаблона:

- `day` — объект типа `date`, представляющий заданный день;
- `previous_day` — объект типа `date`, представляющий предыдущий день;
- `next_day` — объект типа `date`, представляющий следующий день.

К автоматически сгенерированному пути к шаблону по умолчанию добавляется суффикс `_archive_day`.

Пример использования контроллера `DayArchiveView`:

```
urlpatterns = [
    # Пример интернет-адреса: /2019/12/09/
    path('<int:year>/<int:month>/<int:day>/',
         DayArchiveView.as_view(model=Bb, date_field="published",
                                month_format='%m')),
]
```

10.6.6. Контроллер *TodayArchiveView*: вывод записей за текущее число

Контроллер-класс `TodayArchiveView` наследует классы `DayArchiveView` и `MultipleObjectTemplateResponseMixin`. Он выводит хронологический список записей, относящихся к текущему числу.

Набор записей, относящихся к текущему числу, будет храниться в переменной контекста шаблона, имеющей имя по умолчанию: `object_list`.

В контексте шаблона будут созданы дополнительные переменные:

- `day` — объект типа `date`, представляющий текущее число;
- `previous_day` — объект типа `date`, представляющий предыдущий день;
- `next_day` — объект типа `date`, представляющий следующий день.

К автоматически сгенерированному пути к шаблону по умолчанию добавляется суффикс `_archive_today`.

10.6.7. Контроллер *DateDetailView*: вывод одной записи за указанное число

Контроллер-класс `DateDetailView` наследует классы `DetailView`, `DateMixin`, `DayMixin`, `MonthMixin`, `YearMixin` и `SingleObjectTemplateResponseMixin`. Он выводит одну-единственную запись, относящуюся к текущему числу, и может быть полезен в случаях, когда поле даты, по которому выполняется поиск записи, хранит уникальные значения.

Запись, относящаяся к текущему числу, будет храниться в переменной контекста шаблона, имеющей имя по умолчанию: `object`.

К автоматически сгенерированному пути к шаблону по умолчанию добавляется суффикс `_detail`.

Листинг 10.10 представляет код контроллера `BbDetailView`, основанного на классе `DateDetailView`.

Листинг 10.10. Использование контроллера-класса `DateDetailView`

```
from django.views.generic.dates import DateDetailView
from .models import Bb, Rubric

class BbDetailView(DateDetailView):
    model = Bb
    date_field = 'published'
    month_format = '%m'

    def get_context_data(self, *args, **kwargs):
        context = super().get_context_data(*args, **kwargs)
        context['rubrics'] = Rubric.objects.all()
        return context
```

Для его использования в список маршрутов нужно добавить маршрут такого вида:

```
path('detail/<int:year>/<int:month>/<int:day>/<int:pk>/',
                        BbDetailView.as_view(), name='detail'),
```

К сожалению, в маршрут, указывающий на контроллер `DateDetailView` или его подкласс, следует записать URL-параметр ключа (`pk`) или слага (`slug`). Это связано с тем, что упомянутый ранее класс является производным от примеси `SingleObjectMixin`, которая требует для нормальной работы один из этих URL-параметров. Такая особенность сильно ограничивает область применения контроллера-класса `DateDetailView`.

10.7. Контроллер *RedirectView*: перенаправление

Контроллер-класс `RedirectView`, производный от класса `View`, выполняет перенаправление по указанному интернет-адресу. Он объявлен в модуле `django.views.generic.base`.

Этот класс поддерживает такие атрибуты и методы:

- `url` — атрибут, задает строку с интернет-адресом, на который следует выполнить перенаправление.

  Этот адрес может включать спецификаторы, поддерживаемые оператором `%` языка Python (за подробностями — на страницу **https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting**). В таких спецификаторах следует указывать имена URL-параметров — в этом случае в результирующий интернет-адрес будут подставлены их значения;

- `pattern_name` — атрибут, задает имя именованного маршрута. Обратное разрешение интернет-адреса будет проведено с теми же URL-параметрами, которые получил контроллер;
- `query_string` — атрибут. Если `True`, то все GET-параметры, присутствующие в текущем интернет-адресе, будут добавлены к интернет-адресу, на который выполняется перенаправление. Если `False`, то GET-параметры передаваться не будут (поведение по умолчанию);
- `get_redirect_url(self, *args, **kwargs)` — метод, должен возвращать строку с интернет-адресом, на который следует выполнить перенаправление.

  В параметре `kwargs` передается словарь со значениями именованных URL-параметров. Параметр `args` в старых версиях Django хранил список со значениями неименованных URL-параметров, но в настоящее время не используется.

  В реализации по умолчанию сначала извлекает значение атрибута `url` и выполняет форматирование, передавая значения полученных URL-параметров оператору `%`. Если атрибут `url` хранит значение `None`, то проводит обратное разрешение на основе значения атрибута `pattern_name` и, опять же, значений URL-параметров. Если и эта попытка увенчалась неудачей, то отправляет ответ с кодом 410 (запрошенная страница более не существует);
- `permanent` — атрибут. Если `True`, то будет выполнено постоянное перенаправление (с кодом статуса 301). Если `False`, то будет выполнено временное перенаправление (с кодом статуса 302). Значение по умолчанию: `False`.

В качестве примера организуем перенаправление с интернет-адресов вида **/detail/*<год>*/*<месяц>*/*<число>*/*<ключ>*/** по адресу **/detail/*<ключ>*/**. Для этого добавим в список маршруты:

```
path('detail/<int:pk>/', BbDetailView.as_view(), name='detail'),
path('detail/<int:year>/<int:month>/<int:day>/<int:pk>/',
                        BbRedirectView.as_view(), name='old_detail'),
```

Код контроллера `BbRedirectView`, который мы используем для этого, очень прост и приведен в листинге 10.11.

Листинг 10.11. Применение контроллера-класса `RedirectView`

```
class BbRedirectView(RedirectView):
    url = '/detail/%(pk)d/'
```

Для формирования целевого пути использован атрибут `url` и строка со спецификатором, обрабатываемым оператором `%`. Вместо этого спецификатора в строку будет подставлено значение URL-параметра `pk`, т. е. ключ записи.

10.8. Контроллеры-классы смешанной функциональности

Бóльшая часть функциональности контроллеров-классов наследуется ими от классов-примесей. Наследуя классы от нужных примесей, можно создавать контроллеры смешанной функциональности.

Так, мы можем объявить класс, производный от классов `SingleObjectMixin` и `ListView`. В результате получится контроллер, одновременно выводящий сведения о выбранной записи (функциональность, унаследованная от `SingleObjectMixin`) и набор связанных с ней записей (функциональность класса `ListView`).

В листинге 10.12 приведен код класса `BbByRubricView`, созданного на подобном принципе и имеющего смешанную функциональность.

Листинг 10.12. Пример контроллера-класса смешанной функциональности

```
from django.views.generic.detail import SingleObjectMixin
from django.views.generic.list import ListView
from .models import Bb, Rubric

class BbByRubricView(SingleObjectMixin, ListView):
    template_name = 'bboard/by_rubric.html'
    pk_url_kwarg = 'rubric_id'

    def get(self, request, *args, **kwargs):
        self.object = self.get_object(queryset=Rubric.objects.all())
        return super().get(request, *args, **kwargs)

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['current_rubric'] = self.object
        context['rubrics'] = Rubric.objects.all()
        context['bbs'] = context['object_list']
        return context

    def get_queryset(self):
        return self.object.bb_set.all()
```

Намереваясь использовать уже существующие шаблон и маршрут, мы указали в атрибутах класса путь к нашему шаблону и имя URL-параметра, через который передается ключ рубрики.

В переопределенном методе `get()` вызовом метода `get_object()`, унаследованного от примеси `SingleObjectMixin`, извлекаем рубрику с заданным ключом. Эту рубрику сохраняем в атрибуте `object` класса — она нам еще понадобится.

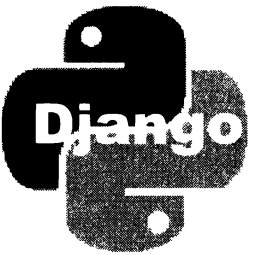
В переопределенном методе `get_context_data()` заносим в переменную `current_rubric` контекста шаблона найденную рубрику (взяв ее из атрибута `object`), а в переменную `rubrics` — набор всех рубрик.

Здесь мы столкнемся с проблемой. Наш шаблон за перечнем объявлений обращается к переменной `bbs` контекста шаблона. Ранее, разрабатывая предыдущую редакцию контроллера (см. листинг 10.4), мы занесли имя этой переменной в атрибут класса `contect_object_name`. Но здесь такой "номер" не пройдет: указав новое имя для переменной в атрибуте класса `contect_object_name`, мы зададим новое имя для переменной, в которой будет храниться рубрика, а не перечень объявлений (это связано с особенностями механизма множественного наследования Python). В результате чего получим чрезвычайно трудно диагностируемую ошибку.

Поэтому мы поступили по-другому: в том же переопределенном методе `get_context_data()` создали переменную `bbs` контекста шаблона и присвоили ей значение переменной `object_list`, в которой по умолчанию хранится набор записей, выводимых контроллером `ListView`.

И наконец, в переопределенном методе `get_queryset()` возвращаем перечень объявлений, связанных с найденной рубрикой и полученных через диспетчер обратной связи.

Вообще, на взгляд автора, лучше избегать контроллеров смешанной функциональности. Удобнее взять за основу контроллер-класс более низкого уровня и реализовать в нем всю нужную логику самостоятельно.



ГЛАВА 11

Шаблоны и статические файлы: базовые инструменты

Шаблон — это образец для генерирования веб-страницы, отправляемой клиенту в ответ на его запрос. Django также использует шаблоны для формирования электронных писем.

Рендеринг — собственно генерирование веб-страницы (или электронного письма) на основе заданного шаблона и контекста шаблона, содержащего все необходимые данные. *Шаблонизатор* — подсистема фреймворка, выполняющая рендеринг.

11.1. Настройки проекта, касающиеся шаблонов

Все настройки проекта, касающиеся шаблонов и шаблонизатора, записываются в параметре `TEMPLATES` модуля settings.py из пакета конфигурации. Параметру присваивается массив, каждый элемент которого является словарем, задающим параметры одного из шаблонизаторов, доступных во фреймворке.

> ***Внимание!***
>
> Практически всегда в Django-сайтах используется только один шаблонизатор. Применение двух и более шаблонизаторов — очень специфическая ситуация, не рассматриваемая в этой книге.

В каждом таком словаре можно задать следующие элементы:

- `BACKEND` — путь к модулю шаблонизатора, записанный в виде строки.

  В составе Django поставляются два шаблонизатора:

  - `django.template.backends.django.DjangoTemplates` — стандартный шаблонизатор, применяемый в большинстве случаев;
  - `django.template.backends.jinja2.Jinja2` — шаблонизатор Jinja2;

- `NAME` — псевдоним для шаблонизатора. Если не указан, то для обращения к шаблонизатору используется последняя часть пути к его модулю;

- DIRS — список путей к папкам, в которых шаблонизатор будет искать шаблоны (по умолчанию — "пустой" список);
- APP_DIRS — если True, то шаблонизатор дополнительно будет искать шаблоны в папках templates, располагающихся в пакетах приложений. Если False, то шаблонизатор станет искать шаблоны исключительно в папках из списка DIRS. Значение по умолчанию — False, однако во вновь созданном проекте устанавливается в True;
- OPTIONS — дополнительные параметры, поддерживаемые конкретным шаблонизатором. Также указываются в виде словаря, элементы которого задают отдельные параметры.

  Стандартный шаблонизатор django.template.backends.django.DjangoTemplates поддерживает такие параметры:

  - autoescape — если True, то все недопустимые знаки HTML (двойная кавычка, знаки "меньше" и "больше") при их выводе будут преобразованы в соответствующие специальные символы (поведение по умолчанию). Если False, то такое преобразование выполняться не будет;
  - string_if_invalid — строка, выводящаяся на экран в случае, если попытка доступа к переменной контекста шаблона или вычисления выражения потерпела неудачу (значение по умолчанию — "пустая" строка);
  - file_charset — обозначение кодировки, в которой записан код шаблонов, в виде строки (по умолчанию: "utf-8");
  - context_processors — список имен модулей, реализующих обработчики контекста, которые должны использоваться совместно с заданным шаблонизатором. Имена модулей должны быть заданы в виде строк.

    Обработчик контекста — это программный модуль, добавляющий в контекст шаблона какие-либо дополнительные переменные уже после его формирования контроллером. Список всех доступных в Django обработчиков контекста будет приведен позже;
  - debug — если True, то будут выводиться развернутые сообщения об ошибках в коде шаблона, если False — совсем короткие сообщения. Если параметр не указан, то будет использовано значение параметра проекта DEBUG (см. *разд. 3.3.1*);
  - loaders — список имен модулей, выполняющих загрузку шаблонов.

    Django — в зависимости от значений параметров DIRS и APP_DIRS — сам выбирает, какие загрузчики шаблонов использовать, и явно указывать их список не требуется. На всякий случай, перечень доступных во фреймворке загрузчиков шаблонов и их описания приведены на странице **https://docs.djangoproject.com/en/3.0/ref/templates/api/#template-loaders**;
  - builtins — список строк с путями к встраиваемым библиотекам тегов, которые должны использоваться с текущим шаблонизатором. Значение по умолчанию — "пустой" список.

Библиотека тегов — это программный модуль Python, расширяющий набор доступных тегов шаблонизатора. *Встраиваемая* библиотека тегов загружается в память непосредственно при запуске проекта, и объявленные в ней дополнительные теги доступны к использованию в любой момент времени без каких бы то ни было дополнительных действий;

- `libraries` — перечень загружаемых библиотек шаблонов. Записывается в виде словаря, ключами элементов которого станут псевдонимы библиотек тегов, а значениями элементов — строковые пути к модулям, реализующим эти библиотеки. Значение по умолчанию — "пустой" словарь.

  В отличие от встраиваемой библиотеки тегов, *загружаемая* библиотека перед использованием должна быть явно загружена с помощью тега шаблонизатора `load`.

  Параметры `builtins` и `libraries` служат для указания исключительно сторонних библиотек тегов, поставляемых отдельно от Django. Библиотеки тегов, входящие в состав фреймворка, записывать туда не нужно.

Теперь рассмотрим обработчики контекста, доступные в Django:

- `django.template.context_processors.request` — добавляет в контекст шаблона переменную `request`, хранящую объект текущего запроса (в виде экземпляра класса `Request`);
- `django.template.context_processors.csrf` — добавляет в контекст шаблона переменную `csrf_token`, хранящую электронный жетон, который используется тегом шаблонизатора `csrf_token`;
- `django.contrib.auth.context_processors.auth` — добавляет в контекст шаблона переменные `user` и `perms`, хранящие соответственно сведения о текущем пользователе и его правах;
- `django.template.context_processors.static` — добавляет в контекст шаблона переменную `STATIC_URL`, хранящую значение одноименного параметра проекта (мы рассмотрим его в конце этой главы);
- `django.template.context_processors.media` — добавляет в контекст шаблона переменную `MEDIA_URL`, хранящую значение одноименного параметра проекта (мы рассмотрим его в *главе 20*);
- `django.contrib.messages.context_processors.messages` — добавляет в контекст шаблона переменные `messages` и `DEFAULT_MESSAGE_LEVELS`, хранящие соответственно список всплывающих сообщений и словарь, сопоставляющий строковые обозначения уровней сообщений с их числовыми кодами (о работе со всплывающими сообщениями будет рассказано в *главе 23*);
- `django.template.context_processors.tz` — добавляет в контекст шаблона переменную `TIME_ZONE`, хранящую наименование текущей временной зоны;
- `django.template.context_processors.debug` — добавляет в контекст шаблона переменные:

- `debug` — хранит значение параметра проекта `DEBUG` (см. *разд. 3.3.1*);
- `sql_queries` — хранит сведения о запросах к базе данных. Представляют собой список словарей, каждый из которых представляет один запрос. Элемент `sql` такого словаря хранит SQL-код запроса, а элемент `time` — время его выполнения.

Может пригодиться при отладке сайта.

Листинг 11.1 показывает код, задающий настройки шаблонов по умолчанию, которые формируются при создании нового проекта.

Листинг 11.1. Настройки шаблонов по умолчанию

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

11.2. Вывод данных. Директивы

Для вывода данных в коде шаблона применяются *директивы*. Директива записывается в формате `{{ <источник значения> }}` и размещается в том месте шаблона, в которое нужно поместить значение из указанного *источника*. В его качестве можно задать:

- переменную из контекста шаблона. Пример вставки значения из переменной `rubric`:

  ```
  {{ rubric }}
  ```

- элемент последовательности, применив синтаксис:

  <переменная с последовательностью>.<индекс элемента>

  Вывод первой (с индексом 0) рубрики из списка `rubrics`:

  ```
  {{ rubrics.0 }}
  ```

- элемент словаря, применив синтаксис:

  <переменная со словарем>.<ключ элемента>

Вывод элемента `kind` словаря `current_bb`:

```
{{ current_bb.kind }}
```

- атрибут класса или экземпляра, применив привычную запись "с точкой". Вывод названия рубрики (атрибут `name`) из переменной `current_rubric`:

```
{{ current_rubric.name }}
```

- результат, возвращенный методом. Применяется запись "с точкой", круглые скобки не ставятся. Пример вызова метода `get_absolute_url()` у рубрики `rubric`:

```
{{ rubric.get_absolute_url }}
```

> ***Внимание!***
> Шаблонизатор Django не позволяет указать параметры у вызываемого метода. Поэтому в шаблонах можно вызывать только методы, не принимающие параметров или принимающие только необязательные параметры.

- обычные константы. Они записываются в том же виде, что и в Python-коде: строки должны быть взяты в одинарные или двойные кавычки, а числа с плавающей точкой должны включать дробную часть.

> ***Внимание!***
> Использовать в директивах выражения не допускается.

11.3. Теги шаблонизатора

Теги шаблонизатора управляют генерированием содержимого страницы. Они заключаются в последовательности символов `{%` и `%}`. Как и HTML-теги, теги шаблонизатора бывают одинарными и парными.

Одинарный тег, как правило, выводит на страницу какое-либо значение, вычисляемое самим фреймворком. Пример одинарного тега `csrf_token`, выводящего электронный жетон, который используется подсистемой безопасности фреймворка:

```
{% csrf_token %}
```

Парный тег "охватывает" фрагмент кода и выполняет над ним какие-либо действия. Он фактически состоит из двух тегов: *открывающего*, помечающего начало "охватываемого" фрагмента (*содержимого*), и *закрывающего*, который помечает его конец. Закрывающий тег имеет то же имя, что и открывающий, но с добавленным впереди префиксом `end`.

Например, парный тег `for . . . endfor` повторяет содержащийся в нем фрагмент столько раз, сколько элементов находится в указанной в этом теге последовательности. Тег `for` — открывающий, а тег `endfor` — закрывающий:

```
{% for bb in bbs %}
<div>
    <h2>{{ bb.title }}</h2>
    <p>{{ bb.content }}</p>
```

```
    <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
</div>
{% endfor %}
```

Теги, поддерживаемые шаблонизатором Django:

- `url` — формирует интернет-адрес путем обратного разрешения:

```
{% url <имя маршрута> <список значений параметров, разделенных ↵
пробелами> [as <переменная>] %}
```

В *имени маршрута* при необходимости можно указать пространство имен. Параметры в *списке* могут быть как позиционными, так и именованными. Примеры:

```
<a href="{% url 'bboard:detail' bb.pk %}">{{ bb.title }}</a>
<a href="{% url 'bboard:by_rubric' rubric_id=bb.rubric.pk %}">
{{ bb.rubric.name }}</a>
```

По умолчанию тег вставляет сформированный адрес в шаблон. Но можно указать шаблонизатору сохранить адрес в *переменной*, записав ее после ключевого слова `as`. Пример:

```
{% url 'bboard:detail' bb.pk as detail_url %}
<a href="{{ detail_url }}">{{ bb.title }}</a>
```

- `for . . . endfor` — перебирает в цикле элементы указанной *последовательности* (или *словаря*) и для каждого элемента создает в коде страницы копию своего *содержимого*. Аналог цикла `for . . . in` языка Python. Формат записи тега:

```
{% for <переменные> in <последовательность или словарь> %}
    <содержимое тега>
[{% empty %}
    <содержимое, выводящееся, если последовательность или словарь ↵
    не имеет элементов>]
{% endfor %}
```

Значение очередного элемента *последовательности* заносится в указанную *переменную*, если же перебирается *словарь*, то можно указать две *переменные* — под ключ и значение элемента соответственно. Эти *переменные* можно использовать в *содержимом* тега.

Также в *содержимом* присутствует следующий набор переменных, создаваемых самим тегом:

- `forloop.counter` — номер текущей итерации цикла (нумерация начинается с 1);
- `forloop.counter0` — номер текущей итерации цикла (нумерация начинается с 0);
- `forloop.revcounter` — число оставшихся итераций цикла (нумерация начинается с 1);
- `forloop.revcounter0` — число оставшихся итераций цикла (нумерация начинается с 0);

- `forloop.first` — `True`, если это первая итерация цикла, `False`, если не первая;
- `forloop.last` — `True`, если это последняя итерация цикла, `False`, если не последняя;
- `forloop.parentloop` — применяется во вложенном цикле и хранит ссылку на "внешний" цикл. Пример использования: `forloop.parentloop.counter` (получение номера текущей итерации "внешнего" цикла).

Пример:

```
{% for bb in bbs %}
<div>
  <p>№№ {{ forloop.counter }}</p>
  . . .
</div>
{% endfor %}
```

❒ `if . . . elif . . . else . . . endif` — аналог условного выражения Python:

```
{% if <условие 1> %}
    <содержимое 1>
[{% elif <условие 2> %}
    <содержимое 2>
. . .
{% elif <условие n> %}
    <содержимое n>]
[{% else %}
    <содержимое else>]
{% endif %}
```

В *условиях* можно указывать операторы сравнения `==`, `!=`, `<`, `>`, `<=`, `>=`, `in`, `not in`, `is` и `is not`, логические операторы `and`, `or` и `not`.

Пример:

```
{% if bbs %}
<h2>Список объявлений</h2>
<% else %}
<p>Объявлений нет</p>
{% endif %}
```

❒ `ifchanged . . . endifchanged` — применяется в циклах. Форматы использования:

```
{% ifchanged %} <содержимое> {% endifchanged %}

{% ifchanged <список значений, разделенных пробелами> %}
    <содержимое>
{% endifchanged %}
```

Первый формат выводит *содержимое*, если оно изменилось после предыдущей итерации цикла. Второй формат выводит *содержимое*, если изменилось одно из значений, приведенных в *списке*.

Выводим название рубрики, в которую вложена текущая рубрика, только если это название изменилось (т. е. если текущая рубрика вложена в другую рубрику, нежели предыдущая):

```
{% for rubric in rubrics %}
{% ifchanged %}{{ rubric.parent.name }}{% endifchanged %}
. . .
{% endfor %}
```

То же самое, только с использованием второго формата записи тега:

```
{% for rubric in rubrics %}
{% ifchanged rubric.parent %}
{{ rubric.parent.name }}
{% endifchanged %}
. . .
{% endfor %}
```

- `cycle` — последовательно помещает в шаблон очередное значение из указанного *перечня*:

  `cycle <перечень значений, разделенных пробелами> [as <переменная>]`

  По достижении конца *перечня* перебор начинается с начала. Количество значений в *перечне* не ограничено.

  В следующем примере при каждом проходе цикла `for . . . in` к блоку будут последовательно привязываться стилевые классы `b1`, `b2`, `b3`, потом снова `b1`, `b2` и т. д.:

```
{% for bb in bbs %}
<div class="{% cycle 'bb1' 'bb2' 'bb3' %}">
    <h2>{{ bb.title }}</h2>
    <p>{{ bb.content }}</p>
    <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
</div>
{% endfor %}
```

  Текущее значение *перечня* может быть занесено в *переменную*, указанную после ключевого слова `as`, и вставлено в другом месте страницы:

```
{% for bb in bbs %}
{% cycle 'bb1' 'bb2' 'bb3' as currentclass %}
<div>
    <h2 class="{{ currentclass }}">{{ bb.title }}</h2>
    <p>{{ bb.content }}</p>
    <p class="{{ currentclass }}">{{ bb.published|date:"d.m.Y H:i:s" }}</p>
</div>
{% endfor %}
```

- `{% resetcycle [<переменная>] %}` — сбрасывает тег `cycle`, после чего тот начинает перебирать указанные в нем значения с начала. По умолчанию сбрасывает последний тег `cycle`, записанный в шаблоне. Если же нужно сбросить конкрет-

ный тег, то следует указать *переменную*, записанную в нужном теге `cycle` после ключевого слова `as`;

- `{% firstof` *<перечень значений, разделенных пробелами>* `%}` — помещает в шаблон первое из приведенных в *перечне* значений, не равное `False` (т. е. не "пустое").

  Следующий пример помещает на страницу либо значение поля `phone` объявления `bb`; либо, если оно "пусто", значение поля `email`; либо, если и оно не заполнено, строку `"На деревню дедушке"`;

```
{% firstof bb.phone bb.email 'На деревню дедушке' %}
```

- `with . . . endwith` — заносит какие-либо значения в *переменные* и делает их доступными внутри своего *содержимого*:

```
{% with <набор операций присваивания, разделенных пробелами> %}
    <содержимое>
{% endwith %}
```

  Операции присваивания записываются так же, как и в Python.

  Может использоваться для временного сохранения в переменных результатов каких-либо вычислений (например, полученных при обращении к методу класса) — чтобы потом не выполнять эти вычисления повторно.

  Пример:

```
{% with bb_count=bbs.count %}
{% if bb_count > 0 %}
<p>Всего {{ bb_count }} объявлений.</p>
{% endwith %}
```

- `regroup` — выполняет группировку указанной *последовательности* словарей или объектов по значению элемента с заданным *ключом* или атрибута с заданным *именем* и помещает результат в *переменную*:

```
{% regroup <последовательность> by ↳
<ключ элемента или атрибут объекта> as <переменная> %}
```

  Сохраненный в *переменной* результат представляет собой список объектов типа `namedtuple` с элементами:

  - `grouper` — значение элемента или атрибута, по которому выполнялась группировка;
  - `list` — список словарей или объектов, относящихся к созданной группе.

  Пример группировки объявлений по рубрике:

```
{% regroup bbs by rubric.name as grouped_bbs %}
{% for rubric_name, gbbs in grouped_bbs %}
    <h3>{{ rubric_name }}</h3>
    {% for bb in gbbs %}
    <div>
        <h2>{{ bb.title }}</h2>
```

```
        <p>{{ bb.content }}</p>
        <p>{{ bb.published|date:"d.m.Y H:i:s" }}</p>
    </div>
    {% endfor %}
{% endfor %}
```

- `{% now` *<формат>* `%}` — выводит текущие дату и время, оформленные согласно заданному *формату* (форматирование значений даты и времени описано далее — при рассмотрении фильтра `date`):

```
{% now 'SHORT_DATETIME_FORMAT' %}
```

- `filter . . . endfilter` — применяет к *содержимому* указанные *фильтры*:

```
{% filter <фильтры> %}
    <содержимое>
{% endfilter %}
```

Применяем к абзацу фильтры `force_escape` и `upper`;

```
{% filter force_escape|upper %}
<p>Текст тега filter</p>
{% endfilter %}
```

- `csrf_token` — выводит электронный жетон, используемый подсистемой безопасности Django. Применяется исключительно в веб-формах;

- `autoescape on|off . . . endautoescape` — включает или отключает в *содержимом* автоматическое преобразование недопустимых знаков HTML (двойной кавычки, знаков "меньше" и "больше") в соответствующие специальные символы при их выводе:

```
{% autoescape on|off %}
    <содержимое>
{% endautoescape %}
```

Значение `on` включает автоматическое преобразование, значение `off` — отключает;

- `spaceless . . . endspaceless` — удаляет в *содержимом* пробельные символы (в число которых входят пробел, табуляция, возврат каретки и перевод строки) между тегами:

```
{% spaceless %}
    <содержимое>
{% endspaceless %}
```

Пример:

```
{% spaceless %}
<h3>
    <em>Последние объявления</em>
</h3>
{% endspaceless %}
```

В результате в код страницы будет помещен фрагмент:

```
<h3><em>Последние объявления</em></h3>
```

- `{% templatetag` *<обозначение последовательности символов>* `%}` — выводит последовательность символов, которую иначе вывести не получится (примеры: `{{, }}`, `{%, %}`). Поддерживаются следующие *обозначения последовательностей символов*:
  - `openblock: {%;`
  - `closeblock: %};`
  - `openvariable: {{;`
  - `closevariable: }};`
  - `openbrace: {;`
  - `closebrace: };`
  - `opencomment: {#;`
  - `closecomment: #};`
- `verbatim . . . endverbatin` — выводит *содержимое* как есть, не обрабатывая записанные в нем директивы, теги и фильтры шаблонизатора:

```
{% verbatim %}
    <содержимое>
{% endverbatim %}
```

Пример:

```
{% verbatim %}
<p>Текущие дата и время выводятся тегом {% now %}.</p>
{% endverbatim %}
```

- `{% load` *<псевдонимы библиотек тегов через пробел>* `%}` — загружает библиотеки тегов с указанными *псевдонимами*:

```
{% load static %}
```

- `widthratio` — применяется для создания диаграмм:

```
{% widthratio <текущее значение> <максимальное значение> ↵
<максимальная ширина> %}
```

Текущее значение делится на *максимальное значение*, после чего получившееся частное умножается на *максимальную ширину*, и результат всего этого вставляется в шаблон. Пример использования этого довольно странного тега:

```
<img src="bar.png"
style="height:10px;width:{% widthratio current_value 200 100 %}px">
```

- `comment . . . encomment` — создает в коде шаблона комментарий, не обрабатываемый шаблонизатором:

```
{% comment [<заголовок>] %}
    <содержание комментария>
{% encomment %}
```

У комментария можно задать необязательный *заголовок*. Пример:

```
{% comment 'Доделать завтра!' %}
<p>Здесь будет список объявлений</p>
{% endcomment %}
```

Если комментарий занимает всего одну строку или часть ее, то его можно создать, заключив в последовательности символов {# и #}:

```
{# Не забыть сделать вывод рубрик! #}
```

- {% debug %} — выводит разнообразную отладочную информацию, включая содержимое контекста шаблона и список отладочных модулей. Эта информация весьма объемна и не очень удобна на практике.

11.4. Фильтры

Фильтры шаблонизатора выполняют заданные преобразования значения перед его выводом.

Фильтр записывается непосредственно в директиве, после источника значения, и отделяется от него символом вертикальной черты (|). Пример:

```
{{ bb.published|date:"d.m.Y H:i:s" }}
```

Фильтры можно объединять, перечислив через вертикальную черту, в результате чего они будут обрабатываться последовательно, слева направо:

```
{{ bb.content|lower|default:'--описания нет--' }}
```

Вот список фильтров, поддерживаемых шаблонизатором Django:

- date:*<формат>* — форматирует значение даты и времени согласно заданному *формату*. В строке *формата* допустимы следующие специальные символы:
  - j — число без начального нуля;
  - d — число с начальным нулем;
  - m — номер месяца с начальным нулем;
  - n — номер месяца без начального нуля;
  - F и N — полное название месяца в именительном падеже с большой буквы;
  - E — полное название месяца в родительном падеже и в нижнем регистре;
  - M — сокращенное название месяца с большой буквы;
  - b — сокращенное название месяца в нижнем регистре;
  - Y и o — год из четырех цифр;
  - y — год из двух цифр;
  - L — True, если это високосный год, False, если обычный;
  - w — номер дня недели от 0 (воскресенье) до 6 (суббота);
  - l — сокращенное название дня недели с большой буквы;
  - D — полное название дня недели в именительном падеже с большой буквы;
  - G — часы в 24-часовом формате без начального нуля;

- `H` — часы в 24-часовом формате с начальным нулем;
- `g` — часы в 12-часовом формате без начального нуля;
- `h` — часы в 12-часовом формате с начальным нулем;
- `i` — минуты;
- `s` — секунды с начальным нулем;
- `u` — микросекунды;
- `a` — обозначение половины суток в нижнем регистре (`"д.п."` или `"п.п."`);
- `A` — обозначение половины суток в верхнем регистре (`"ДП"` или `"ПП"`);
- `I` — 1, если сейчас летнее время, 0, если зимнее;
- `P` — часы в 12-часовом формате и минуты. Если минуты равны 0, то они не указываются. Вместо 00:00 выводится строка `"полночь"`, а вместо 12:00 — `"полдень"`;
- `f` — часы в 12-часовом формате и минуты. Если минуты равны 0, то они не указываются;
- `t` — число дней в текущем месяце;
- `z` — порядковый номер дня в году;
- `W` — порядковый номер недели (неделя начинается с понедельника);
- `e` — название временной зоны;
- `O` — разница между текущим и гринвичским временем в часах;
- `Z` — разница между текущим и гринвичским временем в секундах;
- `c` — дата и время в формате ISO 8601;
- `r` — дата и время в формате RFC 5322;
- `U` — время в формате UNIX (выражается как количество секунд, прошедших с полуночи 1 января 1970 года);
- `T` — название временной зоны, установленной в настройках компьютера.

Пример:

```
{{ bb.published|date:'d.m.Y H:i:s' }}
```

Также можно использовать следующие встроенные в Django форматы:

- `DATE_FORMAT` — развернутый формат даты;
- `DATETIME_FORMAT` — развернутый формат даты и времени;
- `SHORT_DATE_FORMAT` — сокращенный формат даты;
- `SHORT_DATETIME_FORMAT` — сокращенный формат даты и времени.

Пример:

```
{{ bb.published|date:'DATETIME_FORMAT' }}
```

- time[:*<формат времени>*] — форматирует выводимое значение времени согласно заданному *формату*. При написании *формата* применяются те же специальные символы, что и в случае фильтра date. Пример:

```
{{ bb.published|time:'H:i' }}
```

Для вывода времени с применением формата по умолчанию следует использовать обозначение TIME_FORMAT:

```
{{ bb.published|time:'TIME_FORMAT' }}
```

или вообще не указывать формат:

```
{{ bb.published|time }}
```

- timesince[:*<значение для сравнения>*] — выводит промежуток времени, разделяющий выводимое значение даты и времени и заданное *значение для сравнения*, относящееся к будущему (если таковое не указано, в его качестве принимается сегодняшние дата и время). Результат выводится в виде, например, "3 недели, 6 дней", "6 дней 23 часа" и т. п. Если *значение для сравнения* относится к прошлому, выведет строку: "0 минут";

- timeuntil[:*<значение для сравнения>*] — то же самое, что и timesince, но *значение для сравнения* должно относиться к прошлому;

- yesno[:*<строка образцов>*] — преобразует значения True, False и, возможно, None в слова "да", "нет" и "может быть".

Можно указать свои слова для преобразования, записав их в *строке образцов* вида <*строка для* `True`>,<*строка для* `False`>[,<*строка для* `None`>]. Если *строка для* `None` не указана, то вместо нее будет выводиться *строка для* `False` (поскольку None будет неявно преобразовываться в False).

Примеры:

```
{{ True|yesno }}, {{ False|yesno }}, {{ None|yesno }}
{# Результат: да, нет, может быть #}

{{ True|yesno:'так точно,никак нет,дело темное' }},
{{ False|yesno:'так точно,никак нет,дело темное' }},
{{ None|yesno:'так точно,никак нет,дело темное' }}
{# Результат: так точно, никак нет, дело темное #}

{{ True|yesno:'да,нет' }}, {{ False|yesno:'да,нет' }},
{{ None|yesno:'да,нет' }}
{# Результат: да, нет, нет #}
```

- default:*<величина>* — если выводимое значение равно False, то возвращает указанную *величину*.

Следующий пример выведет строку "У товара нет цены", если поле price товара bb не заполнено или хранит 0:

```
{{ bb.price|default:'У товара нет цены' }}
```

- `default_if_none:<величина>` — то же самое, что и `default`, но возвращает *величину* только в том случае, если выводимое значение равно `None`;
- `upper` — переводит все буквы выводимого значения в верхний регистр;
- `lower` — переводит все буквы выводимого значения в нижний регистр;
- `capfirst` — переводит первую букву выводимого значения в верхний регистр;
- `title` — переводит первую букву каждого слова в выводимом значении в верхний регистр;
- `truncatechars:<длина>` — обрезает выводимое значение до указанной *длины*, помещая в конец символ многоточия (...);
- `truncatechars_html:<длина>` — то же самое, что и `truncatechars`, но сохраняет все HTML-теги, которые встретятся в выводимом значении;
- `truncatewords:<количество слов>` — обрезает выводимое значение, оставляя в нем указанное *количество слов*. В конце обрезанного значения помещается символ многоточия (...);
- `truncatewords_html:<количество слов>` — то же самое, что и `truncatewords`, но сохраняет все HTML-теги, которые встретятся в выводимом значении;
- `wordwrap:<величина>` — выполняет перенос выводимого строкового значения по словам таким образом, чтобы длина каждой получившейся в результате строки не превышала указанную *величину*;
- `cut:<удаляемая подстрока>` — удаляет из выводимого значения все вхождения заданной *подстроки*:

```
{{ 'Python'|cut:'t' }}          {# Результат: 'Pyhon' #}
{{ 'Python'|cut:'th' }}         {# Результат: 'Pyon' #}
```

- `slugify` — преобразует выводимое строковое значение в слаг;
- `stringformat:<формат>` — форматирует выводимое значение согласно указанному *формату*. При написании *формата* применяются специальные символы, поддерживаемые оператором `%` языка Python (см. страницу **https://docs.python.org/3/library/stdtypes.html#old-string-formatting**);
- `floatformat[:<количество знаков после запятой>]` — округляет выводимое вещественное число до заданного *количества знаков после запятой*. Целые числа автоматически приводятся к вещественному типу. Если указать положительное значение *количества знаков*, у преобразованных целых чисел дробная часть будет выводиться, если отрицательное — не будет. Значение *количество знаков* по умолчанию: `-1`. Примеры:

```
{{ 34.23234|floatformat }}          {# Результат: 34.2 #}
{{ 34.00000|floatformat }}          {# Результат: 34 #}
{{ 34.26000|floatformat }}          {# Результат: 34.3 #}
{{ 34.23234|floatformat:3 }}        {# Результат: 34.232 #}
{{ 34.00000|floatformat:3 }}        {# Результат: 34.000 #}
{{ 34.26000|floatformat:3 }}        {# Результат: 34.260 #}
```

```
{{ 34.23234|floatformat:-3 }}        {# Результат: 34.232 #}
{{ 34.00000|floatformat:-3 }}        {# Результат: 34 #}
{{ 34.26000|floatformat:-3 }}        {# Результат: 34.260 #}
```

- `filesizeformat` — выводит числовую величину как размер файла (примеры: `"100 байт"`, `"8,8 КБ"`, `"47,7 МБ"`);
- `add:<величина>` — прибавляет к выводимому значению указанную *величину*. Можно складывать числа, строки и последовательности;
- `divisibleby:<делитель>` — возвращает `True`, если выводимое значение делится на указанный *делитель* без остатка, и `False` — в противном случае;
- `wordcount` — возвращает число слов в выводимом строковом значении;
- `length` — возвращает число элементов в выводимой последовательности. Также работает со строками;
- `length_is:<величина>` — возвращает `True`, если длина выводимой последовательности равна указанной *величине*, и `False` — в противном случае;
- `first` — возвращает первый элемент выводимой последовательности;
- `last` — возвращает последний элемент выводимой последовательности;
- `random` — возвращает случайный элемент выводимой последовательности;
- `slice:<оператор взятия среза Python>` — возвращает срез выводимой последовательности. *Оператор взятия среза* записывается без квадратных скобок. Пример:

  ```
  {{ rubric_names|slice:'1:3' }}
  ```

- `join:<разделитель>` — возвращает строку, составленную из элементов выводимой последовательности, которые отделяются друг от друга *разделителем*;
- `make_list` — преобразует выводимую строку в список, содержащий символы этой строки;
- `dictsort:<ключ элемента>` — если выводимое значение представляет собой последовательность словарей или объектов, то сортирует ее по значениям элементов с указанным *ключом*. Сортировка выполняется по возрастанию значений.

  Пример вывода объявлений с сортировкой по цене:

  ```
  {% for bb in bbs|dictsort:'price' %}
      . . .
  {% endfor %}
  ```

  Можно сортировать последовательность списков или кортежей, только вместо ключа нужно указать индекс элемента вложенного списка (кортежа), по значениям которого следует выполнить сортировку. Пример:

  ```
  {% for el in list_of_lists|dictsort:1 %}
      . . .
  {% endfor %}
  ```

- `dictsortreversed:<ключ элемента>` — то же самое, что `dictsort`, только сортировка выполняется по убыванию значений;
- `unordered_list` — используется, если выводимым значением является список или кортеж, элементы которого представлены также списками или кортежами. Возвращает HTML-код, создающий набор вложенных друг в друга неупорядоченных списков, без "внешних" тегов `<ul>` и `</ul>`. Пример:

```
ulist = [
         'PHP',
         ['Python', 'Django'],
         ['JavaScript', 'Node.js', 'Express']
]
. . .
<ul>
    {{ ulist:unordered_list }}
</ul>
```

Выводимый результат:

```
<ul>
    <li>PHP
        <ul>
            <li>Python</li>
            <li>Django</li>
        </ul>
        <ul>
            <li>JavaScript</li>
            <li>None.js</li>
            <li>Express</li>
        </ul>
    </li>
</ul>
```

- `linebreaksbr` — заменяет в выводимом строковом значении все символы перевода строки на HTML-теги `<br>`;
- `linebreaks` — разбивает выводимое строковое значение на отдельные строки. Если в значении встретится одинарный символ перевода строки, то он будет заменен HTML-тегом `<br>`. Если встретится двойной символ перевода строки, то разделяемые им части значения будут заключены в теги `<p>`;
- `urlize` — преобразует все встретившиеся в выводимом значении интернет-адреса и адреса электронной почты в гиперссылки (теги `<a>`). В каждый тег, создающий обычную гиперссылку, добавляется атрибут `rel` со значением `nofollow`.

  Фильтр `urlize` нормально работает только с обычным текстом. При попытке обработать им HTML-код результат окажется непредсказуемым;
- `urlizetrunc:<длина>` — то же самое, что и `urlize`, но дополнительно обрезает текст гиперссылок до указанной *длины*, помещая в его конец символ многоточия (...);

- `safe` — подавляет у выводимого значения автоматическое преобразование недопустимых знаков HTML в соответствующие специальные символы;
- `safeseq` — подавляет у всех элементов выводимой последовательности автоматическое преобразование недопустимых знаков HTML в соответствующие специальные символы. Обычно применяется совместно с другими фильтрами. Пример:

```
{{ rubric_names|safeseq|join:", " }}
```

- `escape` — преобразует недопустимые знаки HTML в соответствующие специальные символы. Обычно применяется в содержимом парного тега `autoescape` с отключенным автоматическим преобразованием недопустимых знаков. Пример:

```
{% autoescape off %}
    {{ blog.content|escape }}
{% endautoescape %}
```

- `force_escape` — то же самое, что и `escape`, но выполняет преобразование принудительно. Может быть полезен, если требуется провести преобразование у результата, возвращенного другим фильтром;
- `escapejs` — преобразует выводимое значение таким образом, чтобы его можно было использовать как строковое значение JavaScript;
- `striptags` — удаляет из выводимого строкового значения все HTML-теги;
- `urlencode` — кодирует выводимое значение таким образом, чтобы его можно было включить в состав интернет-адреса (например, передать с GET-параметром);
- `iriencode` — кодирует выводимый интернационализированный идентификатор ресурса (IRI) таким образом, чтобы его можно было включить в состав интернет-адреса (например, передать с GET-параметром);
- `addslashes` — добавляет символы обратного слеша перед одинарными и двойными кавычками;
- `ljust:`*<ширина пространства в символах>* — помещает выводимое значение в левой части пространства указанной *ширины*:

```
{{ 'Python'|ljust:20 }}
```

  Результатом станет строка: "Python ";

- `center:`*<ширина пространства в символах>* — помещает выводимое значение в середине пространства указанной *ширины*:

```
{{ 'Python'|center:20 }}
```

  Результатом станет строка: " Python ";

- `rjust:`*<ширина пространства в символах>* — помещает выводимое значение в правой части пространства указанной *ширины*:

```
{{ 'Python'|rjust:20 }}
```

  Результатом станет строка: " Python";

- `get_digit:<позиция цифры>` — возвращает цифру, присутствующую в выводимом числовом значении по указанной *позиции*, отсчитываемой справа. Если текущее значение не является числом или если указанная *позиция* меньше 1 или больше общего количества цифр в числе, то возвращается значение *позиции*. Пример:

```
{{ 123456789|get_digit:4 }}          {# Результат: 6 #}
```

- `linenumbers` — выводит строковое значение, разбитое на отдельные строки посредством символов перевода строки, с номерами строк, поставленными слева.

11.5. Наследование шаблонов

Аналогично наследованию классов в Python, Django предлагает механизм наследования шаблонов. Базовый шаблон содержит элементы, присутствующие на всех страницах сайта: шапку, поддон, главную панель навигации, элементы разметки и др. А производный шаблон выводит лишь уникальное содержимое генерируемой им страницы: список объявлений, объявление, выбранное посетителем, и др.

Базовый шаблон содержит *блоки*, помечающие места, куда будут выведены фрагменты уникального содержимого, сгенерированного производным шаблоном. Блоки в базовом шаблоне объявляются с применением парного тега `block . . . endblock`:

```
{% block <имя блока> %}
    <содержимое по умолчанию>
{% endblock [<имя блока>] %}
```

Имя блока должно быть уникальным в пределах базового шаблона. Его также можно указать в теге `endblock`, чтобы в дальнейшем не гадать, какому тегу `block` он соответствует.

Содержимое по умолчанию будет выведено, если производный шаблон его не сгенерирует.

Пример:

```
<title>{% block title %}Главная{% endblock %} - Доска объявлений</title>
. . .
{% block content %}
    <p>Содержимое базового шаблона</p>
{% endblock content %}
```

В коде производного шаблона необходимо явно указать, что он является производным от определенного базового шаблона, вставив в начало его кода тег `{% extends <путь к базовому шаблону> %}`.

> ***Внимание!***
>
> Тег `extends` должен находиться в самом начале кода шаблона, на отдельной строке.

Пример (подразумевается, что базовый шаблон хранится в файле layout\basic.html):

```
{% extends "layout/basic.html" %}
```

После этого в производном шаблоне точно так же объявляются блоки, но теперь уже в них записывается создаваемое шаблоном содержимое:

```
{% block content %}
    <p>Содержимое производного шаблона</p>
{% endblock %}
```

Содержимое по умолчанию, заданное в базовом шаблоне, в соответствующем блоке производного шаблона доступно через переменную `block.super`, создаваемую в контексте шаблона самим Django:

```
{% block content %}
    <p>Содержимое производного шаблона 1</p>
    {{ block.super }}
    <p>Содержимое производного шаблона 2</p>
{% endblock %}
```

В результате на экран будут выведены три абзаца:

```
Содержимое производного шаблона 1
Содержимое базового шаблона
Содержимое производного шаблона 2
```

В производном шаблоне можно создать другие блоки и таким образом сделать его базовым для других производных шаблонов.

Конкретные примеры использования наследования шаблонов можно увидеть в листингах *разд. 2.7*.

11.6. Обработка статических файлов

В терминологии Django *статическими* называются файлы, отправляемые клиенту как есть: таблицы стилей, графические изображения, аудио- и видеоролики, файлы статических веб-страниц и т. п.

Обработку статических файлов выполняет подсистема, реализованная во встроенном приложении `django.contrib.staticfiles`. Оно включается в список зарегистрированных приложений (см. *разд. 3.3.3*) уже при создании нового проекта, и, если сайт содержит статические файлы, удалять его оттуда нельзя.

11.6.1. Настройка подсистемы статических файлов

Подсистемой статических файлов управляет ряд настроек, записываемых в модуле settings.py пакета конфигурации:

- `STATIC_URL` — префикс, добавляемый к интернет-пути статического файла. Встретив в начале полученного в запросе пути этот префикс, Django "поймет", что запрашивается статический файл. Значение по умолчанию — `None`, но при создании нового проекта оно устанавливается в `"/static/"`;
- `STATIC_ROOT` — файловый путь к основной папке, в которой хранятся все статические файлы (значение по умолчанию — `None`).

В этой же папке будут собираться все статические файлы, если отдать команду `collectstatic` утилиты manage.py;

- `STATICFILES_DIRS` — список файловых путей к дополнительным папкам, в которых хранятся статические файлы. Каждый путь может быть задан в двух форматах:
  - как строка с файловым путем. Пример:

```
STATICFILES_DIRS = [
    'c:/site/static',
    'c:/work/others/images',
]
```

  - как кортеж из двух элементов: префикса интернет-пути и файлового пути к папке. Чтобы сослаться на файл, хранящийся в определенной папке, нужно предварить интернет-путь этого файла заданным для папки префиксом. Пример:

```
STATICFILES_DIRS = [
    ('main', 'c:/site/static'),
    ('images', 'c:/work/imgs'),
]
```

  Теперь, чтобы вывести на страницу файл logo.png, хранящийся в папке c:\work\imgs\others\, следует записать в шаблоне тег:

```
<img src="{% static 'images/others/logo.png' %}">
```

- `STATICFILES_FINDERS` — список имен классов, реализующих подсистемы поиска статических файлов. По умолчанию включает два класса, объявленные в модуле `django.contrib.staticfiles.finders`:
  - `FileSystemFinder` — ищет статические файлы в папках, заданных параметрами `STATIC_ROOT` и `STATICFILES_DIRS`;
  - `AppDirectoriesFinder` — ищет статические файлы в папках static, находящихся в пакетах приложений.

  Если статические файлы хранятся в каком-то определенном местоположении (только в папках, заданных параметрами `STATIC_ROOT` и `STATICFILES_DIRS`, или только в папках static в пакетах приложений), можно указать в параметре `STATICFILES_FINDERS` только один класс — соответствующий случаю. Это несколько уменьшит потребление системных ресурсов;
- `STATICFILES_STORAGE` — имя класса, реализующего хранилище статических файлов. По умолчанию используется хранилище `StaticFilesStorage` из модуля `django.contrib.staticfiles.storage`.

11.6.2. Обслуживание статических файлов

Встроенный в Django отладочный веб-сервер обслуживает статические файлы самостоятельно. Но если сайт находится в эксплуатационном режиме, придется по-

заботиться об обслуживании статических файлов веб-сервером самостоятельно. Как это сделать, будет рассказано в *главе 30*.

11.6.3. Формирование интернет-адресов статических файлов

Формировать адреса статических файлов в коде шаблонов можно посредством трех разных программных механизмов:

- `static` — тег шаблонизатора, вставляющий в шаблон полностью сформированный интернет-адрес статического файла. Формат записи:

  ```
  {% static <относительный путь к статическому файлу> [as <переменная>] %}
  ```

  Относительный путь к статическому файлу записывается в виде строки и отсчитывается от папки, путь которой записан в параметрах `STATIC_ROOT` и `STATICFILES_DIRS`, или папки static пакета приложения.

  Тег реализован в библиотеке тегов с псевдонимом `static`, которую следует предварительно загрузить тегом `load`.

  Пример:

  ```
  {% load static %}
  . . .
  <link . . . href="{% static 'bboard/style.css' %}">
  ```

  По умолчанию сформированный адрес непосредственно вставляется в код страницы. Также можно сохранить адрес в *переменной*, записав ее после ключевого слова `as`. Пример:

  ```
  {% static 'bboard/style.css' as css_url %}
  <link . . . href="{{ css_url }}">
  ```

- `{% get_static_prefix %}` — тег шаблонизатора, который вставляет в код страницы префикс из параметра `STATIC_URL`. Также реализован в библиотеке тегов `static`. Пример:

  ```
  {% load static %}
  . . .
  <link . . . href="{% get_static_prefix %}bboard/style.css">
  ```

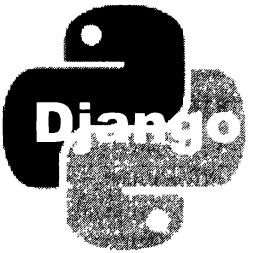
- `django.template.context_processors.static` — обработчик контекста, добавляющий в контекст шаблона переменную `STATIC_URL`, которая хранит префикс из одноименного параметра проекта. Поскольку этот обработчик по умолчанию не включен в список активных (элемент `context_processors` параметра `OPTIONS` — см. *разд. 11.1*), его нужно добавить туда:

  ```
  TEMPLATES = [
      {
          'BACKEND': 'django.template.backends.django.DjangoTemplates',
          . . .
  ```

```
        'OPTIONS': {
            'context_processors': [
                . . .
                'django.template.context_processors.static',
            ],
        },
    },
]
```

После этого можно обращаться к переменной `STATIC_URL`, созданной данным обработчиком в контексте шаблона:

```
<link . . . href="{{ STATIC_URL }}bboard/style.css">
```



ГЛАВА 12

Пагинатор

При выводе большие списки практически всегда разбивают на отдельные пронумерованные части, включающие не более определенного количества позиций. Это позволяет уменьшить размер страниц и ускорить их загрузку. Для перехода на нужную часть списка на страницах создают набор гиперссылок.

Разбиением списков на части занимается программный механизм, носящий название *пагинатора*.

> ***На заметку***
>
> Пагинатор явно применяется только в контроллерах-функциях (см. *главу 9*) и в контроллерах-классах самого низкого уровня (см. *главу 10*). Высокоуровневые контроллеры-классы, наподобие `ListView` (см. *разд. 10.4.3*), используют пагинатор неявно.

12.1. Класс *Paginator*: сам пагинатор. Создание пагинатора

Класс `Paginator` из модуля `django.core.paginator` представляет сам пагинатор. Экземпляр именно этого класса необходимо создать, чтобы реализовать пагинацию. Формат его конструктора:

```
Paginator(<набор записей>, <количество записей в части>[, orphans=0][,
                          allow_empty_first_page=True])
```

Первый параметр задает набор записей, который должен разбиваться на части, второй — количество записей в части.

Необязательный параметр `orphans` указывает минимальное количество записей, которые могут присутствовать в последней части пагинатора. Если последняя часть пагинатора содержит меньше записей, то все эти записи будут выведены в составе предыдущей части. Если задать значение 0, то в последней части может присутствовать сколько угодно записей (поведение по умолчанию).

Необязательный параметр `allow_empty_first_page` указывает, будет ли создаваться "пустая" часть, если набор не содержит записей. Значение `True` разрешает это де-

лать (поведение по умолчанию), значение `False`, напротив, предписывает возбудить в таком случае исключение `EmptyPage` из модуля `django.core.paginator`.

Класс `Paginator` поддерживает три атрибута:

- `count` — общее количество записей во всех частях пагинатора;
- `num_pages` — количество частей, на которые разбит набор записей;
- `page_range` — итератор, последовательно возвращающий номера всех частей пагинатора, начиная с 1.

Однако для нас полезнее будут два метода этого класса:

- `get_page(<номер части>)` — возвращает экземпляр класса `Page` (он будет описан далее), представляющий часть с указанным *номером*. Нумерация частей начинается с 1.

  Если *номер части* не является целочисленной величиной, то возвращается первая страница. Если *номер части* является отрицательным числом или превышает общее количество частей в пагинаторе, то возвращается последняя часть. Если получаемая часть "пуста", а при создании экземпляра класса `Paginator` параметру `allow_empty_first_page` было дано значение `False`, возбуждается исключение `EmptyPage`;

- `page(<номер части>)` — то же самое, что и `get_page()`, но в любом случае, если *номер части* не целое число, либо отрицательное число, либо оно превышает общее количество частей в пагинаторе, то возбуждается исключение `PageNotAnInteger` из модуля `django.core.paginator`.

  Этот метод оставлен для совместимости с предыдущими версиями Django.

Листинг 12.1 показывает пример использования пагинатора. В нем приведен код контроллера-функции `index()`, которая выводит список объявлений с разбиением на части (подразумевается, что номер части передается через GET-параметр `page`).

Листинг 12.1. Пример использования пагинатора

```
from django.shortcuts import render
from django.core.paginator import Paginator
from .models import Bb, Rubric

def index(request):
    rubrics = Rubric.objects.all()
    bbs = Bb.objects.all()
    paginator = Paginator(bbs, 2)
    if 'page' in request.GET:
        page_num = request.GET['page']
    else:
        page_num = 1
    page = paginator.get_page(page_num)
    context = {'rubrics': rubrics, 'page': page, 'bbs': page.object_list}
    return render(request, 'bboard/index.html', context)
```

Мы проверяем, присутствует ли в наборе GET-параметров, полученных в запросе, параметр `page`. Если это так, извлекаем из него номер части, которую нужно вывести на странице. В противном случае подразумевается, что посетитель запрашивает первую часть, которую мы и выводим.

В контексте шаблона создаем переменную `bbs`, которой присваиваем список записей, входящих в запрошенную часть (его можно извлечь из атрибута `object_list` части пагинатора). Это позволит использовать уже имеющийся шаблон bboard\index.html.

12.2. Класс *Page*: часть пагинатора. Вывод пагинатора

Класс `Page` из модуля `django.core.paginator` представляет отдельную часть пагинатора, возвращенную методом `get_page()` или `page()` (см. *разд. 12.1*).

Класс `Page` поддерживает следующие атрибуты:

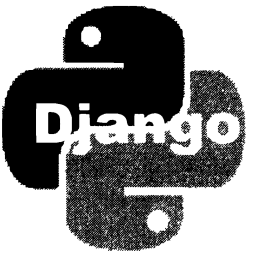
- `object_list` — список записей, входящих в состав текущей части;
- `number` — порядковый номер текущей части пагинатора (нумерация частей начинается с 1);
- `paginator` — пагинатор (в виде экземпляра класса `Paginator`), создавший эту часть.

А вот набор методов этого класса:

- `has_next()` — возвращает `True`, если существуют следующие части пагинатора, и `False` — в противном случае;
- `has_previous()` — возвращает `True`, если существуют предыдущие части пагинатора, и `False` — в противном случае;
- `has_other_pages()` — возвращает `True`, если существуют предыдущие или следующие части пагинатора, и `False` — в противном случае;
- `next_page_number()` — возвращает номер следующей части пагинатора. Если это последняя часть (т. е. следующей части не существует), то возбуждает исключение `EmptyPage`;
- `previous_page_number()` — возвращает номер предыдущей части пагинатора. Если это первая часть (т. е. предыдущей части не существует), то возбуждает исключение `EmptyPage`;
- `start_index()` — возвращает порядковый номер первой записи, присутствующей в текущей части. Нумерация записей в этом случае начинается с 1;
- `end_index()` — возвращает порядковый номер последней записи, присутствующей в текущей части. Нумерация записей в этом случае начинается с 1.

Далее приведен фрагмент кода шаблона bboard\index.html, выводящий набор гиперссылок для перехода между частями пагинатора:

```
<div>
    {% if page.has_previous %}
    <a href="?page={{ page.previous_page_number }}">&lt;</a>
      |  
    {% endif %}
    Часть №{{ page.number }} из {{ page.paginator.num_pages }}
    {% if page.has_next %}
      |  
    <a href="?page={{ page.next_page_number }}">&gt;</a>
    {% endif %}
</div>
```



ГЛАВА 13

Формы, связанные с моделями

Форма в терминологии Django — это объект, выводящий на страницу веб-форму для занесения данных и проверяющий введенные данные на корректность. Форма определяет набор полей, в которые будут вводиться отдельные значения, типы заносимых в них значений, элементы управления, посредством которых будет осуществляться ввод данных, и правила валидации.

Форма, связанная с моделью, отличается от обычной формы тем, что представляет какую-либо запись модели — хранящуюся в базе данных или еще не существующую. В частности, поля такой формы соответствуют одноименным полям модели. Помимо этого, такая форма поддерживает метод `save()`, сохраняющий занесенные в форму данные в базе.

13.1. Создание форм, связанных с моделями

Существуют три способа создать форму, связанную с моделью: два простых и сложный.

13.1.1. Создание форм с помощью фабрики классов

Первый, самый простой способ создать форму, связанную с моделью, — использовать функцию `modelform_factory()` из модуля `django.forms`. Вот формат ее вызова:

```
modelform_factory(<модель>[, fields=None][, exclude=None][,
                  labels=None][, help_texts=None][,
                  error_messages=None][, field_classes=None][,
                  widgets=None][,
                  form=<базовая форма, связанная с моделью>])
```

В первом параметре указывается ссылка на класс модели, на основе которой нужно создать форму.

Параметр `fields` задает последовательность имен полей модели, которые должны быть включены в создаваемую форму. Любые поля, не включенные в эту последо-

вательность, не войдут в состав формы. Чтобы указать все поля модели, нужно присвоить этому параметру строку "__all__".

Параметр exclude задает последовательность имен полей модели, которые, напротив, не должны включаться в форму. Соответственно, все поля, отсутствующие в этой последовательности, войдут в состав формы.

> ***Внимание!***
>
> **В вызове функции modelform_factory() должен присутствовать либо параметр fields, либо параметр exclude. Указание сразу обоих параметров приведет к ошибке.**

Параметр labels задает надписи для полей формы. Его значение должно представлять собой словарь, ключи элементов которого соответствуют полям формы, а значения задают надписи для них.

Параметр help_texts указывает дополнительные текстовые пояснения для полей формы (такой текст будет выводиться возле элементов управления). Значение этого параметра должно представлять собой словарь, ключи элементов которого соответствуют полям формы, а значения задают пояснения.

Параметр error_messages указывает сообщения об ошибках. Его значением должен быть словарь, ключи элементов которого соответствуют полям формы, а значениями элементов также должны быть словари. Во вложенных словарях ключи элементов соответствуют строковым кодам ошибок (см. *разд. 4.7.2*), а значения зададут строковые сообщения об ошибках.

Параметр field_classes указывает, поле какого типа должно быть создано в форме для соответствующего ему поля модели. Значением должен быть словарь, ключи элементов которого представляют имена полей модели, а значениями элементов станут ссылки на соответствующие им классы полей формы.

Параметр widgets позволяет задать элемент управления, которым будет представляться на веб-странице то или иное поле модели. Значением должен быть словарь, ключи элементов которого представляют имена полей формы, а значениями элементов станут экземпляры классов элементов управления или ссылки на сами эти классы.

Если какой-либо параметр не указан, то его значение либо будет взято из модели, либо установлено по умолчанию. Так, если не указать надпись для поля формы, то будет использовано название сущности, указанное в параметре verbose_name конструктора поля модели, а если не указать тип элемента управления, то будет использован элемент управления по умолчанию для поля этого типа.

И наконец, параметр form служит для указания *базовой формы, связанной с моделью*, на основе которой будет создана новая форма. Заданная *форма* может задавать какие-либо параметры, общие для целой группы форм.

Функция modelform_factory() в качестве результата возвращает готовый к использованию класс формы, связанной с моделью (подобные функции, генерирующие целые классы, называются *фабриками классов*).

Листинг 13.1 показывает код, создающий на основе модели `Bb` класс формы `BbForm` с применением фабрики классов.

Листинг 13.1. Использование фабрики классов `modelform_factory()`

```
from django.forms import modelform_factory, DecimalField
from django.forms.widgets import Select

from .models import Bb, Rubric

BbForm = modelform_factory(Bb,
         fields=('title', 'content', 'price', 'rubric'),
         labels={'title': 'Название товара'},
         help_texts={'rubric': 'Не забудьте выбрать рубрику!'},
         field_classes={'price': DecimalField},
         widgets={'rubric': Select(attrs={'size': 8})})
```

Ради эксперимента мы изменили надпись у поля названия товара, задали поясняющий текст у поля рубрики, сменили тип поля цены на `DecimalField` и указали для поля рубрики представление в виде обычного списка высотой в 8 пунктов.

Класс, сохраненный в переменной `BbForm`, можно использовать точно так же, как и любой написанный "вручную", — например, указать его в контроллере-классе:

```
class BbCreateView(CreateView):
    form_class = BbForm
    . . .
```

На основе фабрики классов удобно создавать в контроллерах-функциях редко используемые формы. В этом случае класс формы создается только при необходимости и уничтожается сразу же, как только перестанет существовать хранящая его переменная, благодаря чему экономится оперативная память.

13.1.2. Создание форм путем быстрого объявления

Если же форма, связанная с моделью, используется часто, то целесообразнее прибегнуть ко второму способу — объявить ее явно.

Класс формы, связанной с моделью, должен быть производным от класса `ModelForm` из модуля `django.forms`. В этом классе объявляется вложенный класс `Meta`, в котором записывается набор атрибутов, имеющих те же имена, что параметры функции `modelform_factory()`, и то же назначение.

Такой способ объявления формы, при котором в ее классе записываются лишь общие указания, носит название *быстрого объявления*.

В листинге 13.2 можно увидеть код класса формы `BbForm`, созданный посредством быстрого объявления.

Листинг 13.2. Быстрое объявление формы, связанной с моделью

```
from django.forms import ModelForm, DecimalField
from django.forms.widgets import Select
from .models import Bb

class BbForm(ModelForm):
    class Meta:
        model = Bb
        fields = ('title', 'content', 'price', 'rubric')
        labels = {'title': 'Название товара'}
        help_texts = {'rubric': 'Не забудьте задать рубрику!'}
        field_classes = {'price': DecimalField}
        widgets = {'rubric': Select(attrs={'size': 8})}
```

13.1.3. Создание форм путем полного объявления

Оба описанных ранее способа создания форм позволяли задать для ее полей весьма ограниченный набор параметров. Если же требуется описать поля формы во всех деталях, придется прибегнуть к третьему, сложному способу объявления.

13.1.3.1. Как выполняется полное объявление

При *полном объявлении* формы в ее классе детально описываются параметры как отдельных полей, так и — во вложенном классе `Meta` — самой формы. Полное объявление формы напоминает объявление модели (см. *главу 4*).

В листинге 13.3 приведен код полного объявления класса формы `BbForm`, связанной с моделью `Bb`.

Листинг 13.3. Полное объявление формы

```
from django import forms
from .models import Bb, Rubric

class BbForm(forms.ModelForm):
    title = forms.CharField(label='Название товара')
    content = forms.CharField(label='Описание',
              widget=forms.widgets.Textarea())
    price = forms.DecimalField(label='Цена', decimal_places=2)
    rubric = forms.ModelChoiceField(queryset=Rubric.objects.all(),
             label='Рубрика', help_text='Не забудьте задать рубрику!',
             widget=forms.widgets.Select(attrs={'size': 8}))

    class Meta:
        model = Bb
        fields = ('title', 'content', 'price', 'rubric')
```

При полном объявлении можно детально описать лишь некоторые поля формы, у которых требуется существенно изменить поведение. Параметры остальных полей формы можно указать путем быстрого объявления (см. *разд. 13.1.2*).

Такой подход иллюстрирует листинг 13.4. Там путем полного объявления создаются только поля `price` и `rubric`, а остальные поля созданы с применением быстрого объявления.

Листинг 13.4. Полное объявление отдельных полей формы

```
from django import forms
from .models import Bb, Rubric

class BbForm(forms.ModelForm):
    price = forms.DecimalField(label='Цена', decimal_places=2)
    rubric = forms.ModelChoiceField(queryset=Rubric.objects.all(),
             label='Рубрика', help_text='Не забудьте задать рубрику!',
             widget=forms.widgets.Select(attrs={'size': 8}))

    class Meta:
        model = Bb
        fields = ('title', 'content', 'price', 'rubric')
        labels = {'title': 'Название товара'}
```

Применение полного объявления позволило, в частности, задать число знаков после запятой для поля типа `DecimalField` (параметр `decimal_places`). Быстрое объявление не даст это сделать.

> ***Внимание!***
>
> **Если в классе формы присутствует и полное, и быстрое объявление какого-либо поля, то будет обработано только полное объявление. Параметры, записанные в быстром объявлении поля, будут проигнорированы.**

Применяя полное объявление, можно добавить в форму дополнительные поля, отсутствующие в связанной с формой модели. Этот прием показан в листинге 13.5, где в форму регистрации нового пользователя `RegisterUserForm` добавлены поля `password1` и `password2`, не существующие в модели `User`.

Листинг 13.5. Объявление в форме полей, не существующих в связанной модели

```
class RegisterUserForm(forms.ModelForm):
    password1 = forms.CharField(label='Пароль')
    password2 = forms.CharField(label='Пароль (повторно)')

    class Meta:
        model = User
        fields = ('username', 'email', 'first_name', 'last_name')
```

Поля, созданные таким образом, необязательно заносить в список из атрибута `fields` вложенного класса `Meta`. Однако в этом случае такие поля при выводе формы на экран окажутся в ее конце. Чтобы задать для них нужное местоположение, их следует включить в список из атрибута `fields` по нужным позициям. Пример:

```
class RegisterUserForm(forms.ModelForm):
    . . .
    class Meta:
        . . .
        fields = ('username', 'email', 'password1', 'password2',
                  'first_name', 'last_name')
```

13.1.3.2. Параметры, поддерживаемые всеми типами полей

Поле, созданное путем полного объявления, представляется отдельным атрибутом класса формы. Ему присваивается экземпляр класса, представляющего поле определенного типа. Дополнительные параметры создаваемого поля указываются в соответствующих им именованных параметрах конструктора класса этого поля.

Рассмотрим параметры, поддерживаемые полями всех типов:

- `label` — надпись для поля. Если не указан, то в качестве надписи будет использовано имя текущего поля;
- `help_text` — дополнительный поясняющий текст для текущего поля, который будет выведен возле элемента управления;
- `label_suffix` — суффикс, который будет добавлен к надписи для текущего поля.

  Если параметр не указан, то будет взято значение одноименного параметра, поддерживаемого конструктором класса формы (эти параметры мы рассмотрим в *главе 17*). Если и тот не указан, будет использовано значение по умолчанию — символ двоеточия;
- `initial` — начальное значение для поля формы. Если не указан, то поле не будет иметь начального значения;
- `required` — если `True`, то в поле обязательно должно быть занесено значение, если `False`, то поле может быть "пустым". Значение по умолчанию — `True`;
- `widget` — элемент управления, представляющий текущее поле на веб-странице. Значение может представлять собой либо ссылку на класс элемента управления, либо экземпляр этого класса.

  Если параметр не указан, будет использован элемент управления по умолчанию, применяемый для поля такого типа;
- `validators` — валидаторы для текущего поля. Задаются в таком же формате, что и для поля модели (см. *разд. 4.7.1*);
- `error_messages` — сообщения об ошибках. Задаются в таком же формате, что и аналогичные сообщения для поля модели (см. *разд. 4.7.2*);
- `disabled` — если `True`, то поле при выводе на экран станет недоступным, если `False` — доступным. Значение по умолчанию — `False`.

13.1.3.3. Классы полей форм

Все классы полей форм, поддерживаемые Django, объявлены в модуле `django.forms`. Каждое такое поле предназначено для занесения значения строго определенного типа. Многие классы полей поддерживают дополнительные параметры, указываемые в вызовах конструкторов.

- `CharField` — строковое или текстовое поле. Дополнительные параметры:
  - `min_length` — минимальная длина значения, заносимого в поле, в символах;
  - `max_length` — максимальная длина значения, заносимого в поле, в символах;
  - `strip` — если `True`, то из заносимого в поле значения будут удалены начальные и конечные пробелы, если `False`, то пробелы удаляться не будут (по умолчанию — `True`);
  - `empty_value` — величина, которой будет представляться "пустое" поле (по умолчанию — "пустая" строка);
- `EmailField` — адрес электронной почты в строковом виде. Дополнительные параметры:
  - `min_length` — минимальная длина почтового адреса в символах;
  - `max_length` — максимальная длина почтового адреса в символах;
- `URLField` — интернет-адрес в виде строки. Дополнительные параметры:
  - `min_length` — минимальная длина интернет-адреса в символах;
  - `max_length` — максимальная длина интернет-адреса в символах;
- `SlugField` — слаг.

  Поддерживается дополнительный параметр `allow_unicode`. Если его значение равно `True`, то хранящийся в поле слаг может содержать символы Unicode, если `False` — только символы из кодировки ASCII. Значение по умолчанию — `False`;
- `RegexField` — строковое значение, совпадающее с заданным регулярным выражением. Дополнительные параметры:
  - `regex` — регулярное выражение. Может быть задано как в виде строки, так и в виде объекта типа `re`;
  - `min_length` — минимальная длина значения, заносимого в поле, в символах;
  - `max_length` — максимальная длина значения, заносимого в поле, в символах;
  - `strip` — если `True`, то из заносимого в поле значения будут удалены начальные и конечные пробелы, если `False`, то пробелы удаляться не будут. Значение по умолчанию — `False`;
- `BooleanField` — логическое поле;
- `NullBooleanField` — то же самое, что `BooleanField`, но дополнительно позволяет хранить значение `null`;

- `IntegerField` — знаковое целочисленное поле обычной длины (32-разрядное). Дополнительные параметры:
  - `min_value` — минимальное значение, которое можно занести в поле;
  - `max_value` — максимальное значение, которое можно занести в поле;
- `FloatField` — вещественное число. Дополнительные параметры:
  - `min_value` — минимальное значение, которое можно занести в поле;
  - `max_value` — максимальное значение, которое можно занести в поле;
- `DecimalField` — вещественное число фиксированной точности, представленное объектом типа `Decimal` из модуля `decimal` Python. Дополнительные параметры:
  - `min_value` — минимальное значение, которое можно занести в поле;
  - `max_value` — максимальное значение, которое можно занести в поле;
  - `max_digits` — максимальное количество цифр в числе;
  - `decimal_places` — количество цифр в дробной части числа;
- `DateField` — значение даты, представленное в виде объекта типа `date` из модуля `datetime` Python.

  Дополнительный параметр `input_formats` задает последовательность поддерживаемых полем форматов даты. Значение по умолчанию определяется языковыми настройками проекта или параметром `DATE_INPUT_FORMATS` (см. *разд. 3.3.5*);
- `DateTimeField` — временна́я отметка в виде объекта типа `datetime` из модуля `datetime`.

  Дополнительный параметр `input_formats` задает последовательность поддерживаемых полем форматов временны́х отметок. Значение по умолчанию определяется языковыми настройками проекта или параметром `DATETIME_INPUT_FORMATS`;
- `TimeField` — значение времени в виде объекта типа `time` из модуля `datetime` Python.

  Дополнительный параметр `input_formats` задает последовательность поддерживаемых полем форматов времени. Значение по умолчанию определяется языковыми настройками проекта или параметром `TIME_INPUT_FORMATS`;
- `SplitDateTimeField` — то же самое, что и `DateTimeField`, но для занесения значений даты и времени применяются разные элементы управления. Дополнительные параметры:
  - `input_date_formats` — последовательность поддерживаемых полем форматов даты. Значение по умолчанию определяется языковыми настройками проекта или параметром `DATE_INPUT_FORMATS`;
  - `input_time_formats` — последовательность поддерживаемых полем форматов времени. Значение по умолчанию определяется языковыми настройками проекта или параметром `TIME_INPUT_FORMATS`;

- `DurationField` — промежуток времени, представленный объектом типа `timedelta` из модуля `datetime` Python;
- `ModelChoiceField` — поле внешнего ключа вторичной модели, создающее связь "один-со-многими" или "один-с-одним". Позволяет выбрать в списке только одну связываемую запись первичной модели. Дополнительные параметры:
  - `queryset` — набор записей первичной модели, на основе которого будет формироваться список;
  - `empty_label` — строка, обозначающая "пустой" пункт в списке связываемых записей. Значение по умолчанию: `"---------"`. Также можно убрать "пустой" пункт из списка, присвоив этому параметру значение `None`;
  - `to_field_name` — имя поля первичной модели, значение из которого будет сохраняться в текущем поле внешнего ключа. Значение по умолчанию — `None` (обозначает ключевое поле);
- `ModelMultipleChoiceField` — поле внешнего ключа ведущей модели, создающее связь "многие-со-многими". Позволяет выбрать в списке произвольное количество связываемых записей. Дополнительные параметры конструктора:
  - `queryset` — набор записей ведомой модели, на основе которого будет формироваться список;
  - `to_field_name` — имя поля ведомой модели, значение из которого будет сохраняться в текущем поле внешнего ключа. Значение по умолчанию — `None` (обозначает ключевое поле);
- `ChoiceField` — поле со списком, в которое можно занести только те значения, что приведены в списке. Значение записывается в поле в строковом формате.

  Обязательный параметр `choices` задает перечень значений, которые будут представлены в списке, в виде:
  - последовательности — в таком же формате, который применяется для задания параметра `choices` поля со списком моделей (см. *разд. 4.2.3*). Объекты-перечисления не поддерживаются;
  - ссылки на функцию, не принимающую параметров и возвращающую в качестве результата последовательность в таком же формате;
- `TypedChoiceField` — то же самое, что `ChoiceField`, но позволяет хранить в поле значение любого типа, а не только строкового. Дополнительные параметры:
  - `choices` — перечень значений для выбора, в описанном ранее формате;
  - `coerce` — ссылка на функцию, выполняющую преобразование типа значения, предназначенного для сохранения в поле. Должна принимать с единственным параметром исходное значение и возвращать в качестве результата то же значение, преобразованное к нужному типу;
  - `empty_value` — значение, которым будет представляться "пустое" поле (по умолчанию — "пустая" строка).

Значение, представляющее "пустое" поле, можно записать непосредственно в последовательность, заданную в параметре `choices`;

- `MultipleChoiceField` — то же самое, что `ChoiceField`, но позволяет выбрать в списке произвольное число пунктов;
- `TypedMultipleChoiceField` — то же самое, что и `TypedChoiceField`, но позволяет выбрать в списке произвольное число пунктов;
- `GenericIPAddressField` — IP-адрес, записанный для протокола IPv4 или IPv6, в виде строки. Дополнительные параметры:
  - `protocol` — допустимый протокол для записи IP-адресов, представленный в виде строки. Доступны значения: `"IPv4"`, `"IPv6"` и `"both"` (поддерживаются оба протокола). Значение по умолчанию: `"both"`;
  - `inpack_ipv4` — если `True`, то IP-адреса протокола IPv4, записанные в формате IPv6, будут преобразованы к виду, применяемому в IPv4. Если `False`, то такое преобразование не выполняется. Значение по умолчанию — `False`. Этот параметр принимается во внимание, только если для параметра `protocol` указано значение `"both"`.
- `UUIDField` — уникальный универсальный идентификатор, представленный объектом типа `UUID` из модуля `uuid` Python, в виде строки.

На заметку

Также поддерживаются классы полей формы `ComboField` и `MultiValueField`, из которых первый применяется в крайне специфических случаях, а второй служит для разработки на его основе других классов полей. Описание этих двух классов можно найти на странице https://docs.djangoproject.com/en/3.0/ref/forms/fields/.

13.1.3.4. Классы полей форм, применяемые по умолчанию

Каждому классу поля модели поставлен в соответствие определенный класс поля формы, используемый по умолчанию, если класс поля не указан явно. В табл. 13.1 приведены классы полей модели и соответствующие им классы полей формы, используемые по умолчанию.

***Таблица 13.1.** Классы полей модели и соответствующие им классы полей формы, используемые по умолчанию*

Классы полей модели	Классы полей формы
`CharField`	`CharField`. Параметр `max_length` получает значение от параметра `max_length` конструктора поля модели. Если параметр `null` конструктора поля модели имеет значение `True`, то параметр `empty_value` конструктора поля формы получит значение `None`
`TextField`	`CharField`, у которого в качестве элемента управления указана область редактирования (параметр `widget` имеет значение `Textarea`)

Таблица 13.1. (окончание)

Классы полей модели	Классы полей формы
EmailField	EmailField
URLField	URLField
SlugField	SlugField
BooleanField	BooleanField
BooleanField, если параметру null дано значение True	NullBooleanField
NullBooleanField	
IntegerField	IntegerField
SmallIntegerField	
BigIntegerField	IntegerField, у которого параметр min_value имеет значение –9223372036854775808, а параметр max_value — 9223372036854775807
PositiveIntegerField	IntegerField
PositiveSmallIntegerField	
FloatField	FloatField
DecimalField	DecimalField
DateField	DateField
DateTimeField	DateTimeField
TimeField	TimeField
DurationField	DurationField
GenericIPAddressField	GenericIPAddressField
AutoField	Не представляются в формах
BigAutoField	
ForeignKey	ModelChioceField
ManyToManyField	ModelMultipleChoiceField
Поле со списком любого типа	TypedChoiceField

13.1.4. Задание элементов управления

Любому полю формы можно сопоставить элемент управления, посредством которого в него будет заноситься значение, указав его в параметре widget поля.

13.1.4.1. Классы элементов управления

Все классы элементов управления являются производными от класса Widget из модуля django.forms.widgets. Этот класс поддерживает параметр конструктора attrs,

указывающий значения атрибутов HTML-тега, который помещается в код генерируемой страницы и создает элемент управления. Значением параметра должен быть словарь, ключи элементов которого совпадают с именами атрибутов тега, а значения элементов задают значения этих атрибутов.

Вот пример указания значения `8` для атрибута `size` тега `<select>`, создающего список (в результате будет создан обычный список высотой 8 пунктов):

```
widget = forms.widgets.Select(attrs={'size': 8})
```

Далее приведен список поддерживаемых Django классов элементов управления, которые также объявлены в модуле `django.forms.widgets`:

- `TextInput` — обычное поле ввода;
- `NumberInput` — поле для ввода числа;
- `EmailInput` — поле для ввода адреса электронной почты;
- `URLInput` — поле для ввода интернет-адреса;
- `PasswordInput` — поле для ввода пароля.

  Поддерживается дополнительный параметр `render_value`. Если присвоить ему значение `True`, то после неудачной валидации и повторного вывода формы на экран в поле ввода пароля будет подставлен набранный ранее пароль. Если задать параметру значение `False`, то поле будет выведено "пустым". Значение по умолчанию — `False`;
- `HiddenInput` — скрытое поле;
- `DateInput` — поле для ввода значения даты.

  Дополнительный параметр `format` задает формат для представления значения даты. Если он не указан, будет использован формат, заданный языковыми настройками проекта, или первый формат из списка, хранящегося в параметре `DATE_INPUT_FORMATS` (см. *разд. 3.3.5*);
- `SelectDateWidget` — то же, что и `DateInput`, но выводит три раскрывающихся списка: для выбора числа, месяца и года соответственно. Дополнительные параметры:

  - `years` — список или кортеж значений года, которые будут выводиться в раскрывающемся списке, задающем год. Если не указан, будет выведен набор из текущего года и 9 следующих за ним годов;
  - `months` — словарь месяцев, которые будут выводиться в раскрывающемся списке, задающем месяц. Ключами элементов этого словаря должны быть порядковые номера месяцев, начиная с 1 (1 — январь, 2 — февраль и т. д.), а значениями элементов — названия месяцев. Если параметр не указан, то будут выведены все месяцы;
  - `empty_label` — величина, представляющая "пустое" значение числа, месяца и года. Может быть указана в виде строки (она будет использована для представления "пустых" даты, месяца и года), списка или кортежа из трех строко-

вых элементов (первый будет представлять "пустой" год, второй — "пустой" месяц, третий — "пустое" число). Значение по умолчанию: "---". Пример:

```
published = forms.DateField(
            widget=forms.widgets.SelectDateWidget(
                  empty_label=('Выберите год', 'Выберите месяц',
                               'Выберите число')))
```

- `DateTimeInput` — поле для ввода временно́й отметки.

  Дополнительный параметр `format` указывает формат выводимой временно́й отметки. Если не указан, то будет использовано значение, заданное языковыми настройками проекта, или первый формат из списка, хранящегося в параметре `DATETIME_INPUT_FORMATS` (см. *разд. 3.3.5*);

- `SplitDateTimeWidget` — то же, что и `DateTimeInput`, но ввод значений даты и времени осуществляется в разные поля. Дополнительные параметры:

  - `date_format` — формат выводимой даты. Если не указан, то будет взято значение, заданное языковыми настройками проекта, или первый формат из списка, хранящегося в параметре `DATE_INPUT_FORMATS`;
  - `time_format` — формат выводимого времени. Если не указан, то будет использовано значение, заданное языковыми настройками проекта, или первый формат из списка, хранящегося в параметре `TIME_INPUT_FORMATS`;
  - `date_attrs` — значения атрибутов тега, создающего поля ввода даты;
  - `time_attrs` — значения атрибутов тега, создающего поля ввода времени.

  Значения обоих параметров задаются в том же виде, что и у описанного ранее параметра `attrs`;

- `TimeInput` — поле для ввода значения времени.

  Дополнительный параметр `format` задает формат выводимого времени. Если не указан, то будет использовано значение, заданное языковыми настройками проекта, или первый формат из списка, хранящегося в параметре `TIME_INPUT_FORMATS`;

- `Textarea` — область редактирования;

- `CheckboxInput` — флажок.

  Дополнительному параметру `check_test` можно присвоить ссылку на функцию, которая в качестве параметра принимает хранящееся в поле значение и возвращает `True`, если флажок должен быть выведен установленным, и `False`, если сброшенным;

- `Select` — список, обычный или раскрывающийся (зависит от значения атрибута `size` тега `<select>`), с возможностью выбора только одного пункта. Перечень выводимых пунктов он берет из параметра `choices` конструктора поля формы, с которым связан.

  Перечень пунктов, выводимых в списке, также можно присвоить дополнительному параметру `choices`. Он задается в том же формате, что и у параметра

`choices` конструктора поля `ChoiceField` (см. *разд. 13.1.3.3*), и имеет приоритет перед таковым, указанным в параметрах поля формы;

- `RadioSelect` — аналогичен `Select`, но выводится в виде группы переключателей;
- `SelectMultiple` — то же самое, что и `Select`, но позволяет выбрать произвольное число пунктов;
- `CheckboxSelectMultiple` — аналогичен `SelectMultiple`, но выводится в виде набора флажков;
- `NullBooleanSelect` — раскрывающийся список с пунктами **Да**, **Нет** и **Неизвестно**.

13.1.4.2. Элементы управления, применяемые по умолчанию

Для каждого класса поля формы существует класс элемента управления, применяемый для его представления по умолчанию. Эти классы приведены в табл. 13.2.

Таблица 13.2. *Классы полей формы и соответствующие им классы элементов управления, используемые по умолчанию*

Классы полей формы	Классы элементов управления
`CharField`	`TextInput`
`EmailField`	`EmailInput`
`URLField`	`URLInput`
`SlugField`	`TextInput`
`RegexField`	
`BooleanField`	`CheckboxInput`
`NullBooleanField`	`NullBooleanSelect`
`IntegerField`	`NumberInput`
`FloatField`	
`DecimalField`	
`DateField`	`DateInput`
`DateTimeField`	`DateTimeInput`
`TimeField`	`TimeInput`
`SplitDateTimeField`	`SplitDateTimeWidget`
`DurationField`	`TextInput`
`ModelChoiceField`	`Select`
`ModelMultipleChoiceField`	`SelectMultiple`
`ChoiceField`	`Select`
`TypedChoiceField`	
`MultipleChoiceField`	`SelectMultiple`
`TypedMultipleChoiceField`	

Таблица 13.2 (окончание)

Классы полей формы	Классы элементов управления
GenericIPAddressField	TextInput
UUIDField	

13.2. Обработка форм

Высокоуровневые контроллеры-классы (см. *главу 11*) обработают и сохранят данные из формы самостоятельно. Но при использовании контроллеров-классов низкого уровня или контроллеров-функций обрабатывать формы придется вручную.

13.2.1. Добавление записи посредством формы

13.2.1.1. Создание формы для добавления записи

Для добавления записи следует создать экземпляр класса формы, поместить его в контекст шаблона и выполнить рендеринг шаблона, представляющего страницу добавления записи, тем самым выведя форму на экран. Все эти действия выполняются при отправке клиентом запроса с помощью HTTP-метода GET.

Экземпляр класса формы создается вызовом конструктора без параметров:

```
bbf = BbForm()
```

Если требуется поместить в форму какие-то изначальные данные, то используется необязательный параметр `initial` конструктора класса формы. Ему присваивается словарь, ключи элементов которого задают имена полей формы, а значения элементов — изначальные значения для этих полей. Пример:

```
bbf = BbForm(initial={price=1000.0})
```

Пример простейшего контроллера-функции, создающего форму и выводящего ее на экран, можно увидеть в листинге 9.1. А листинг 9.3 иллюстрирует более сложный контроллер-функцию, который при получении запроса, отправленного методом GET, выводит форму на экран, а при получении POST-запроса сохраняет данные из формы в модели, там самым создавая новую запись.

13.2.1.2. Повторное создание формы

После ввода посетителем данных в форму и нажатия кнопки отправки веб-обозреватель выполняет POST-запрос, отправляющий введенные в форму данные. Получив такой запрос, контроллер "поймет", что нужно выполнить валидацию данных из формы и, если она пройдет успешно, сохранить эти данные в новой записи модели.

Сначала нужно создать экземпляр класса формы еще раз и поместить в него данные, полученные в составе POST-запроса. Это выполняется вызовом конструктора

класса формы с передачей ему в качестве первого позиционного параметра словаря с полученными данными, который можно извлечь из атрибута POST объекта запроса (подробности — в *разд. 9.4*). Пример:

```
bbf = BbForm(request.POST)
```

После этого форма обработает полученные из запроса данные и подготовится к валидации.

Имеется возможность проверить, были ли в форму при ее создании помещены данные из запроса. Для этого достаточно вызвать метод is_bound(), поддерживаемый классом ModelForm. Метод вернет True, если при создании формы в нее были помещены данные из POST-запроса (т. е. выполнялось повторное создание формы), и False — в противном случае (т. е. форма создавалась впервые).

13.2.1.3. Валидация данных, занесенных в форму

Чтобы запустить валидацию формы, нужно выполнить одно из двух действий:

- вызвать метод is_valid() формы. Он вернет True, если занесенные в форму данные корректны, и False — в противном случае. Пример:

```
if bbf.is_valid():
    # Данные корректны, и их можно сохранять
else:
    # Данные некорректны
```

- обратиться к атрибуту errors формы. Он хранит словарь с сообщениями о допущенных посетителем ошибках. Ключи элементов этого словаря совпадают с именами полей формы, а значениями являются списки текстовых сообщений об ошибках.

  Ключ, совпадающий со значением переменной NON_FIELD_ERRORS из модуля django.core.exceptions, хранит сообщения об ошибках, относящихся не к определенному полю формы, а ко всей форме.

  Если данные, занесенные в форму, корректны, то атрибут errors будет хранить "пустой" словарь.

  Пример:

```
from django.core.exceptions import NON_FIELDS_ERRORS
. . .
if bbf.errors:
    # Данные некорректны
    # Получаем список сообщений об ошибках, допущенных при вводе
    # названия товара
    title_errors = bbf.errors['title']
    # Получаем список ошибок, относящихся ко всей форме
    form_errors = bbf.errors[NON_FIELDS_ERRORS]
else:
    # Данные корректны, и их можно сохранять
```

Если данные корректны, то их следует сохранить и выполнить перенаправление на страницу со списком записей или содержанием только что добавленной записи, чтобы посетитель сразу смог увидеть, увенчалась ли его попытка успехом.

Если же данные некорректны, то необходимо повторно вывести страницу с формой на экран. В форме, рядом с элементами управления, будут показаны все относящиеся к ним сообщения об ошибках, и посетитель сразу поймет, что́ он сделал не так.

13.2.1.4. Сохранение данных, занесенных в форму

Сохранить данные, занесенные в связанную с моделью форму, можно вызовом метода `save()` формы:

```
bbf.save()
```

Перед сохранением данных из формы настоятельно рекомендуется выполнить их валидацию. Если этого не сделать, то метод `save()` перед сохранением выполнит валидацию самостоятельно и, если она не увенчалась успехом, возбудит исключение `ValueError`. А обрабатывать результат, возвращенный методом `is_valid()`, удобнее, чем исключение (по крайней мере, на взгляд автора).

Метод `save()` в качестве результата возвращает объект созданной или исправленной записи модели, связанной с текущей формой.

Есть возможность получить только что созданную, но еще не сохраненную, запись модели, чтобы внести в нее какие-либо правки. Сделать это можно, записав в вызове метода `save()` необязательный параметр `commit` и присвоив ему значение `False`. Объект записи будет возвращен методом `save()` в качестве результата, но сама запись сохранена не будет. Пример:

```
bb = bbf.save(commit=False)
if not bb.kind:
    bb.kind = 's'
bb.save()
```

При сохранении записи модели, связанной с другой моделью связью "многие-со-многими", нужно иметь в виду один момент. Чтобы связь между записями была успешно создана, связываемая запись должна иметь ключ (поскольку именно он записывается в связующей таблице). Однако пока запись не сохранена, ключа у нее нет. Поэтому сначала нужно сохранить запись вызовом метода `save()` у самой модели, а потом создать связь вызовом метода `save_m2m()` формы. Вот пример:

```
mf = MachineForm(request.POST)
if mf.is_valid():
    machime = mf.save(commit=False)
    # Выполняем какие-либо дополнительные действия с записью
    machine.save()
    mf.save_m2m()
```

Отметим, что метод `save_m2m()` нужно вызывать только в том случае, если сохранение записи выполнялось вызовом метода `save()` формы с параметром `commit`, рав-

ным `False`, и последующим вызовом метода `save()` модели. Если запись сохранялась вызовом `save()` формы без параметра `commit` (или если для этого параметра было указано значение по умолчанию `True`), то метод `save_m2m()` вызывать не нужно — форма сама сохранит запись и создаст связь. Пример:

```
mf = MachineForm(request.POST)
if mf.is_valid():
    mf.save()
```

13.2.1.5. Доступ к данным, занесенным в форму

Иногда бывает необходимо извлечь из формы занесенные в нее данные, чтобы, скажем, сформировать на их основе интернет-адрес перенаправления. Эти данные, приведенные к нужному типу (строковому, целочисленному, логическому и др.), хранятся в атрибуте `cleaned_data` формы в виде словаря, ключи элементов которого совпадают с именами полей формы, а значениями элементов станут значения, введенные в эти поля.

Вот пример использования ключа рубрики, указанной в только что созданном объявлении, для формирования адреса перенаправления:

```
return HttpResponseRedirect(reverse('bboard:by_rubric',
       kwargs={'rubric_id': bbf.cleaned_data['rubric'].pk}))
```

Полный пример контроллера, создающего записи с применением формы, приведен в листинге 9.2. А листинг 9.3 показывает пример более сложного контроллера, который и выводит форму, и сохраняет занесенную в нее запись.

13.2.2. Правка записи посредством формы

Чтобы исправить уже имеющуюся в модели запись посредством формы, связанной с моделью, нужно выполнить следующие шаги:

1. При получении запроса по HTTP-методу GET создать форму для правки записи. В этом случае нужно указать исправляемую запись, задав ее в параметре `instance` конструктора класса формы. Пример:

   ```
   bb = Bb.objects.get(pk=pk)
   bbf = BbForm(instance=bb)
   ```

2. Вывести страницу с формой на экран.
3. После получения POST-запроса с исправленными данными создать форму во второй раз, указав первым позиционным параметром полученные данные, извлеченные из атрибута `POST` запроса, а параметром `instance` — исправляемую запись.
4. Выполнить валидацию формы:
   - если валидация прошла успешно, сохранить запись. После этого обычно выполняется перенаправление;
   - если валидация завершилась неудачей, повторно вывести страницу с формой.

В листинге 13.6 приведен код контроллера-функции, осуществляющего правку записи, ключ которой был получен с URL-параметром `pk`. Здесь используется шаблон bboard\bb_form.html, написанный в *главе 10*.

Листинг 13.6. Контроллер-функция, исправляющий запись

```
def edit(request, pk):
    bb = Bb.objects.get(pk=pk)
    if request.method == 'POST':
        bbf = BbForm(request.POST, instance=bb)
        if bbf.is_valid():
            bbf.save()
            return HttpResponseRedirect(reverse('bboard:by_rubric',
                   kwargs={'rubric_id': bbf.cleaned_data['rubric'].pk}))
        else:
            context = {'form': bbf}
            return render(request, 'bboard/bb_form.html', context)
    else:
        bbf = BbForm(instance=bb)
        context = {'form': bbf}
        return render(request, 'bboard/bb_form.html', context)
```

При правке записи (и, в меньшей степени, при ее создании) может пригодиться метод `has_changed()`. Он возвращает `True`, если данные в форме были изменены посетителем, и `False` — в противном случае. Пример:

```
if bbf.is_valid():
    if bbf.has_changed():
        bbf.save()
```

Атрибут `changed_data` формы хранит список имен полей формы, значения которых были изменены посетителем.

13.2.3. Некоторые соображения касательно удаления записей

Для удаления записи нужно выполнить такие шаги:

1. Извлечь запись, подлежащую удалению.
2. Вывести на экран страницу с предупреждением об удалении записи.

   Эта страница должна содержать форму с кнопкой отправки данных. После нажатия кнопки веб-обозреватель отправит POST-запрос, который послужит контроллеру сигналом того, что посетитель подтвердил удаление записи.
3. После получения POST-запроса в контроллере удалить запись.

Код контроллера-функции, удаляющего запись, ключ которой был получен через URL-параметр `pk`, приведен в листинге 13.7. Код шаблона bboard\bb_confirm_delete.html можно найти в листинге 10.8.

Листинг 13.7. Контроллер-функция, удаляющий запись

```
def delete(request, pk):
    bb = Bb.objects.get(pk=pk)
    if request.method == 'POST':
        bb.delete()
        return HttpResponseRedirect(reverse('bboard:by_rubric',
               kwargs={'rubric_id': bb.rubric.pk}))
    else:
        context = {'bb': bb}
        return render(request, 'bboard/bb_confirm_delete.html', context)
```

13.3. Вывод форм на экран

13.3.1. Быстрый вывод форм

Быстрый вывод форм осуществляется вызовом одного метода из трех, поддерживаемых Django:

- `as_p()` — вывод по абзацам. Надпись для элемента управления и сам элемент управления, представляющий какое-либо поле формы, выводятся в отдельном абзаце и отделяются друг от друга пробелами. Пример использования:

```
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Добавить">
</form>
```

- `as_ul()` — вывод в виде маркированного списка. Надпись и элемент управления выводятся в отдельном пункте списка и отделяются друг от друга пробелами. Теги `<ul>` и `</ul>` не формируются. Пример:

```
<form method="post">
    {% csrf_token %}
    <ul>
        {{ form.as_ul }}
    </ul>
    <input type="submit" value="Добавить">
</form>
```

- `as_table()` — вывод в виде таблицы из двух столбцов: в левом выводятся надписи, в правом — элементы управления. Каждая пара "надпись—элемент управления" занимает отдельную строку таблицы. Теги `<table>` и `</table>` не выводятся. Пример:

```
<form method="post">
    {% csrf_token %}
    <table>
        {{ form.as_table }}
    </table>
```

```
    <input type="submit" value="Добавить">
</form>
```

Можно просто указать переменную, хранящую форму, — метод `as_table()` будет вызван автоматически:

```
<form method="post">
    {% csrf_token %}
    <table>
        {{ form }}
    </table>
    <input type="submit" value="Добавить">
</form>
```

Обязательно уясним следующие моменты:

- тег `<form>`, создающий саму форму, не выводится в любом случае. Его придется вставить в код шаблона самостоятельно;
- кнопка отправки данных также не выводится, и ее тоже следует поместить в форму самостоятельно. Такая кнопка формируется одинарным тегом `<input>` с атрибутом `type`, значение которого равно `"submit"`;
- не забываем поместить в форму тег шаблонизатора `csrf_token`. Он создаст в форме скрытое поле с электронным жетоном, по которому Django проверит, пришел ли запрос с того же самого сайта. Это сделано ради безопасности.

По умолчанию в веб-формах применяется метод кодирования данных `application/x-www-form-urlencoded`. Но если форма отправляет файлы, то в ней нужно указать метод `multipart/form-data`.

Выяснить, какой метод следует указать в теге `<form>`, поможет метод `is_multipart()`, поддерживаемый формой. Он возвращает `True`, если форма содержит поля, предназначенные для хранения файлов (мы познакомимся с ними в *главе 20*), и, соответственно, требует указания метода `multipart/form-data`, и `False` — в противном случае. Пример:

```
{% if form.is_multipart %}
    <form enctype="multipart/form-data" method="post">
{% else %}
    <form method="post">
{% endif %}
```

13.3.2. Расширенный вывод форм

Django предоставляет инструменты *расширенного вывода* форм, позволяющие располагать отдельные элементы управления произвольно.

Прежде всего экземпляр класса `ModelForm`, представляющий связанную с моделью форму, поддерживает функциональность словаря. Ключи элементов этого словаря совпадают с именами полей формы, а значениями элементов являются экземпляры класса `BoundField`, которые представляют отдельные поля формы.

Если указать в директиве шаблонизатора непосредственно экземпляр класса `BoundField`, то он будет выведен как HTML-код, создающий элемент управления для текущего поля. Вот так можно вывести область редактирования для ввода описания товара:

```
{{ form.content }}
```

В результате мы получим такой HTML-код:

```
<textarea name="content" cols="40" rows="10" id="id_content"></textarea>
```

Еще класс `BoundField` поддерживает следующие атрибуты:

- `label_tag` — HTML-код, создающий надпись для элемента управления, включая тег `<label>`. Вот пример вывода кода, создающего надпись для области редактирования, в которую заносится описание товара:

  ```
  {{ form.content.label_tag }}
  ```

  Результирующий HTML-код:

  ```
  <label for="id_content">Описание:</label>
  ```

- `label` — только текст надписи;
- `help_text` — дополнительный поясняющий текст;
- `errors` — список сообщений об ошибках, относящихся к текущему полю.

  Список ошибок можно вывести в шаблоне непосредственно:

  ```
  {{ form.content.errors }}
  ```

  В этом случае будет сформирован маркированный список с привязанным стилевым классом `errorlist`, а отдельные ошибки будут выведены как пункты этого списка, например:

  ```
  <ul class="errorlist">
      <li>Укажите описание продаваемого товара</li>
  </ul>
  ```

  Также можно перебрать список ошибок в цикле и вывести отдельные его элементы с применением любых других HTML-тегов.

  Чтобы получить список сообщений об ошибках, относящихся ко всей форме, следует вызвать метод `non_field_errors()` формы:

  ```
  {{ form.non_field_errors }}
  ```

- `is_hidden` — `True`, если это скрытое поле, `False`, если какой-либо иной элемент управления.

Методы класса `ModelForm`:

- `visible_fields()` — возвращает список видимых полей, которые представляются на экране обычными элементами управления;
- `hidden_fields()` — возвращает список невидимых полей, представляющихся скрытыми полями HTML.

В листинге 13.8 приведен код, создающий форму для добавления объявления, в которой сообщения об ошибках выводятся курсивом; надписи, элементы управления и поясняющие тексты разделяются разрывом строки (HTML-тегом
); а невидимые поля (если таковые есть) выводятся отдельно от видимых. Сама форма показана на рис. 13.1.

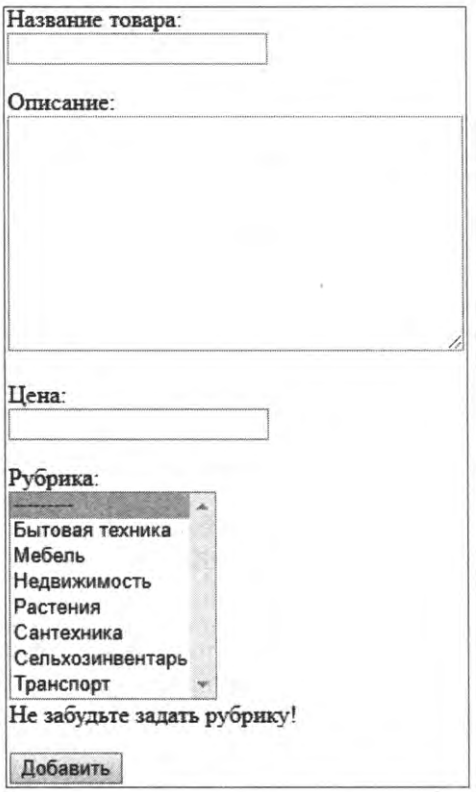


Рис. 13.1. Веб-форма, код которой приведен в листинге 13.8

Листинг 13.8. Отображение формы средствами расширенного вывода

```
<form method="post">
    {% csrf_token %}
    {% for hidden in form.hidden_fields %}
        {{ hidden }}
    {% endfor %}
    {% if form.non_field_errors %}
    <ul>
        {% for error in form.non_field_errors %}
        <li><em>{{ error|escape }}</em></li>
        {% endfor %}
    </ul>
    {% endif %}
```

```
    {% for field in form.visible_fields %}
        {% if field.errors %}
        <ul>
            {% for error in field.errors %}
            <li><em>{{ error|escape }}</em></li>
            {% endfor %}
        </ul>
        {% endif %}
        <p>{{ field.label_tag }}<br>{{ field }}<br>
        {{ field.help_text }}</p>
    {% endfor %}
    <p><input type="submit" value="Добавить"></p>
</form>
```

13.4. Валидация в формах

Валидацию можно выполнять не только в модели (см. *разд. 4.7*), но и в формах, связанных с моделями, применяя аналогичные инструменты.

13.4.1. Валидация полей формы

Валидацию отдельных полей формы можно реализовать двумя способами: с применением валидаторов или путем переопределения методов формы.

13.4.1.1. Валидация с применением валидаторов

Валидация с помощью валидаторов в полях формы выполняется так же, как и в полях модели (см. *разд. 4.7*). Вот пример проверки, содержит ли название товара больше четырех символов, с выводом собственного сообщения об ошибке:

```
from django.core import validators

class BbForm(forms.ModelForm):
    title = forms.CharField(label='Название товара',
          validators=[validators.RegexValidator(regex='^.{4,}$')],
          error_messages={'invalid': 'Слишком короткое название товара'})
    . . .
```

Разумеется, мы можем использовать не только стандартные валидаторы, объявленные в модуле `django.core.validators`, но и свои собственные.

13.4.1.2. Валидация путем переопределения методов формы

Более сложная валидация значения какого-либо *поля* реализуется в классе формы, в переопределенном методе с именем вида `clean_<имя поля>(self)`. Этот метод должен получать значение *поля* из словаря, хранящегося в атрибуте `cleaned_data`, не должен принимать параметров и всегда обязан возвращать значение проверяе-

мого поля. Если значение не проходит валидацию, то в методе следует возбудить исключение `ValidationError`.

Вот пример проверки, не собирается ли посетитель выставить на продажу прошлогодний снег:

```
from django.core.exceptions import ValidationError

class BbForm(forms.ModelForm):
    . . .
    def clean_title(self):
        val = self.cleaned_data['title']
        if val == 'Прошлогодний снег':
            raise ValidationError('К продаже не допускается')
        return val
```

13.4.2. Валидация формы

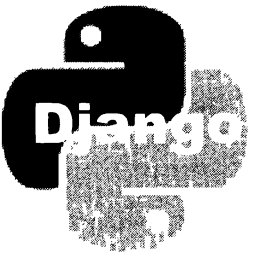
Выполнить более сложную проверку или проверить значения сразу нескольких полей формы, т. е. провести *валидацию формы*, можно в переопределенном методе `clean(self)` класса формы.

Метод не должен принимать параметров, возвращать результата, а обязан при неудачной валидации возбудить исключение `ValidationError`, чтобы указать на возникшую ошибку. Первым же действием он должен вызвать одноименный метод базового класса, чтобы он заполнил словарь, хранящийся в атрибуте `cleaned_data` (если мы этого не сделаем, то не сможем получить данные, занесенные в форму).

Далее приведен пример реализации такой же проверки, как в *разд. 4.7.4* (описание товара должно быть занесено, а значение цены должно быть неотрицательным).

```
from django.core.exceptions import ValidationError

class BbForm(forms.ModelForm):
    . . .
    def clean(self):
        super().clean()
        errors = {}
        if not self.cleaned_data['content']:
            errors['content'] = ValidationError(
                                'Укажите описание продаваемого товара')
        if self.cleaned_data['price'] < 0:
            errors['price'] = ValidationError(
                              'Укажите неотрицательное значение цены')
        if errors:
            raise ValidationError(errors)
```



ГЛАВА 14

Наборы форм, связанные с моделями

Если обычная форма, связанная с моделью, позволяет работать лишь с одной записью, то *набор форм, связанный с моделью*, дает возможность работы сразу с несколькими записями. Внешне он представляет собой группу форм, в каждой из которых отображается содержимое одной записи. Помимо того, там могут быть выведены "пустые" формы для добавления записей и специальные средства для переупорядочивания и удаления записей.

Можно сказать, что один набор форм, связанный с моделью, заменяет несколько страниц: страницу списка записей и страницы добавления, правки и удаления записей. Вот только, к сожалению, с наборами форм удобно работать лишь тогда, когда количество отображающихся в них записей невелико.

14.1. Создание наборов форм, связанных с моделями

Для создания наборов форм, связанных с моделями, применяется быстрое объявление посредством фабрики классов — функции `modelformset_factory()` из модуля `django.forms`:

```
modelformset_factory(<модель>[, form=<форма, связанная с моделью>][,
                     fields=None][, exclude=None][, labels=None][,
                     help_texts=None][, error_messages=None][,
                     field_classes=None][, widgets=None][, extra=1][,
                     can_order=False][, can_delete=False][,
                     min_num=None][, validate_min=False][,
                     max_num=None][, validate_max=False][,
                     formset=<базовый набор форм, связанный с моделью>])
```

Параметров здесь очень много:

- первый, позиционный — модель, связываемая с формируемым набором форм;
- `form` — *форма, связанная с моделью*, на основе которой будет создан набор форм. Если параметр не указан, то форма будет создана автоматически;

- `fields` — последовательность имен полей модели, включаемых в форму, автоматически создаваемую для набора. Чтобы указать все поля модели, нужно присвоить этому параметру строку `"__all__"`;
- `exclude` — последовательность имен полей модели, которые, напротив, не должны включаться в форму, автоматически создаваемую для набора;

> ***Внимание!***
>
> В вызове функции `modelformset_factory()` должен присутствовать только один из следующих параметров: `form`, `fields`, `exclude`. Одновременное указание двух или более параметров приведет к ошибке.

- `labels` — надписи для полей формы. Указываются в виде словаря, ключи элементов которого соответствуют полям, а значения задают надписи для них;
- `help_texts` — дополнительные текстовые пояснения для полей формы. Указывается в виде словаря, ключи элементов которого соответствуют полям, а значения задают пояснения для них;
- `error_messages` — сообщения об ошибках. Задаются в виде словаря, ключи элементов которого соответствуют полям формы, а значениями элементов также должны быть словари. Во вложенных словарях ключи элементов соответствуют строковым кодам ошибок (см. *разд. 4.7.2*), а значения зададут строковые сообщения об ошибках;
- `field_classes` — типы полей формы, которыми будут представляться в создаваемой форме различные поля модели. Значением должен быть словарь, ключи элементов которого совпадают с именами полей модели, а значениями элементов станут ссылки на классы полей формы;
- `widgets` — элементы управления, представляющие различные поля формы. Значение — словарь, ключи элементов которого совпадают с именами полей формы, а значениями элементов станут экземпляры классов элементов управления или ссылки на сами эти классы;

> ***Внимание!***
>
> Параметры `labels`, `help_texts`, `error_messages`, `field_classes` и `widgets` указываются только в том случае, если форма для набора создается автоматически (параметр `form` не указан). В противном случае все необходимые сведения о форме должны быть записаны в ее классе.

- `extra` — количество "пустых" форм, предназначенных для ввода новых записей, которые будут присутствовать в наборе (по умолчанию — 1);
- `can_order` — если `True`, то посредством набора форм можно переупорядочивать записи связанной с ним модели, если `False` — нельзя (поведение по умолчанию);
- `can_delete` — если `True`, то посредством набора форм можно удалять записи связанной с ним модели, если `False` — нельзя (поведение по умолчанию);
- `min_num` — минимальное количество форм в наборе, за вычетом помеченных на удаление (по умолчанию не ограничено);

- `validate_min` — если `True`, то Django в процессе валидации будет проверять, не меньше ли количество форм значения из параметра `min_num`, если `False` — не будет (поведение по умолчанию). Если количество форм меньше указанного минимального, будет выведено сообщение об ошибке с кодом `"too_few_forms"` (этот код можно использовать для указания своего сообщения об ошибке, более подробно об этом см. в *разд. 4.7.2*);
- `max_num` — максимальное количество форм в наборе, за вычетом помеченных на удаление (по умолчанию не ограничено);
- `validate_max` — если `True`, то Django в процессе валидации будет проверять, не превосходит ли количество форм значение из параметра `max_num`, если `False` — не будет (поведение по умолчанию). Если количество форм больше указанного максимального, будет выведено сообщение об ошибке с кодом `"too_many_forms"`;
- `formset` — *базовый набор форм, связанный с моделью*, на основе которого будет создан новый набор форм. Он должен быть производным от класса `BaseModelFormSet` из модуля `django.forms` и может указывать какие-либо общие для целой группы наборов форм функции (например, реализовывать валидацию — см. *разд. 14.4*).

Пример создания набора форм, предназначенного для работы со списком рубрик и имеющего минимальную функциональность:

```
from django.forms import modelformset_factory
from .models import Rubric

RubricFormSet = modelformset_factory(Rubric, fields=('name',))
```

На экране он будет выглядеть как простая последовательность форм, каждая из которых служит для правки одной записи модели (рис. 14.1).

Рис. 14.1. Набор форм с базовой функциональностью

А вот пример создания аналогичного набора форм, позволяющего переупорядочивать и удалять рубрики:

```
RubricFormSet = modelformset_factory(Rubric, fields=('name',),
                                     can_order=True, can_delete=True)
```

Набор форм с подобного рода расширенной функциональностью показан на рис. 14.2. Видно, что в составе каждой формы находятся поле ввода **Порядок** и флажок **Удалить**. В поле ввода заносится целочисленное значение, по которому записи модели могут быть отсортированы. А установка флажка приведет к тому, что после нажатия на кнопку отправки данных соответствующие записи будут удалены из модели.

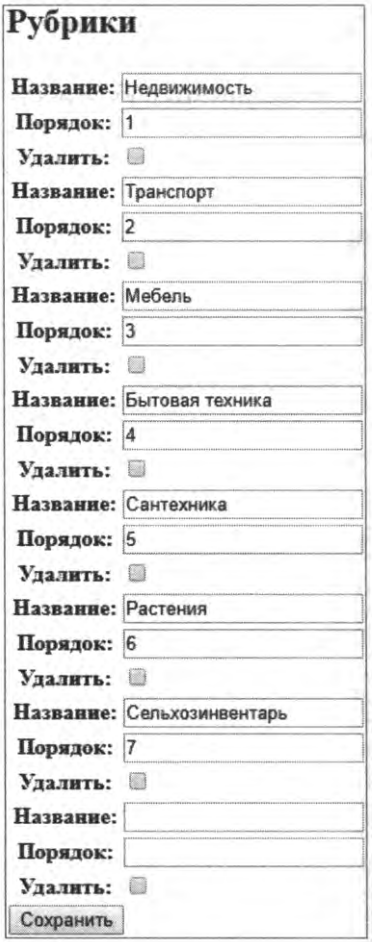


Рис. 14.2. Набор форм с расширенной функциональностью

14.2. Обработка наборов форм, связанных с моделями

Высокоуровневые контроллеры-классы, описанные в *главе 10*, не "умеют" работать с наборами форм. Поэтому их в любом случае придется обрабатывать вручную.

14.2.1. Создание набора форм, связанного с моделью

Экземпляр набора форм создается вызовом конструктора его класса без параметров:

```
formset = RubricFormSet()
```

Конструктор класса набора форм поддерживает два необязательных параметра, которые могут пригодиться:

- `initial` — изначальные данные, которые будут помещены в "пустые" (предназначенные для добавления новых записей) формы. Значение параметра должно представлять собой последовательность, каждый элемент которой задаст изначальные значения для одной из "пустых" форм. Этим элементом должен выступать словарь, ключи элементов которого совпадают с именами полей "пустой" формы, а значения элементов зададут изначальные значения для этих полей;
- `queryset` — набор записей для вывода в наборе форм.

Для примера зададим в качестве изначального значения для поля названия в первой "пустой" форме строку `"Новая рубрика"`, во второй — строку `"Еще одна новая рубрика"` и сделаем так, чтобы в наборе форм выводились только первые пять рубрик:

```
formset = RubricFormSet(initial=[{'name': 'Новая рубрика'},
                                 {'name': 'Еще одна новая рубрика'}],
                        queryset=Rubric.objects.all()[0:5])
```

14.2.2. Повторное создание набора форм

Закончив работу, посетитель нажмет кнопку отправки данных, в результате чего веб-обозреватель отправит POST-запрос с данными, занесенными в набор форм. Этот запрос нужно обработать.

Прежде всего, следует создать набор форм повторно. Это выполняется так же, как и в случае формы, — вызовом конструктора класса с передачей ему единственного позиционного параметра — словаря из атрибута `POST` объекта запроса. Пример:

```
formset = RubricFormSet(request.POST)
```

Если при первом создании набора форм в параметре `queryset` был указан набор записей, то при повторном создании также следует его указать:

```
formset = RubricFormSet(request.POST, queryset=Rubric.objects.all()[0:5])
```

14.2.3. Валидация и сохранение набора форм

Валидация и сохранение набора форм выполняется описанными в *разд. 13.2* методами `is_valid()`, `save()` и `save_m2m()`, поддерживаемыми классом набора форм:

```
if formset.is_valid():
    formset.save()
```

Метод `save()` в качестве результата вернет последовательность всех записей модели, представленных в текущем наборе форм. Эту последовательность можно перебрать в цикле и выполнить над записями какие-либо действия (что может пригодиться при вызове метода `save()` с параметром `commit`, равным `False`).

После вызова метода `save()` будут доступны три атрибута, поддерживаемые классом набора форм:

- `new_objects` — последовательность добавленных записей модели, связанной с текущим набором форм;

- `changed_objects` — последовательность исправленных записей модели, связанной с текущим набором форм;
- `deleted_objects` — последовательность удаленных записей модели, связанной с текущим набором форм.

Метод `save()` самостоятельно обрабатывает удаление записей (если при вызове конструктора набора форм был указан параметр `can_delete` со значением `True`). Если посетитель в форме установит флажок **Удалить**, то соответствующая запись модели будет удалена.

Однако при вызове метода `save()` с параметром `commit`, равным `False`, помеченные на удаление записи удалены не будут. Понадобится перебрать все удаленные записи, перечисленные в списке из атрибута `deleted_objects`, и вызвать у каждой метод `delete()`:

```
formset.save(commit=False)
for rubric in formset.deleted_objects:
    rubric.delete()
```

14.2.4. Доступ к данным, занесенным в набор форм

Каждая форма, входящая в состав набора, поддерживает описанный в *разд. 13.2.1.5* атрибут `cleaned_data`. Его значением является словарь, хранящий все данные, которые были занесены в текущую форму, в виде объектов языка Python.

Сам набор форм поддерживает функциональность последовательности. На каждой итерации он возвращает очередную форму, входящую в его состав. Вот так можно перебрать в цикле входящие в набор формы:

```
for form in formset:
    # Что-либо делаем с формой и введенными в нее данными
```

Нужно иметь в виду, что в наборе, возможно, будет присутствовать "пустая" форма, в которую не были занесены никакие данные. Такая форма будет хранить в атрибуте `cleaned_data` "пустой" словарь. Обработать эту ситуацию можно следующим образом:

```
for form in formset:
    if form.cleaned_data:
        # Форма не пуста, и мы можем получить занесенные в нее данные
```

Листинг 14.1 содержит полный код контроллера-функции, обрабатывающего набор форм и позволяющего удалять записи.

Листинг 14.1. Обработка набора форм, связанного с моделью

```
from django.shortcuts import render, redirect
from django.forms import modelformset_factory
from .models import Rubric
```

```
def rubrics(request):
    RubricFormSet = modelformset_factory(Rubric, fields=('name',),
                                         can_delete=True)
    if request.method == 'POST':
        formset = RubricFormSet(request.POST)
        if formset.is_valid():
            formset.save()
            return redirect('bboard:index')
    else:
        formset = RubricFormSet()
    context = {'formset': formset}
    return render(request, 'bboard/rubrics.html', context)
```

14.2.5. Реализация переупорядочивания записей

К сожалению, метод `save()` не выполняет переупорядочивание записей, и его придется реализовывать вручную.

Переупорядочивание записей достигается тем, что в каждой форме, составляющей набор, автоматически создается целочисленное поле с именем, заданным в переменной `ORDERING_FIELD_NAME` из модуля `django.forms.formsets`. В этом поле хранится целочисленный порядковый номер текущей записи в последовательности.

При выводе набора форм в такие поля будут подставлены порядковые номера записей, начиная с 1. Меняя эти номера, посетитель и переупорядочивает записи.

Чтобы сохранить заданный порядок записей, указанные для них значения порядковых номеров нужно куда-то записать. Для этого следует:

1. Добавить в модель поле целочисленного типа.
2. Указать порядок сортировки по значению этого поля (обычно по возрастанию — так будет логичнее).

Например, для переупорядочивания рубрик в модели `Rubric` можно предусмотреть поле `order`:

```
class Rubric(models.Model):
    . . .
    order = models.SmallIntegerField(default=0, db_index=True)
    . . .
    class Meta:
        . . .
        ordering = ['order', 'name']
```

Полный код контроллера-функции, обрабатывающего набор форм, позволяющего удалять и переупорядочивать записи, приведен в листинге 14.2. Обратим внимание, как порядковые номера записей сохраняются в поле `order` модели `Rubric`.

Листинг 14.2. Обработка набора форм, позволяющего переупорядочивать записи

```
from django.shortcuts import render, redirect
from django.forms import modelformset_factory
from django.forms.formsets import ORDERING_FIELD_NAME
from .models import Rubric

def rubrics(request):
    RubricFormSet = modelformset_factory(Rubric, fields=('name',),
                                         can_order=True, can_delete=True)
    if request.method == 'POST':
        formset = RubricFormSet(request.POST)
        if formset.is_valid():
            for form in formset:
                if form.cleaned_data:
                    rubric = form.save(commit=False)
                    rubric.order = form.cleaned_data[ORDERING_FIELD_NAME]
                    rubric.save()
            return redirect('bboard:index')
    else:
        formset = RubricFormSet()
    context = {'formset': formset}
    return render(request, 'bboard/rubrics.html', context)
```

14.3. Вывод наборов форм на экран

В целом вывод наборов форм на экран выполняется так же и теми же средствами, что и вывод обычных форм (см. *разд. 13.3*).

14.3.1. Быстрый вывод наборов форм

Быстрый вывод наборов форм выполняют три следующих метода:

- `as_p()` — вывод по абзацам. Надпись для элемента управления и сам элемент управления каждой формы из набора выводятся в отдельном абзаце и отделяются друг от друга пробелами. Пример использования:

```
<form method="post">
    {% csrf_token %}
    {{ formset.as_p }}
    <input type="submit" value="Сохранить">
</form>
```

- `as_ul()` — вывод в виде маркированного списка. Надпись и сам элемент управления выводятся в отдельном пункте списка и отделяются друг от друга пробелами. Теги `<ul>` и `</ul>` не формируются. Пример:

```
<form method="post">
    {% csrf_token %}
```

```
    <ul>
        {{ formset.as_ul }}
    </ul>
    <input type="submit" value="Сохранить">
</form>
```

- `as_table()` — вывод в виде таблицы с двумя столбцами: в левом выводятся надписи, в правом — элементы управления. Каждая пара "надпись—элемент управления" занимает отдельную строку таблицы. Теги `<table>` и `</table>` не выводятся. Пример:

```
<form method="post">
    {% csrf_token %}
    <table>
        {{ formset.as_table }}
    </table>
    <input type="submit" value="Сохранить">
</form>
```

  Можно просто указать переменную, хранящую набор форм, — метод `as_table()` будет вызван автоматически:

```
<form method="post">
    {% csrf_token %}
    <table>
        {{ formset }}
    </table>
    <input type="submit" value="Сохранить">
</form>
```

Парный тег `<form>`, создающий саму форму, в этом случае не создается, равно как и кнопка отправки данных. Также следует поместить в форму тег шаблонизатора `csrf_token`, который создаст скрытое поле с электронным жетоном безопасности.

14.3.2. Расширенный вывод наборов форм

Инструментов для расширенного вывода наборов форм Django предоставляет совсем немного.

Прежде всего, это атрибут `management_form`, поддерживаемый всеми классами наборов форм. Он хранит ссылку на служебную форму, входящую в состав набора и хранящую необходимые для работы служебные данные.

Далее, не забываем, что набор форм поддерживает функциональность итератора, возвращающего на каждом проходе очередную входящую в него форму.

И наконец, метод `non_form_errors()` набора форм возвращает список сообщений об ошибках, относящихся ко всему набору.

В листинге 14.3 приведен код шаблона bboard\rubrics.html, используемого контроллерами из листингов 14.1 и 14.2.

Листинг 14.3. Вывод набора форм в шаблоне расширенными средствами

```
<form method="post">
    {% csrf_token %}
    {{ formset.management_form }}
    {% if formset.non_form_errors %}
    <ul>
        {% for error in formset.non_form_errors %}
        <li><em>{{ error|escape }}</em></li>
        {% endfor %}
    </ul>
    {% endif %}
    {% for form in formset %}
        {% for hidden in form.hidden_fields %}
            {{ hidden }}
        {% endfor %}
        {% if form.non_field_errors %}
        <ul>
            {% for error in form.non_field_errors %}
            <li><em>{{ error|escape }}</em></li>
            {% endfor %}
        </ul>
        {% endif %}
        {% for field in form.visible_fields %}
            {% if field.errors %}
            <ul>
                {% for error in field.errors %}
                <li><em>{{ error|escape }}</em></li>
                {% endfor %}
            </ul>
            {% endif %}
            <p>{{ field.label_tag }}<br>{{ field }}<br>
            {{ field.help_text }}</p>
        {% endfor %}
    {% endfor %}
    <input type="submit" value="Сохранить">
</form>
```

14.4. Валидация в наборах форм

Валидацию удобнее всего реализовать:

- в модели, с которой связан набор форм;
- в связанной с этой моделью форме, на основе которой создается набор. Эта форма задается в параметре `form` функции `modelformset_factory()`.

В самом наборе форм имеет смысл выполнять валидацию лишь тогда, когда необходимо проверить весь массив данных, введенных в этот набор.

Валидация в наборе форм реализуется так:

1. Объявляется класс, производный от класса `BaseModelFormSet` из модуля `django.forms`.
2. В этом классе переопределяется метод `clean(self)`, в котором и выполняется валидация. Этот метод должен удовлетворять тем же требованиям, что и одноименный метод класса формы, связанной с моделью (см. *разд. 13.4.2*).
3. Создается набор форм — вызовом функции `modelformset_factory()`, в параметре `formset` которой указывается объявленный класс набора форм.

Атрибут `forms`, унаследованный от класса `BaseModelFormSet`, хранит последовательность всех форм, что имеются в наборе.

Сообщения об ошибках, генерируемые таким валидатором, будут присутствовать в списке ошибок, относящихся ко всему набору форм (возвращается методом `non_form_errors()`).

Вот пример кода, выполняющего валидацию на уровне набора форм и требующего обязательного присутствия рубрик "Недвижимость", "Транспорт" и "Мебель":

```
class RubricBaseFormSet(BaseModelFormSet):
    def clean(self):
        super().clean()
        names = [form.cleaned_data['name'] for form in self.forms \
                 if 'name' in form.cleaned_data]
        if ('Недвижимость' not in names) or ('Транспорт' not in names) \
           or ('Мебель' not in names):
            raise ValidationError(
                  'Добавьте рубрики недвижимости, транспорта и мебели')
. . .
def rubrics(request):
    RubricFormSet = modelformset_factory(Rubric, fields=('name',),
                    can_order=True, can_delete=True,
                    formset=RubricBaseFormSet)
    . . .
```

14.5. Встроенные наборы форм

Встроенные наборы форм служат для работы с записями вторичной модели, связанными с указанной записью первичной модели.

14.5.1. Создание встроенных наборов форм

Для создания встроенных наборов форм применяется функция `inlineformset_factory()` из модуля `django.forms`. Вот формат ее вызова:

```
inlineformset_factory(<первичная модель>, <вторичная модель>[,
                      form=<форма, связанная с моделью>][,
                      fk_name=None][, fields=None][, exclude=None][,
```

```
labels=None][, help_texts=None][,
error_messages=None][, field_classes=None][,
widgets=None][, extra=3][,
can_order=False][, can_delete=True][,
min_num=None][, validate_min=False][,
max_num=None][, validate_max=False][,
formset=<базовый набор форм, связанных с моделью>])
```

В первом позиционном параметре указывается ссылка на класс первичной модели, а во втором позиционном параметре — ссылка на класс вторичной модели.

Параметр `fk_name` задает имя поля внешнего ключа вторичной модели, по которому устанавливается связь с первичной моделью, в виде строки. Он указывается только в том случае, если во вторичной модели установлено более одной связи с первичной моделью и требуется выбрать, какую именно связь нужно использовать.

Остальные параметры имеют такое же назначение, что и у функции `modelformset_factory()` (см. *разд. 14.1*). Отметим только, что у параметра `extra` (задает количество "пустых" форм) значение по умолчанию 3, а у параметра `can_delete` (указывает, можно ли удалять связанные записи) — `True`. А базовый набор форм, задаваемый в параметре `formset`, должен быть производным от класса `BaseInlineFormSet` из модуля `django.forms`.

14.5.2. Обработка встроенных наборов форм

Обработка встроенных наборов форм выполняется так же, как и обычных наборов форм, связанных с моделями. Только при создании объекта набора форм как в первый, так и во второй раз нужно передать конструктору с параметром `instance` запись первичной модели. После этого набор форм выведет записи вторичной модели, связанные с ней.

В листинге 14.4 приведен код контроллера, который выводит на экран страницу со встроенным набором форм, который показывает все объявления, относящиеся к выбранной посетителем рубрике, позволяет править и удалять их. В нем используется форма `BbForm`, написанная в *главе 2* (см. листинг 2.6).

Листинг 14.4. Применение встроенного набора форм

```
from django.shortcuts import render, redirect
from django.forms import inlineformset_factory

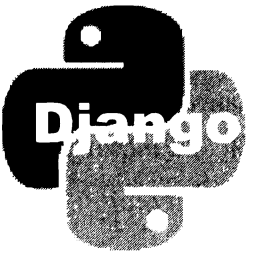
from .models import Bb, Rubric
from .forms import BbForm

def bbs(request, rubric_id):
    BbsFormSet = inlineformset_factory(Rubric, Bb, form=BbForm, extra=1)
    rubric = Rubric.objects.get(pk=rubric_id)
    if request.method == 'POST':
        formset = BbsFormSet(request.POST, instance=rubric)
```

```
        if formset.is_valid():
            formset.save()
            return redirect('bboard:index')
    else:
        formset = BbsFormSet(instance=rubric)
    context = {'formset': formset, 'current_rubric': rubric}
    return render(request, 'bboard/bbs.html', context)
```

Вывод встроенного набора форм на экран выполняется с применением тех же программных инструментов, что и вывод обычного набора форм, связанного с моделью (см. *разд. 14.3*).

И точно такими же средствами во встроенном наборе форм реализуется валидация. Единственное исключение: объявляемый для этого класс должен быть производным от класса `BaseInlineFormSet` из модуля `django.forms`.



ГЛАВА 15

Разграничение доступа: базовые инструменты

К внутренним данным сайта, хранящимся в его информационной базе, следует допускать только посетителей, записанных в особом списке — *зарегистрированных пользователей*, или просто *пользователей*. Также нужно учитывать, что какому-либо пользователю может быть запрещено работать с определенными данными — иначе говоря, принимать во внимание *права*, или *привилегии*, пользователя.

Допущением или недопущением посетителей к работе с внутренними данными сайта на основе того, зарегистрирован ли он в *списке пользователей*, и с учетом его прав, в Django-сайте занимается подсистема *разграничения доступа*.

15.1. Как работает подсистема разграничения доступа

Если посетитель желает получить доступ к внутренним данным сайта (например, чтобы добавить объявление или создать новую рубрику), то он предварительно должен войти на особую веб-страницу и занести в представленную там форму свои регистрационные данные: имя и пароль. Подсистема разграничения доступа проверит, имеется ли пользователь с такими именем и паролем в списке пользователей (т. е. является ли он зарегистрированным пользователем). Если такой пользователь в списке обнаружился, подсистема помечает его как выполнившего процедуру *аутентификации*, или *входа*, на сайт. В противном случае посетитель получит сообщение о том, что его в списке нет и данные сайта ему недоступны.

Когда посетитель пытается попасть на страницу для работы с внутренними данными сайта, подсистема разграничения доступа проверяет, выполнил ли он процедуру входа и имеет ли он права на работу с этими данными, — выполняет *авторизацию*. Если посетитель прошел авторизацию, то он допускается к странице, в противном случае получает соответствующее сообщение.

Закончив работу с внутренними данными сайта, пользователь выполняет процедуру *выхода* с сайта. При этом подсистема разграничения доступа помечает его как не

выполнившего вход. Теперь, чтобы снова получить доступ к внутренним данным сайта, посетитель вновь должен выполнить вход.

Но как посетители заносятся в список пользователей? Во-первых, их может добавить туда (выполнить их *регистрацию*) один из пользователей, имеющих права на работу с этим списком, — такое обычно практикуется в корпоративных решениях с ограниченным кругом пользователей. Во-вторых, посетитель сможет занести себя в список самостоятельно, зайдя на страницу регистрации и введя необходимые данные — в первую очередь, свои регистрационное имя и пароль. Этот способ применяется на общедоступных интернет-ресурсах.

На тот случай, если какой-либо из зарегистрированных пользователей забыл свой пароль, на сайтах часто предусматривают процедуру *восстановления пароля*. Забывчивый пользователь заходит на особую страницу и вводит свой адрес электронной почты. Подсистема разграничения доступа ищет в списке пользователя с таким адресом и отправляет ему особое письмо с гиперссылкой, ведущей на страницу, где пользователь сможет задать новый пароль.

15.2. Подготовка подсистемы разграничения доступа

15.2.1. Настройка подсистемы разграничения доступа

Настройки подсистемы разграничения доступа записываются, как обычно, в модуле settings.py пакета конфигурации.

Чтобы эта подсистема успешно работала, нужно сделать следующее:

- проверить, записаны ли в списке зарегистрированных в проекте приложений (параметр `INSTALLED_APPS`) приложения `django.contrib.auth` и `django.contrib.contenttypes`;
- проверить, записаны ли в списке зарегистрированных посредников (параметр `MIDDLEWARE`) посредники `django.contrib.sessions.middleware.SessionMiddleware` и `django.contrib.auth.middleware.AuthenticationMiddleware`.

Впрочем, во вновь созданном проекте все эти приложения и посредники уже занесены в соответствующие списки.

Кроме того, на работу подсистемы влияют следующие параметры:

- `LOGIN_URL` — интернет-адрес, на который будет выполнено перенаправление после попытки попасть на страницу, закрытую от неавторизованных посетителей (*гостей*). Также можно указать имя маршрута (см. *разд. 8.4*). Значение по умолчанию: `"/accounts/login/"`.

  Обычно в этом параметре указывается интернет-адрес страницы входа на сайт;
- `LOGIN_REDIRECT_URL` — интернет-адрес, на который будет выполнено перенаправление после успешного входа на сайт. Также можно указать имя маршрута. Значение по умолчанию: `"/accounts/profile/"`.

Если переход на страницу входа на сайт был вызван попыткой попасть на страницу, закрытую от неавторизованных посетителей, то Django автоматически выполнит перенаправление на страницу входа, добавив к ее интернет-адресу GET-параметр `next`, в котором запишет интернет-адрес страницы, на которую хотел попасть посетитель. После успешного входа будет выполнено перенаправление на интернет-адрес, сохраненный в этом GET-параметре. Если же такого параметра обнаружить не удалось, перенаправление выполнится по интернет-адресу из параметра `LOGIN_REDIRECT_URL`;

- `LOGOUT_REDIRECT_URL` — интернет-адрес, на который будет выполнено перенаправление после успешного выхода с сайта. Также можно указать имя маршрута или значение `None` — в этом случае будет выведена страница выхода с сайта. Значение по умолчанию — `None`;
- `PASSWORD_RESET_TIMEOUT_DAYS` — число дней, в течение которых будет действителен интернет-адрес сброса пароля, отправленный посетителю в электронном письме;
- `AUTHENTICATION_BACKENDS` — список путей к модулям бэкендов, выполняющих аутентификацию и авторизацию, которые задаются в виде строк (*бэкенд* — это серверная часть какой-либо программной подсистемы). Значение по умолчанию: `["django.contrib.auth.backends.ModelBackend"]` (единственный поставляемый в составе Django аутентификационный бэкенд, хранящий список пользователей в таблице базы данных).

Еще с несколькими параметрами, касающимися работы низкоуровневых механизмов аутентификации и авторизации, мы познакомимся в *главе 21*.

> ***Внимание!***
>
> Перед использованием подсистемы разграничения доступа требуется хотя бы раз выполнить миграции (за подробностями — к *разд. 5.3*). Это необходимо для того, чтобы Django создал в базе данных таблицы списков пользователей, групп и прав.

15.2.2. Создание суперпользователя

Суперпользователь — это зарегистрированный пользователь, имеющий права на работу со всеми данными сайта, включая список пользователей.

Для создания суперпользователя применяется команда `createsuperuser` утилиты manage.py:

```
manage.py createsuperuser [--username <имя суперпользователя>]
[--email <адрес электронной почты>]
```

После отдачи этой команды утилита manage.py запросит имя создаваемого пользователя, его адрес электронной почты и пароль, который потребуется ввести дважды.

Поддерживаются два необязательных ключа командной строки:

- `--username` — задает *имя* создаваемого суперпользователя. Если указан, то утилита manage.py не будет запрашивать имя;

- `--email` — задает *адрес электронной почты* суперпользователя. Если указан, то утилита manage.py не будет запрашивать этот адрес.

15.2.3. Смена пароля пользователя

В процессе разработки сайта может потребоваться сменить пароль у какого-либо из пользователей (вследствие забывчивости или по иной причине). Для такого случая утилита manage.py предусматривает команду `changepassword`:

```
manage.py changepassword [<имя пользователя>]
```

После ее отдачи будет выполнена смена пароля пользователя с указанным *именем* или, если таковое не указано, текущего пользователя (выполнившего вход на сайт в данный момент). Новый пароль следует ввести дважды — для надежности.

15.3. Работа со списками пользователей и групп

Административный веб-сайт Django предоставляет удобные средства для работы со списками пользователей и групп (разговор о них пойдет позже). Оба списка находятся в приложении **Пользователи и группы** на главной странице административного сайта (см. рис. 1.6).

15.3.1. Список пользователей

Для каждого пользователя из списка пользователей мы можем указать следующие сведения:

- имя, которое он будет вводить в соответствующее поле ввода в форме входа;
- пароль.

  При создании пользователя на странице будут присутствовать два поля для указания пароля. В эти поля нужно ввести один и тот же пароль (это сделано для надежности).

  При правке существующего пользователя вместо поля ввода пароля будет выведен сам пароль в закодированном виде. Сменить пароль можно, щелкнув на расположенной под закодированным паролем гиперссылке;
- настоящее имя (необязательно);
- настоящая фамилия (необязательно);
- адрес электронной почты;
- является ли пользователь активным. Только *активные* пользователи могут выполнять вход на сайт;
- имеет ли пользователь статус персонала. Только пользователи со статусом *персонала* имеют доступ к административному сайту Django. Однако для получения

доступа к страницам, не относящимся к административному сайту, в том числе закрытым для гостей, статус персонала не нужен;

- является ли пользователь суперпользователем;
- список прав, имеющихся у пользователя.

Для указания списка прав предусмотрен элемент управления в виде двух списков и четырех кнопок (рис. 15.1).

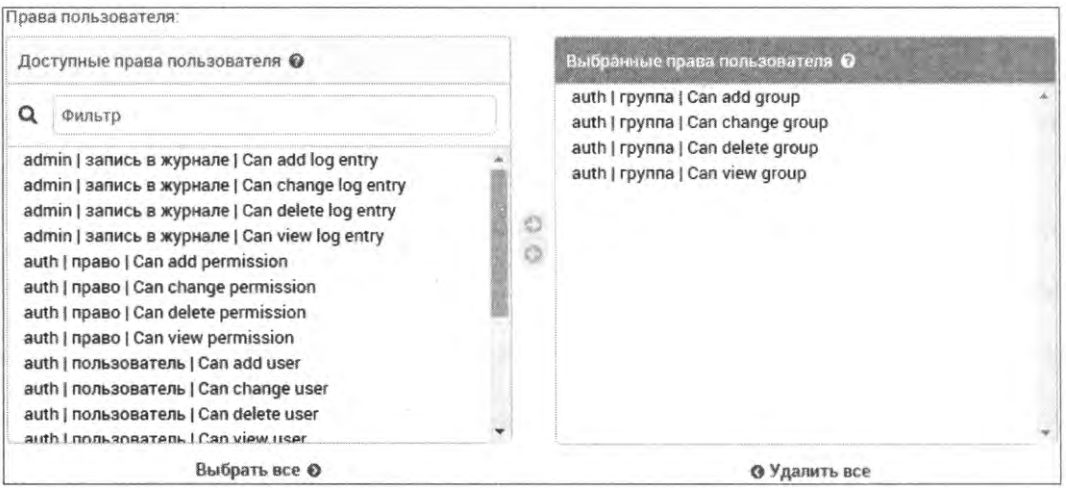


Рис. 15.1. Элемент управления для указания списка прав пользователя

В левом списке выводятся все доступные для пользователей права. Представляющие их пункты списка имеют следующий вид:

<приложение>* | *<модель>* | Can *<операция>* *<модель>

Приложение всегда выводится в виде своего псевдонима, *модель* — либо в виде имени класса, либо как заданное для него название (указывается в параметре `verbose_name` модели, описанном в *разд. 4.4*). *Операция* обозначается словом **view** (просмотр), **add** (добавление), **change** (правка) или **delete** (удаление).

Рассмотрим несколько примеров:

- **auth | пользователь | Can add user** — право добавлять пользователей (**auth** — это псевдоним приложения, реализующего систему разграничения доступа);
- **auth | группа | Can delete group** — право удалять группы пользователей;
- **bboard | Объявление | Can add bb** — право добавлять объявления;
- **bboard | Рубрика | Can change Рубрика** — право править рубрики.

В правом списке (см. рис. 15.1) показываются права, уже имеющиеся у пользователя. Выводятся они в точно таком же виде.

Оба списка (и левый, и правый) предоставляют возможность выбора произвольного числа пунктов:

- чтобы предоставить пользователю какие-либо права, следует выбрать их в левом списке и нажать расположенную между списками кнопку со стрелкой вправо;
- чтобы удалить права, данные пользователю ранее, нужно выбрать их в правом списке и нажать расположенную между списками кнопку со стрелкой влево;
- чтобы дать пользователю какое-либо одно право, достаточно найти его в левом списке и щелкнуть на нем двойным щелчком;
- чтобы удалить у пользователя какое-либо одно право, следует найти его в правом списке и щелкнуть на нем двойным щелчком;
- чтобы дать пользователю все доступные права, нужно нажать находящуюся под левым списком кнопку **Выбрать все**;
- чтобы удалить у пользователя все права, нужно нажать находящуюся под правым списком кнопку **Удалить все**.

> ***Внимание!***
>
> Пользователь может выполнять только те операции над внутренними данными сайта, на которые он явно получил права. Модели, на которые он не имеет никаких прав, при этом вообще не будут отображаться в административном сайте.
>
> Однако суперпользователь может выполнять любые операции над любыми моделями, независимо от того, какие права он имеет.

15.3.2. Группы пользователей. Список групп

На сайтах с большим числом зарегистрированных пользователей, выполняющих разные задачи, для быстрого указания прав у пользователей можно включать последних в *группы*. Каждая такая группа объединяет произвольное количество пользователей и задает для них одинаковый набор прав. Любой пользователь может входить в любое число групп.

Для каждой группы на Django-сайте указываются ее имя и список прав, которые будут иметь входящие в группу пользователи. Список прав для группы задается точно так же и с помощью точно такого же элемента управления, что и аналогичный параметр у отдельного пользователя (см. *разд. 15.3.1*).

Для указания групп, в которые входит пользователь, применяется такой же элемент управления.

> ***Внимание!***
>
> При проверке прав, предоставленных пользователю, принимаются в расчет как права, заданные непосредственно для него, так и права всех групп, в которые он входит.

> ***На заметку***
>
> Для своих нужд Django создает в базе данных таблицы `auth_user` (список пользователей), `auth_group` (список групп), `auth_permission` (список прав), `auth_user_groups` (связующая таблица, реализующая связь "многие-со-многими" между списками пользователей и групп), `auth_user_user_permissions` (связующая между списками пользователей и прав) и `auth_group_permissions` (связующая между списками групп и прав).

15.4. Аутентификация и служебные процедуры

Для выполнения аутентификации, т. е. входа на сайт, и различных служебных процедур (выхода, смены и сброса пароля) Django предлагает ряд контроллеров-классов, объявленных в модуле `django.contrib.auth.views`.

15.4.1. Контроллер *LoginView*: вход на сайт

Контроллер-класс `LoginView`, наследующий от `FormView` (см. *разд. 10.5.1.3*), реализует вход на сайт. При получении запроса по HTTP-методу GET он выводит на экран страницу входа с формой, в которую следует занести имя и пароль пользователя. При получении POST-запроса (т. е. после отправки формы) он ищет в списке пользователя с указанными именем и паролем. Если такой пользователь обнаружился, выполняется перенаправление по интернет-адресу, взятому из GET- или POST-параметра `next`, или, если такой параметр отсутствует, из параметра `LOGIN_REDIRECT_URL` настроек проекта. Если же подходящего пользователя не нашлось, то страница входа выводится повторно.

Класс `LoginView` поддерживает следующие атрибуты:

- `template_name` — путь к шаблону страницы входа в виде строки (по умолчанию: `"registration/login.html"`);
- `redirect_field_name` — имя GET- или POST-параметра, из которого будет извлекаться интернет-адрес для перенаправления после успешного входа, в виде строки (по умолчанию: `"next"`);
- `redirect_authenticated_user` — если `True`, то пользователи, уже выполнившие вход, при попытке попасть на страницу входа будут перенаправлены по интернет-адресу, взятому из GET- или POST-параметра `next` или параметра `LOGIN_REDIRECT_URL` настроек проекта. Если `False`, то пользователи, выполнившие вход, все же смогут попасть на страницу входа.

  Значение этого атрибута по умолчанию — `False`, и менять его на `True` следует с осторожностью, т. к. это может вызвать нежелательные эффекты. В частности, если пользователь попытается попасть на страницу с данными, для работы с которыми у него нет прав, он будет перенаправлен на страницу входа. Но если атрибут `redirect_authenticated_user` имеет значение `True`, то сразу же после этого будет выполнено перенаправление на страницу, с которой пользователь попал на страницу входа. В результате возникнет зацикливание, которое закончится аварийным завершением работы сайта;
- `extra_context` — дополнительное содержимое контекста шаблона. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст;
- `success_url_allowed_hosts` — множество, задающее хосты, на которые можно выполнить перенаправление после успешного входа, в дополнение к текущему хосту (по умолчанию — "пустое" множество);

- `authentication_form` — ссылка на класс формы входа (по умолчанию — класс `AuthenticationForm` из модуля `django.contrib.auth.forms`).

Контекст шаблона, создающий страницу входа, содержит следующие переменные:

- `form` — форма для ввода имени и пароля;
- `next` — интернет-адрес, на который будет выполнено перенаправление после успешного входа.

> ***Внимание!***
>
> Шаблон *registration\login.html* изначально не существует ни в одном из зарегистрированных в проекте приложений, включая встроенные во фреймворк. Поэтому мы можем просто создать такой шаблон в одном из своих приложений.
>
> Шаблоны, указанные по умолчанию во всех остальных контроллерах-классах, которые будут рассмотрены в этом разделе, уже существуют во встроенном приложении `django.contrib.admin`, реализующем работу административного сайта Django. Поэтому в остальных контроллерах следует задать другие имена шаблонов (в противном случае будут использоваться шаблоны административного сайта).

Чтобы реализовать процедуру аутентификации (входа), достаточно добавить в список маршрутов уровня проекта (он объявлен в модуле urls.py пакета конфигурации) такой элемент:

```
from django.contrib.auth.views import LoginView

urlpatterns = [
    . . .
    path('accounts/login/', LoginView.as_view(), name='login'),
]
```

Код простейшего шаблона страницы входа registration\login.html приведен в листинге 15.1.

Листинг 15.1. Код шаблона страницы входа

```
{% extends "layout/basic.html" %}

{% block title %}Вход{% endblock %}

{% block content %}
<h2>Вход</h2>
{% if user.is_authenticated %}
<p>Вы уже выполнили вход.</p>
{% else %}
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="hidden" name="next" value="{{ next }}">
    <input type="submit" value="Войти">
</form>
```

```
{% endif %}
{% endblock %}
```

Поскольку разработчики Django предостерегают от автоматического перенаправления со страницы входа пользователей, уже выполнивших вход, придется использовать другие средства предотвращения повторного входа на сайт. В контекст любого шаблона помещается переменная `user`, хранящая объект текущего пользователя. Атрибут `is_authenticated` этого объекта хранит `True`, если пользователь уже вошел на сайт, и `False` — если еще нет. С учетом этого можно вывести на странице либо форму входа, либо сообщение о том, что вход уже был выполнен.

Также в форме входа следует создать скрытое поле с именем `next` и занести в него интернет-адрес для перенаправления при успешном входе.

15.4.2. Контроллер *LogoutView*: выход с сайта

Контроллер-класс `LogoutView`, наследующий от `TemplateView` (см. *разд. 10.2.4*), реализует выход с сайта при получении GET-запроса, после чего осуществляет перенаправление на интернет-адрес, указанный в GET-параметре `next` или, если такового нет, в атрибуте `next_page`. Если значение этого атрибута равно `None`, то он выводит страницу с сообщением об успешном выходе.

Класс поддерживает атрибуты:

- `next_page` — интернет-адрес, на который будет выполнено перенаправление после успешного выхода с сайта (значение по умолчанию берется из параметра `LOGOUT_REDIRECT_URL` настроек сайта). Также можно указать имя нужного маршрута.

  Если задать этому атрибуту значение `None`, то перенаправление выполняться не станет, а вместо этого на экран будет выведена страница с сообщением об успешном выходе;

- `template_name` — путь к шаблону страницы сообщения об успешном выходе в виде строки (по умолчанию: `"registration/logged_out.html"`).

  Эта страница будет выведена, только если в текущем интернет-адресе отсутствует GET-параметр `next` и значение атрибута `next_page` равно `None`;

- `redirect_field_name` — имя GET-параметра, из которого будет извлекаться интернет-адрес для перенаправления после успешного выхода, в виде строки (значение по умолчанию: `"next"`);

- `extra_context` — дополнительное содержимое контекста шаблона. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст;

- `success_url_allowed_hosts` — множество, задающее хосты, на которые можно выполнить перенаправление после успешного выхода, в дополнение к текущему хосту (по умолчанию — "пустое" множество).

В контексте шаблона создается переменная `title`, в которой хранится сообщение об успешном выходе.

> ***Внимание!***
>
> Шаблоны, указанные по умолчанию в этом и во всех остальных контроллерах-классах, рассматриваемых в этом разделе, уже существуют во встроенном приложении `django.contrib.admin`, реализующем работу административного сайта Django. Поэтому придется указать для этих классов другие имена шаблонов (в противном случае будут использоваться шаблоны административного сайта).

Реализовать выход можно добавлением в список маршрутов уровня проекта элемента следующего вида:

```
from django.contrib.auth.views import LogoutView

urlpatterns = [
    . . .
    path('accounts/logout/',
         LogoutView.as_view(next_page='bboard:index'), name='logout'),
]
```

Интернет-адрес перенаправления или, как в нашем случае, имя маршрута также можно записать в настройках сайта:

```
path('accounts/logout/', LogoutView.as_view(), name='logout'),
. . .
LOGOUT_REDIRECT_URL = 'bboard:index'
```

15.4.3. Контроллер *PasswordChangeView*: смена пароля

Контроллер-класс `PasswordChangeView`, наследующий от класса `FormView`, выполняет смену пароля у текущего пользователя. При получении GET-запроса он выводит на экран страницу с формой, где нужно ввести старый пароль и, дважды, новый пароль. При получении POST-запроса он сохраняет введенный новый пароль и перенаправляет пользователя на страницу с сообщением об успешной смене пароля.

Вот атрибуты, поддерживаемые этим классом:

- `template_name` — путь к шаблону страницы с формой для смены пароля в виде строки (по умолчанию: `"registration/password_change_form.html"`);
- `success_url` — интернет-адрес, по которому будет выполнено перенаправление после успешной смены пароля (по умолчанию — по маршруту с именем `password_change_done`);
- `extra_context` — дополнительное содержимое контекста шаблона. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст;
- `form_class` — ссылка на класс формы для ввода нового пароля (по умолчанию — класс `PasswordChangeForm` из модуля `django.contrib.auth.forms`).

Контекст шаблона содержит переменные:

- `form` — форма для ввода нового пароля;
- `title` — текст вида `"Смена пароля"`, который можно использовать в заголовке страницы.

Реализовать смену пароля можно добавлением в список маршрутов уровня проекта такого элемента:

```
from django.contrib.auth.views import PasswordChangeView

urlpatterns = [
    . . .
    path('accounts/password_change/', PasswordChangeView.as_view(
                template_name='registration/change_password.html'),
        name='password_change'),
]
```

15.4.4. Контроллер *PasswordChangeDoneView*: уведомление об успешной смене пароля

Контроллер-класс `PasswordChangeDoneView`, наследующий от `TemplateView`, выводит страницу с уведомлением об успешной смене пароля.

Он поддерживает атрибуты:

- `template_name` — путь к шаблону страницы с уведомлением в виде строки (по умолчанию: `"registration/password_change_done.html"`);
- `extra_context` — дополнительное содержимое контекста шаблона. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст.

В контексте шаблона создается переменная `title`, в которой хранится текст уведомления.

Чтобы на сайте выводилось уведомление о смене пароля, достаточно добавить в список маршрутов уровня проекта такой элемент:

```
from django.contrib.auth.views import PasswordChangeDoneView

urlpatterns = [
    . . .
    path('accounts/password_change/done/',
        PasswordChangeDoneView.as_view(
                template_name='registration/password_changed.html'),
        name='password_change_done'),
]
```

15.4.5. Контроллер *PasswordResetView*: отправка письма для сброса пароля

Контроллер-класс `PasswordResetView`, производный от `FormView`, инициирует процедуру сброса пароля. При получении GET-запроса он выводит страницу с формой, в которую пользователю нужно занести свой адрес электронной почты. После получения POST-запроса он проверит существование этого адреса в списке пользователей и, если такой адрес есть, отправит по нему электронное письмо с гиперссылкой на страницу собственно сброса пароля.

Этот класс поддерживает следующие атрибуты:

- `template_name` — путь к шаблону страницы с формой для ввода адреса в виде строки (по умолчанию: `"registration/password_reset_form.html"`);
- `subject_template_name` — путь к шаблону темы электронного письма (по умолчанию: `"registration/password_reset_subject.txt"`);

  > ***Внимание!***
  > Шаблон темы электронного письма не должен содержать символов возврата каретки и перевода строки.

- `email_template_name` — путь к шаблону тела электронного письма в формате обычного текста (по умолчанию: `"registration/password_reset_email.html"`);
- `html_email_template_name` — путь к шаблону тела электронного письма в формате HTML. Если `None` (это значение по умолчанию), письмо в формате HTML отправляться не будет;
- `success_url` — интернет-адрес, по которому будет выполнено перенаправление после успешной отправки электронного письма (по умолчанию — по маршруту с именем `password_reset_done`);
- `from_email` — адрес электронной почты отправителя, который будет вставлен в отправляемое письмо (значение по умолчанию берется из параметра `DEFAULT_FROM_EMAIL` настроек проекта);
- `extra_context` — дополнительное содержимое контекста шаблона для страницы с формой. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст;
- `extra_email_context` — дополнительное содержимое контекста шаблона для электронного письма. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст;
- `form_class` — ссылка на класс формы для ввода адреса (по умолчанию — класс `PasswordResetForm` из модуля `django.contrib.auth.forms`);
- `token_generator` — экземпляр класса, выполняющего формирование электронного жетона безопасности, который будет включен в интернет-адрес страницы сброса пароля (по умолчанию — экземпляр класса `PasswordResetTokenGenerator` из модуля `django.contrib.auth.tokens`).

Контекст шаблона страницы содержит следующие переменные:

- `form` — форма для ввода адреса электронной почты;
- `title` — текст вида `"Сброс пароля"`, который можно использовать в заголовке страницы.

В контексте шаблона темы и тела электронного письма создаются переменные:

- `protocol` — обозначение протокола (`"http"` или `"https"`);
- `domain` — строка с комбинацией IP-адреса (или доменного имени, если его удастся определить) и номера TCP-порта, через который работает веб-сервер;
- `uid` — закодированный ключ пользователя;
- `token` — электронный жетон безопасности, выступающий в качестве электронной подписи;
- `email` — адрес электронной почты пользователя, по которому высылается это письмо;
- `user` — текущий пользователь, представленный экземпляром класса `User`.

Сброс пароля реализуется добавлением в список маршрутов уровня проекта таких элементов:

```
from django.contrib.auth.views import PasswordResetView

urlpatterns = [
    . . .
    path('accounts/password_reset/',
         PasswordResetView.as_view(
                template_name='registration/reset_password.html',
                subject_template_name='registration/reset_subject.txt',
                email_template_name='registration/reset_email.txt'),
         name='password_reset'),
]
```

Листинг 15.2 иллюстрирует код шаблона registration\reset_subject.txt, создающего тему электронного письма, а листинг 15.3 — код шаблона registration\reset_email.txt, который создаст тело письма.

Листинг 15.2. Код шаблона темы электронного письма с гиперссылкой для сброса пароля

```
{{ user.username }}: запрос на сброс пароля
```

Листинг 15.3. Код шаблона тела электронного письма с гиперссылкой для сброса пароля

```
{% autoescape off %}
Уважаемый {{ user.username }}!
Вы отправили запрос на сброс пароля. Чтобы выполнить сброс, пройдите
по этому интернет-адресу:
```

```
{{ protocol}}://{{ domain }}{% url 'password_reset_confirm' ↵
uidb64=uid token=token %}

До свидания!
С уважением, администрация сайта "Доска объявлений".
{% endautoescape %}
```

В листинге 15.3 интернет-адрес для перехода на страницу сброса пароля формируется путем обратного разрешения на основе маршрута с именем `password_reset_confirm`, который будет написан чуть позже.

15.4.6. Контроллер *PasswordResetDoneView*: уведомление об отправке письма для сброса пароля

Контроллер-класс `PasswordResetDoneView`, наследующий от `TemplateView`, выводит страницу с уведомлением об успешной отправке электронного письма для сброса пароля.

Он поддерживает атрибуты:

- `template_name` — путь к шаблону страницы с уведомлением в виде строки (по умолчанию: `"registration/password_reset_done.html"`);
- `extra_context` — дополнительное содержимое контекста шаблона. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст.

В контексте шаблона создается переменная `title`, в которой хранится текст уведомления.

Чтобы на сайте выводилось уведомление об отправке письма, нужно добавить в список маршрутов уровня проекта такой элемент:

```
from django.contrib.auth.views import PasswordResetDoneView

urlpatterns = [
    . . .
    path('accounts/password_reset/done/',
         PasswordResetDoneView.as_view(
                template_name='registration/email_sent.html'),
         name='password_reset_done'),
]
```

15.4.7. Контроллер *PasswordResetConfirmView*: собственно сброс пароля

Контроллер-класс `PasswordResetConfirmView`, наследующий от `FormView`, выполняет сброс пароля. Он запускается при переходе по интернет-адресу, отправленному в письме с сообщением о сбросе пароля. С URL-параметром `uidb64` он получает

закодированный ключ пользователя, а с URL-параметром `token` — электронный жетон безопасности, значения обоих параметров должны быть строковыми. Получив GET-запрос, он выводит страницу с формой для задания нового пароля, а после получения POST-запроса производит смену пароля и выполняет перенаправление на страницу с уведомлением об успешном сбросе пароля.

Вот атрибуты, поддерживаемые этим классом:

- `template_name` — путь к шаблону страницы с формой для задания нового пароля в виде строки (по умолчанию: `"registration/password_reset_confirm.html"`);
- `post_reset_login` — если `True`, то после успешного сброса пароля будет автоматически выполнен вход на сайт, если `False`, то этого не произойдет (по умолчанию — `False`);
- `success_url` — интернет-адрес, по которому будет выполнено перенаправление после успешной смены пароля (по умолчанию — по маршруту с именем `password_reset_complete`);
- `extra_context` — дополнительное содержимое контекста шаблона для страницы с формой. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст;
- `form_class` — ссылка на класс формы для сброса пароля (по умолчанию — класс `SetPasswordForm` из модуля `django.contrib.auth.forms`);
- `token_generator` — экземпляр класса, выполняющего формирование электронного жетона безопасности, который был включен в интернет-адрес, ведущий на страницу сброса пароля (по умолчанию — экземпляр класса `PasswordResetTokenGenerator` из модуля `django.contrib.auth.tokens`);
- `reset_url_token` (начиная с Django 3.0) — строковый фрагмент, который при выводе страницы с формой для задания нового пароля будет подставлен в интернет-адрес вместо электронного жетона (это делается для безопасности — чтобы никто не смог подсмотреть жетон и использовать его для атаки на сайт). По умолчанию: `"set-password"`.

Контекст шаблона содержит следующие переменные:

- `form` — форма для ввода нового пароля;
- `validlink` — если `True`, то интернет-адрес, по которому пользователь попал на эту страницу, действителен и еще ни разу не использовался, если `False`, то этот интернет-адрес скомпрометирован;
- `title` — текст вида `"Введите новый пароль"`, который можно использовать в заголовке страницы.

Вот такой элемент нужно добавить в список маршрутов уровня проекта, чтобы на сайте заработал сброс пароля:

```
from django.contrib.auth.views import PasswordResetConfirmView

urlpatterns = [
    . . .
```

```
    path('accounts/reset/<uidb64>/<token>/',
         PasswordResetConfirmView.as_view(
                template_name='registration/confirm_password.html'),
         name='password_reset_confirm'),
]
```

15.4.8. Контроллер *PasswordResetCompleteView*: уведомление об успешном сбросе пароля

Контроллер-класс `PasswordResetCompleteView`, наследующий от `TemplateView`, выводит страницу с уведомлением об успешном сбросе пароля.

Он поддерживает атрибуты:

- `template_name` — путь к шаблону страницы с уведомлением в виде строки (по умолчанию: `"registration/password_reset_complete.html"`);
- `extra_context` — дополнительное содержимое контекста шаблона. Его значение должно представлять собой словарь, элементы которого будут добавлены в контекст.

В контексте шаблона создается переменная `title`, в которой хранится текст уведомления.

Чтобы на сайте выводилось уведомление об успешном сбросе пароля, следует добавить в список маршрутов уровня проекта такой элемент:

```
from django.contrib.auth.views import PasswordResetCompleteView

urlpatterns = [
    . . .
    path('accounts/reset/done/',
         PasswordResetCompleteView.as_view(
                template_name='registration/password_confirmed.html'),
         name='password_reset_complete'),
]
```

15.5. Получение сведений о пользователях

15.5.1. Получение сведений о текущем пользователе

Любой зарегистрированный пользователь представляется в Django экземпляром класса `User` из модуля `django.contrib.auth.models`. Этот класс является моделью.

Доступ к экземпляру класса `User`, представляющему текущего пользователя, можно получить:

- в контроллере — из атрибута `user` объекта текущего запроса (экземпляра класса `Request`), который передается контроллеру-функции и методам контроллера-класса в первом параметре, обычно имеющем имя `request`.

Пример:

```
def index(request):
    # Проверяем, выполнил ли текущий пользователь вход
    if request.user.is_authenticated:
        . . .
```

- в шаблоне — из переменной контекста `user`, создаваемой обработчиком контекста `django.contrib.auth.context_processors.auth`:

```
{% if user.is_authenticated %}
<p>Вы уже выполнили вход.</p>
{% endif %}
```

Класс `User` поддерживает довольно большой набор полей, атрибутов и методов. Начнем знакомство с полей:

- `username` — регистрационное имя пользователя, обязательно к заполнению;
- `password` — пароль в закодированном виде, обязательно к заполнению;
- `email` — адрес электронной почты;
- `first_name` — настоящее имя пользователя;
- `last_name` — настоящая фамилия пользователя;
- `is_active` — `True`, если пользователь является активным, и `False` — в противном случае;
- `is_staff` — `True`, если пользователь имеет статус персонала, и `False` — в противном случае;
- `is_superuser` — `True`, если пользователь является суперпользователем, и `False` — в противном случае;
- `groups` — группы, в которые входит пользователь. Хранит диспетчер обратной связи, дающий доступ к записям связанной модели `Group` из модуля `django.contrib.auth.models`, в которой хранятся все группы;
- `last_login` — дата и время последнего входа на сайт;
- `date_joined` — дата и время регистрации пользователя на сайте.

Полезных атрибутов класс `User` поддерживает всего два:

- `is_authenticated` — `True`, если текущий пользователь выполнил вход, и `False`, если не выполнил (т. е. является гостем);
- `is_anonymous` — `True`, если текущий пользователь не выполнил вход на сайт (т. е. является гостем), и `False`, если выполнил.

Теперь рассмотрим методы этого класса:

- `has_perm(`*<право>*`[, obj=None])` — возвращает `True`, если текущий пользователь имеет указанное *право*, и `False` — в противном случае. *Право* задается в виде строки формата "*<приложение>*.*<операция>*_*<модель>*", где *приложение* указывается его псевдонимом, *операция* — строкой "view" (просмотр), "add" (добавление), "change" (правка) или "delete" (удаление), а *модель* — именем ее класса.

Пример:

```
def index(request):
    # Проверяем, имеет ли текущий пользователь право добавлять рубрики
    if request.user.has_perm('bboard.add_rubric'):
        . . .
```

Если в параметре `obj` указана запись модели, то будут проверяться права пользователя на эту запись, а не на саму модель. Эта возможность поддерживается не всеми бэкендами аутентификации (так, стандартный бэкенд `django.contrib.auth.backends.ModelBackend` ее не поддерживает).

Если пользователь неактивен, метод всегда возвращает `False`;

- `has_perms(`*<последовательность прав>*`[, obj=None])` — то же самое, что и `has_perm()`, но возвращает `True` только в том случае, если текущий пользователь имеет все права из заданной *последовательности*:

```
def index(request):
    # Проверяем, имеет ли текущий пользователь права добавлять,
    # править и удалять рубрики
    if request.user.has_perms(('bboard.add_rubric',
                    'bboard.change_rubric', 'bboard.delete_rubric')):
        . . .
```

- `get_user_permissions([obj=None])` (начиная с Django 3.0) — возвращает множество из прав, которыми непосредственно обладает текущий пользователь. Наименования прав представлены строками.

  Если в параметре `obj` указана запись модели, то метод вернет права на эту запись, а не на саму модель (поддерживается не всеми бэкендами аутентификации).

  Пример:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.get(username='someuser')
>>> user.get_user_permissions()
{'bboard.delete_rubric', 'bboard.view_rubric', 'bboard.add_rubric',
'bboard.change_rubric'}
```

- `get_group_permissions([obj=None])` — возвращает множество из прав групп, в которые входит текущий пользователь. Наименования прав представлены строками.

  Если в параметре `obj` указана запись модели, то метод вернет права на эту запись, а не на саму модель (поддерживается не всеми бэкендами аутентификации).

  Пример:

```
>>> user.get_group_permissions()
{'bboard.delete_bb', 'bboard.add_bb', 'bboard.change_bb',
'bboard.view_bb'}
```

- `get_all_permissions([obj=None])` — возвращает множество из прав, принадлежащих как непосредственно текущему пользователю, так и группами, в которые он входит. Наименования прав представлены строками.

  Если в параметре `obj` указана запись модели, метод вернет права на эту запись, а не на саму модель (поддерживается не всеми бэкендами аутентификации).

  Пример:

```
>>> user.get_all_permissions()
{'bboard.view_bb', 'bboard.change_rubric', 'bboard.delete_rubric',
'bboard.delete_bb', 'bboard.add_rubric', 'bboard.change_bb',
'bboard.add_bb', 'bboard.view_rubric'}
```

- `get_username()` — возвращает регистрационное имя пользователя;
- `get_full_name()` — возвращает строку, составленную из настоящих имени и фамилии пользователя, которые разделены пробелом;
- `get_short_name()` — возвращает настоящее имя пользователя.

Посетитель-гость представляется экземпляром класса `AnonymousUser` из того же модуля `django.contrib.auth.models`. Этот класс полностью аналогичен классу `User`, поддерживает те же поля, атрибуты и методы. Разумеется, гость не может выполнить вход и не имеет никаких прав.

15.5.2. Получение пользователей, обладающих заданным правом

Начиная с Django 3.0, диспетчер записей модели `User` поддерживает метод `with_perms`, возвращающий перечень зарегистрированных пользователей, которые имеют заданное *право*:

```
with_perm(<право>[, is_active=True][, include_superusers=True][,
                                    backend=None][, obj=None])
```

Право указывается в том же формате, который применяется в методах `has_perm()` и `has_perms()` (см. *разд. 15.5.1*).

Если параметру `is_active` задано значение `True`, то будут возвращены подходящие пользователи только из числа активных (поведение по умолчанию), если `False` — только из числа неактивных, если `None` — из числа как активных, так и неактивных.

Если параметру `include_superusers` присвоить `True`, то возвращенный перечень будет включать суперпользователей (поведение по умолчанию), если `False` — не будет включать.

Параметр `backend` задает аутентификационный бэкенд, используемый для поиска пользователей, обязательно из числа перечисленных в параметре `AUTHENTICATION_BACKENDS` настроек проекта. Если параметру `backend` задать значение `None`, то поиск подходящих пользователей будет выполняться с применением всех зарегистрированных в проекте бэкендов.

Если в параметре `obj` указана запись модели, то метод будет искать пользователей, обладающих правом на работу с этой записью, а не моделью целиком (поддерживается не всеми бэкендами аутентификации).

Метод `with_perm()` возвращает перечень пользователей, обладающих указанным *правом*, в виде обычного набора записей (экземпляра класса `QuerySet`).

Примеры:

```
>>> User.objects.with_perm('bboard.add_bb')
<QuerySet [<User: admin>, <User: someuser>]>
>>> User.objects.with_perm('bboard.add_user')
<QuerySet [<User: admin>]>
>>> User.objects.with_perm('bboard.add_bb', include_superusers=False)
<QuerySet [<User: someuser>]>
```

15.6. Авторизация

Авторизацию можно выполнить как в коде контроллеров (например, чтобы не пустить гостя на закрытую от него страницу), так и в шаблонах (чтобы скрыть гиперссылку, ведущую на закрытую для гостей страницу).

15.6.1. Авторизация в контроллерах

15.6.1.1. Авторизация в контроллерах-функциях: непосредственные проверки

В коде контроллера, применив описанные в *разд. 15.5.1* инструменты, мы можем непосредственно проверить, выполнил ли пользователь вход и имеет ли он достаточные права. И, на основе результатов проверок, пустить или не пустить пользователя на страницу.

Пример проверки, выполнил ли пользователь вход, с отправкой в противном случае сообщения об ошибке 403 (доступ к странице запрещен):

```
from django.http import HttpResponseForbidden

def rubrics(request):
    if request.user.is_authenticated:
        # Все в порядке: пользователь выполнил вход
        . . .
    else:
        return HttpResponseForbidden(
               'Вы не имеете допуска к списку рубрик')
```

Вместо отправки сообщения об ошибке можно перенаправлять посетителя на страницу входа:

```
def rubrics(request):
    if request.user.has_perms(('bboard.add_rubric',
                    'bboard.change_rubric', 'bboard.delete_rubric')):
```

```
        # Все в порядке: пользователь выполнил вход
        . . .
    else:
        return redirect('login')
```

Функция `redirect_to_login()` из модуля `django.contrib.auth.views` отправляет посетителя на страницу входа, а после выполнения входа выполняет перенаправление по заданному *интернет-адресу*:

```
redirect_to_login(<интернет-адрес перенаправления>[,
                  redirect_field_name='next'][, login_url=None])
```

Интернет-адрес перенаправления задается в виде строки. Необязательный параметр `redirect_field_name` указывает имя GET-параметра, передающего странице входа заданный *интернет-адрес* (по умолчанию: `"next"`). Параметр `login_url` задает интернет-адрес страницы входа или имя указывающего на нее маршрута (по умолчанию — значение параметра `LOGIN_URL` настроек проекта).

Функция `redirect_to_login()` возвращает объект ответа, выполняющий перенаправление. Этот объект нужно вернуть из контроллера-функции.

Пример:

```
from django.contrib.auth.views import redirect_to_login
def rubrics(request):
    if request.user.is_authenticated:
        . . .
    else:
        return redirect_to_login(reverse('bboard:rubrics'))
```

15.6.1.2. Авторизация в контроллерах-функциях: применение декораторов

Во многих случаях для авторизации удобнее применять декораторы, объявленные в модуле `django.contrib.auth.decorators`. Они указываются у контроллера, выводящего страницу с ограниченным доступом. Всего этих декораторов три:

- `login_required([redirect_field_name='next'][,][login_url=None])` — допускает к странице только пользователей, выполнивших вход. Если пользователь не выполнил вход, то выполняет перенаправление по интернет-адресу из параметра `LOGIN_URL` настроек проекта с передачей через GET-параметр `next` текущего интернет-адреса. Пример:

  ```
  from django.contrib.auth.decorators import login_required
  @login_required
  def rubrics(request):
      . . .
  ```

  Параметр `redirect_field_name` позволяет указать другое имя для GET-параметра, передающего текущий интернет-адрес, а параметр `login_url` — другой адрес страницы входа или другое имя указывающего на нее маршрута. Пример:

```
@login_required(login_url='/login/')
def rubrics(request):
    . . .
```

- `user_passes_test(<проверочная функция>[, redirect_field_name='next'][, login_url=None])` — допускает к странице только тех пользователей, кто выполнил вход и в чьем отношении *проверочная функция* вернет в качестве результата значение `True`. *Проверочная функция* должна принимать в качестве единственного параметра экземпляр класса `User`, представляющий текущего пользователя. Вот пример кода, допускающего к списку рубрик только пользователей, имеющих статус персонала:

```
from django.contrib.auth.decorators import user_passes_test
@user_passes_test(lambda user: user.is_staff)
def rubrics(request):
    . . .
```

Параметр `redirect_field_name` позволяет указать другое имя для GET-параметра, передающего текущий интернет-адрес страницы, а параметр `login_url` — другой интернет-адрес страницы входа или другое имя указывающего на нее маршрута;

- `permission_required(<права>[, raise_exception=False][, login_url=None])` — допускает к странице только пользователей, имеющих заданные *права*. *Права* указываются в том же формате, который применяется в методах `has_perm()` и `has_perms()` (см. *разд. 15.5.1*). Можно указать:

  - одно право:

```
from django.contrib.auth.decorators import permission_required
@permission_required('bboard.view_rubric')
def rubrics(request):
    . . .
```

  - последовательность из произвольного количества прав. В этом случае у текущего пользователя должны иметься все перечисленные в последовательности права. Пример:

```
@permission_required(('bboard.add_rubric',
            'bboard.change_rubric', 'bboard.delete_rubric'))
def rubrics(request):
    . . .
```

Параметр `login_url` позволит задать другой интернет-адрес страницы входа или другое имя маршрута, указывающего на нее.

Если параметру `raise_exception` присвоить значение `True`, то декоратор вместо перенаправления пользователей, не выполнивших вход, на страницу входа будет возбуждать исключение `PermissionDenied`, тем самым выводя страницу с сообщением об ошибке 403. Если это сообщение нужно выводить только пользователям, выполнившим вход и не имеющим необходимых прав, то следует применить декоратор `permission_required()` вместе с декоратором `login_required()`:

```
@login_required
@permission_required(('bboard.add_rubric',
                      'bboard.change_rubric', 'bboard.delete_rubric'))
def rubrics(request):
    . . .
```

В этом случае пользователи, не выполнившие вход, попадут на страницу входа, а те из них, кто выполнил вход, но не имеет достаточных прав, получат сообщение об ошибке 403.

15.6.1.3. Авторизация в контроллерах-классах

Реализовать авторизацию в контроллерах-классах можно посредством классов-примесей, объявленных в модуле `django.contrib.auth.mixins`. Они указываются в числе базовых в объявлении производных контроллеров-классов, причем в списках базовых классов примеси должны стоять первыми.

Класс `AccessMixin` — базовый для остальных классов-примесей. Он поддерживает ряд атрибутов и методов, предназначенных для указания важных параметров авторизации:

- `login_url` — атрибут, задает интернет-адрес или имя маршрута страницы входа (по умолчанию — `None`);
- `get_login_url(self)` — метод, должен возвращать интернет-адрес или имя маршрута страницы входа. В изначальной реализации возвращает значение атрибута `login_url` или, если оно равно `None`, значение параметра `LOGIN_URL` настроек проекта;
- `permission_denied_message` — атрибут, хранит строковое сообщение о возникшей ошибке (по умолчанию — "пустая" строка);
- `get_permission_denied_message(self)` — метод, должен возвращать сообщение об ошибке. В изначальной реализации возвращает значение атрибута `permission_denied_message`;
- `redirect_field_name` — атрибут, указывает имя GET-параметра, передающего интернет-адрес страницы с ограниченным доступом, на которую пытался попасть посетитель (по умолчанию: `"next"`);
- `get_redirect_field_name(self)` — метод, должен возвращать имя GET-параметра, передающего интернет-адрес страницы, на которую пытался попасть посетитель. В изначальной реализации возвращает значение атрибута `redirect_field_name`;
- `raise_exception` — атрибут. Если его значение равно `True`, то при попытке попасть на страницу гость или пользователь с недостаточными правами получит сообщение об ошибке 403. Если значение параметра равно `False`, то посетитель будет перенаправлен на страницу входа. Значение по умолчанию — `False`;
- `has_no_permission(self)` — метод, вызывается в том случае, если текущий пользователь не выполнил вход или не имеет необходимых прав, и на это нужно как-то отреагировать.

В изначальной реализации, если значение атрибута `raise_exception` равно `True`, возбуждает исключение `PermissionDenied` с сообщением, возвращенным методом `get_permission_denied_message()`. Если же значение атрибута `raise_exception` равно `False`, то выполняет перенаправление по интернет-адресу, возвращенному методом `get_login_url()`.

Рассмотрим классы-примеси, производные от `AccessMixin`:

- `LoginRequiredMixin` — допускает к странице только пользователей, выполнивших вход.

  Разрешаем добавлять новые объявления только пользователям, выполнившим вход:

  ```
  from django.contrib.auth.mixins import LoginRequiredMixin
  class BbCreateView(LoginRequiredMixin, CreateView):
      . . .
  ```

- `UserPassesTestMixin` — допускает к странице только тех пользователей, кто выполнил вход и в чьем отношении переопределенный метод `test_func(self)` вернет в качестве результата значение `True` (в изначальной реализации метод `test_func()` возбуждает исключение `NotImplementedError`, поэтому его обязательно следует переопределить).

  Разрешаем создавать новые объявления только пользователям со статусом персонала:

  ```
  from django.contrib.auth.mixins import UserPassesTestMixin
  class BbCreateView(UserPassesTestMixin, CreateView):
      . . .
      def test_func(self):
          return self.request.user.is_staff
  ```

- `PermissionRequiredMixin` — допускает к странице только пользователей, имеющих заданные права. Класс поддерживает дополнительные атрибут и методы:

  - `permission_required` — атрибут, задает требуемые права, которые указываются в том же формате, который применяется в методах `has_perm()` и `has_perms()` (см. *разд. 15.5.1*). Можно задать одно право или последовательность прав;
  - `get_permission_required(self)` — метод, должен возвращать требуемые права. В изначальной реализации возвращает значение атрибута `permission_required`;
  - `has_permission(self)` — метод, должен возвращать `True`, если текущий пользователь имеет заданные права, и `False` — в противном случае. В изначальной реализации возвращает результат вызова метода `has_perms()` у текущего пользователя.

  Разрешаем создавать новые объявления только пользователям с правами на создание, правку и удаление записей модели `Bb`:

```
from django.contrib.auth.mixins import PermissionRequiredMixin
class BbCreateView(PermissionRequiredMixin, CreateView):
    permission_required = ('bboard.add_bb', 'bboard.change_bb',
                           'bboard.delete_bb')
    . . .
```

15.6.2. Авторизация в шаблонах

Если в числе активных обработчиков контекста, указанных в параметре `context_processors` настроек шаблонизатора, имеется `django.contrib.auth.context_processors.auth` (подробности — в *разд. 11.1*), то он будет добавлять в контекст каждого шаблона переменные `user` и `perms`, хранящие соответственно текущего пользователя и его права.

Переменная `user` хранит экземпляр класса `User`, и из него можно извлечь сведения о текущем пользователе.

В следующем примере устанавливается, выполнил ли пользователь вход на сайт, и если выполнил, то выводится его имя:

```
{% if user.is_authenticated %}
<p>Добро пожаловать, {{ user.username }}!</p>
{% endif %}
```

Переменная `perms` хранит особый объект, который можно использовать для выяснения прав пользователя, причем двумя способами:

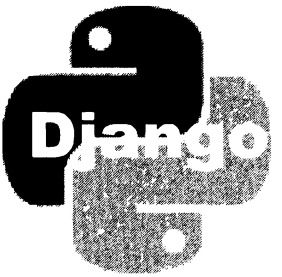
- первый способ — применением оператора `in` или `not in`. Права записываются в виде строки в том же формате, который применяется в вызовах методов `has_perm()` и `has_perms()` (см. *разд. 15.5.1*).

  В следующем примере выполняется проверка, имеет ли пользователь право на добавление объявлений, и если имеет, то выводится гиперссылка на страницу добавления объявления:

  ```
  {% if 'bboard.add_bb' in perms %}
  <a href="{% url 'bboard:add' %}">Добавить</a>
  {% endif %}
  ```

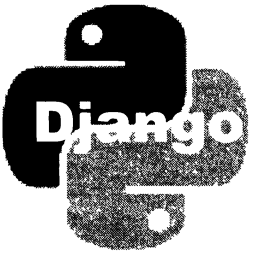
- второй способ — доступ к атрибуту с именем вида *<приложение>.<операция>_<модель>*. Такой атрибут хранит значение `True`, если пользователь имеет право на выполнение заданной *операции* в заданной *модели* указанного *приложения*, и `False` — в противном случае. Пример:

  ```
  {% if perms.bboard.add_bb %}
  <a href="{% url 'bboard:add' %}">Добавить</a>
  {% endif %}
  ```



ЧАСТЬ III

Расширенные инструменты и дополнительные библиотеки



ГЛАВА 16

Модели: расширенные инструменты

Модели Django предоставляют ряд расширенных инструментов: средства для управления выборкой полей, связи с дополнительными параметрами, полиморфные связи, наследование моделей, объявление своих диспетчеров записей и наборов записей и инструменты для управления транзакциями.

16.1. Управление выборкой полей

При выборке набора записей Django извлекает из таблицы значения полей только текущей модели. При обращении к полю связанной модели фреймворк выполняет дополнительный SQL-запрос для извлечения содержимого этого поля, что может снизить производительность.

Помимо этого, выполняется выборка значений из всех полей текущей модели. Если какие-то поля хранят данные большого объема (например, большой текст), их выборка займет много времени и отнимет существенный объем оперативной памяти.

Далее приведены методы, поддерживаемые диспетчером записей (классом `Manager`) и набором записей (классом `QuerySet`), которые позволяют управлять выборкой значений полей:

- `select_related(`*<поле внешнего ключа 1>, <поле внешнего ключа 2> . . . <поле внешнего ключа n>*`)` — будучи вызван у записи вторичной модели, указывает извлечь связанную запись первичной модели. В качестве параметров записываются имена *полей внешних ключей*, устанавливающих связь с нужными первичными моделями.

  Метод извлекает единичную связанную запись. Его можно применять только в моделях, связанных связью "один-с-одним", и вторичных моделях в случае связи "один-со-многими".

  Пример:

```
>>> from bboard.models import Bb
>>> b = Bb.objects.get(pk=1)
```

```
>>> # Никаких дополнительных запросов к базе данных не выполняется,
>>> # т. к. значение поля title, равно как и значения всех прочих
>>> # полей текущей модели, уже извлечены
>>> b.title
'Дача'
>>> # Но для извлечения полей записи связанной модели выполняется
>>> # отдельный запрос к базе данных
>>> b.rubric.name
'Недвижимость'
>>> # Используем метод select_related(), чтобы выбрать поля и текущей,
>>> # и связанной моделей в одном запросе
>>> b = Bb.objects.select_related('rubric').get(pk=1)
>>> # Теперь отдельный запрос к базе для извлечения значения поля
>>> # связанной модели не выполняется
>>> b.rubric.name
'Недвижимость'
```

Применяя синтаксис, описанный в *разд. 7.3.7*, можно выполнить выборку модели, связанной с первичной моделью. Предположим, что модель `Bb` связана с моделью `Rubric`, которая, в свою очередь, связана с моделью `SuperRubric` через поле внешнего ключа `super_rubric` и является вторичной для этой модели. Тогда мы можем выполнить выборку связанной записи модели `SuperRubric`, написав выражение вида:

```
>>> b = Bb.objects.select_related('rubric__super_rubric').get(pk=1)
```

Можно выполнять выборку сразу нескольких связанных моделей, написав такой код:

```
>>> b = Bb.objects.select_related('rubric',
                                  'rubric__super_rubric').get(pk=1)
```

или такой:

```
>>> b = Bb.objects.select_related('rubric').select_related(
                                  'rubric__super_rubric').get(pk=1)
```

Чтобы отменить выборку связанных записей, заданную предыдущими вызовами метода `select_related()`, достаточно вызвать этот метод, передав в качестве параметра значение `None`;

- `prefetch_related(`*<связь 1>*`, `*<связь 2>*` . . . `*<связь n>*`)` — будучи вызван у записи первичной модели, указывает извлечь все связанные записи вторичной модели.

Метод извлекает набор связанных записей. Он применяется в моделях, связанных связью "многие-со-многими", и первичных моделях в случае связи "один-со-многими".

В качестве *связи* можно указать:

- строку с именем:
  - атрибута, применяющегося для извлечения связанных записей (см. *разд. 7.2*) — если метод вызывается у записи первичной модели, и установлена связь "один-со-многими":

```
>>> from bboard.models import Rubric
>>> r = Rubric.objects.first()
>>> r
<Rubric: Бытовая техника>
>>> # Здесь для извлечения каждого объявления, связанного
>>> # с рубрикой, выполняется отдельный запрос к базе данных
>>> for bb in r.bb_set.all(): print(bb.title, end=' ')
...
Пылесос Стиральная машина
>>> # Используем метод prefetch_related(), чтобы извлечь все
>>> # связанные объявления в одном запросе
>>> r = Rubric.objects.prefetch_related('bb_set').first()
>>> for bb in r.bb_set.all(): print(bb.title, end=' ')
...
Пылесос Стиральная машина
```

- поля внешнего ключа — если установлена связь "многие-со-многими":

```
>>> from testapp.models import Machine, Spare
>>> # Указываем предварительно извлечь все связанные
>>> # с машиной составные части
>>> m = Machine.objects.prefetch_related('spares').first()
>>> for s in m.spares.all(): print(s.name, end=' ')
...
Гайка Винт
```

- экземпляр класса `Prefetch` из модуля `django.db.models`, хранящий все необходимые сведения для выборки записей. Конструктор этого класса вызывается в формате:

```
Prefetch(<связь>[, queryset=None][, to_attr=None])
```

Связь указывается точно так же, как было описано ранее.

Необязательный параметр `queryset` задает набор записей для выборки связанных записей. В этом наборе записей можно указать какую-либо фильтрацию, сортировку или предварительную выборку полей связанных записей методом `select_related()` (см. ранее).

Необязательный параметр `to_attr` позволяет указать имя атрибута, который будет создан в объекте текущей записи и сохранит набор выбранных записей связанной модели.

Примеры:

```
>>> from django.db.models import Prefetch
>>> # Выполняем выборку объявлений, связанных с рубрикой,
>>> # с одновременной их сортировкой по убыванию названия
>>> pr1 = Prefetch('bb_set', queryset=Bb.objects.order_by('-title'))
>>> r = Rubric.objects.prefetch_related(pr1).first()
>>> for bb in r.bb_set.all(): print(bb.title, end=' ')
...
```

```
Стиральная машина Пылесос
>>> # Выполняем выборку только тех связанных объявлений,
>>> # в которых указана цена свыше 1 000 руб., и помещаем
>>> # получившийся набор записей в атрибут expensive
>>> # объекта рубрики
>>> pr2 = Prefetch('bb_set',
                   queryset=Bb.objects.filter(price__gt=1000),
                   to_attr='expensive')
>>> r = Rubric.objects.prefetch_related(pr2).first()
>>> for bb in r.expensive: print(bb.title, end=' ')
...
Стиральная машина
```

Можно выполнить выборку наборов записей по нескольким связям, записав их либо в одном, либо в нескольких вызовах метода `prefetch_related()`. Чтобы отменить выборку наборов записей, заданную предыдущими вызовами метода `prefetch_related()`, следует вызвать этот метод, передав ему в качестве параметра значение `None`;

- `defer(`*<имя поля 1>, <имя поля 2> . . . <имя поля n>*`)` — указывает не извлекать значения полей с заданными *именами* в текущем запросе. Для последующего извлечения значений этих полей будет выполнен отдельный запрос к базе данных. Пример:

```
>>> bb = Bb.objects.defer('content').get(pk=3)
>>> # Никаких дополнительных запросов к базе данных не выполняется,
>>> # поскольку значение поля title было извлечено в текущем запросе
>>> bb.title
'Дом'
>>> # А значение поля content будет извлечено в отдельном запросе,
>>> # т. к. это поле было указано в вызове метода defer()
>>> bb.content
'Трехэтажный, кирпич'
```

Можно указать не выполнять выборку значений сразу у нескольких полей, записав их либо в одном, либо в нескольких вызовах метода `defer()`. Чтобы отменить запрет выборки, заданный предыдущими вызовами метода `defer()`, следует вызвать этот метод, передав ему в качестве параметра значение `None`;

- `only(`*<имя поля 1>, <имя поля 2> . . . <имя поля n>*`)` — указывает не извлекать значения всех полей, кроме полей с заданными *именами*, в текущем запросе. Для последующего извлечения значений полей, не указанных в вызове метода, будет выполнен отдельный запрос к базе данных. Пример:

```
>>> bb = Bb.objects.only('title', 'price').get(pk=3)
```

Вызов метода `only()` отменяет параметры выборки, заданные предыдущими вызовами методов `only()` и `defer()`. Однако после его вызова можно поставить вызов метода `defer()` — он укажет поля, которые не должны выбираться в текущем запросе.

16.2. Связи "многие-со-многими" с дополнительными данными

В *главе 4* мы написали модели `Machine` (машина) и `Spare` (отдельная деталь), связанные связью "многие-со-многими". Однако совершенно забыли о том, что машина может включать в себя более одной детали каждого наименования, и нам нужно где-то хранить количество деталей, входящих в состав каждой машины.

Реализовать это на практике можно, создав *связь с дополнительными данными*. Делается это в два шага:

1. Объявить связующую модель, которая, во-первых, создаст связь "многие-со-многими" между ведущей и ведомой моделями, а во-вторых, сохранит дополнительные данные этой связи (в нашем случае — количество деталей, входящих в состав машины).

   В связующей модели должны присутствовать:

   - поле типа `ForeignKey` для связи с ведущей моделью;
   - поле типа `ForeignKey` для связи с ведомой моделью;
   - поля нужных типов для хранения дополнительных данных.

2. Объявить в ведущей модели поле типа `ManyToManyField` для связи с ведомой моделью. В параметре `through` этого поля следует указать имя связующей модели, представленное в виде строки, а в параметре `through_fields` — кортеж из двух элементов:

   - имени поля связующей модели, по которому устанавливается связь с ведущей моделью;
   - имени поля связующей модели, по которому устанавливается связь с ведомой моделью.

   На заметку

   Вообще-то параметр `through_fields` обязательно указывается только в том случае, если связующая модель связана с ведущей или ведомой несколькими связями, и, соответственно, в ней присутствуют несколько полей внешнего ключа, устанавливающих эти связи. Но знать о нем все равно полезно.

Ведомая модель объявляется так же, как и в случае обычной связи "многие-со-многими".

В листинге 16.1 приведен код трех моделей: ведомой `Spare`, ведущей `Machine` и связующей `Kit`.

Листинг 16.1. Создание связи "многие-со-многими" с дополнительными данными

```
from django.db import models

class Spare(models.Model):
    name = models.CharField(max_length=40)
```

```
class Machine(models.Model):
    name = models.CharField(max_length=30)
    spares = models.ManyToManyField(Spare, through='Kit',
                                    through_fields=('machine', 'spare'))

class Kit(models.Model):
    machine = models.ForeignKey(Machine, on_delete=models.CASCADE)
    spare = models.ForeignKey(Spare, on_delete=models.CASCADE)
    count = models.IntegerField()
```

Написав классы моделей, сформировав и выполнив миграции, мы можем работать с данными с применением способов, хорошо знакомых нам по *главе 6*. Сначала мы создадим записи в моделях `Spare` и `Machine`:

```
>>> from testapp.models import Spare, Machine, Kit
>>> s1 = Spare.objects.create(name='Болт')
>>> s2 = Spare.objects.create(name='Гайка')
>>> s3 = Spare.objects.create(name='Шайба')
>>> m1 = Machine.objects.create(name='Самосвал')
>>> m2 = Machine.objects.create(name='Тепловоз')
```

Связи мы можем создавать следующими способами:

- напрямую — создавая записи непосредственно в связующей модели. Добавим в состав самосвала 10 болтов (сообщения, выводимые консолью, пропущены ради краткости):

  ```
  >>> Kit.objects.create(machine=m1, spare=s1, count=10)
  ```

- методом `add()`, который был рассмотрен в *разд. 6.6.3* (начиная с Django 2.2), — добавив в него параметр `through_defaults` и присвоив ему значения полей записи, создаваемой в связующей модели. Значением этого параметра должен быть словарь, элементы которого соответствуют полям связующей модели, а значения зададут значения для этих полей.

  Для примера добавим в состав самосвала 100 гаек:

  ```
  >>> m1.spares.add(s2, through_defaults={'count': 100})
  ```

- методом `create()`, описанным в *разд. 6.6.3* (начиная с Django 2.2), — добавив в него аналогичный параметр `through_defaults`.

  В качестве примера создадим новую деталь — шпильку — и добавим две таковых в самосвал:

  ```
  >>> s4 = m1.spares.create(name='Шпилька', through_defaults={'count': 2})
  ```

- методом `set()`, описанным в *разд. 6.6.3* (начиная с Django 2.2), — вставив в его вызов аналогичный параметр `through_defaults`.

  Добавим в состав тепловоза 49 шайб и столько же болтов:

  ```
  >>> m2.spares.set([s1, s3], clear=True, through_defaults={'count': 49})
  ```

Внимание!

Значения, заданные в параметре `through_defaults` метода `set()`, будут сохранены во всех записях, что создаются в связующей модели. Задать отдельные значения для отдельных записей связующей модели, к сожалению, нельзя.

При создании связей "многие-со-многими" вызовом метода `set()` настоятельно рекомендуется указывать в нем параметр `clear` со значением `True`. Это необходимо для гарантированного обновления значений полей в записях связующей модели.

Проверим, какие детали содержит тепловоз:

```
>>> for s in m1.spares.all(): print(s.name, end=' ')
...
Болт Гайка Шпилька
```

Выведем список деталей, которые содержит самосвал, с указанием их количества:

```
>>> for s in m1.spares.all():
...     kit = s.kit_set.get(machine=m1)
...     print(s.name, ' (', kit.count, ')', sep='')
...
Болт (10)
Гайка (100)
Шпилька (2)
```

Уменьшаем количество входящих в состав самосвала болтов до пяти:

```
>>> k1 = Kit.objects.get(machine=m1, spare=s1)
>>> k1.count
10
>>> k1.count = 5
>>> k1.save()
>>> k1.count
5
```

Для удаления связей можно пользоваться методами `remove()` и `clear()`, описанными в *разд. 6.6.3*. Для примера удалим из самосвала все шпильки:

```
>>> m1.spares.remove(s4)
>>> for s in m1.spares.all(): print(s.name, end=' ')
...
Болт Гайка >>>
```

16.3. Полиморфные связи

Полиморфная, или *обобщенная*, связь позволяет связать запись вторичной модели, в которой она объявлена, с записью любой модели, имеющейся в приложениях проекта, без исключений. При этом разные записи такой модели могут оказаться связанными с записями разных моделей.

Предположим, что мы хотим дать посетителю возможность оставлять заметки к рубрикам, объявлениям, машинам и составным частям. Вместо того, чтобы созда-

вать четыре совершенно одинаковые модели RubricNote, BbNote, MachineNote и SpareNote и связывать их с соответствующими моделями обычными связями "один-со-многими", мы можем написать всего одну модель Note и объявить в ней полиморфную связь.

Перед созданием полиморфных связей нужно убедиться, что приложение django.contrib.contenttypes, реализующее функциональность соответствующей подсистемы Django, присутствует в списке зарегистрированных приложений (параметр INSTALLED_APPS настроек проекта). После этого следует хотя бы раз провести выполнение миграций, чтобы Django создал в базе данных необходимые таблицы.

> ***На заметку***
>
> Для хранения списка созданных в проекте моделей, который используется подсистемой полиморфных связей в работе, в базе данных формируется таблица django_content_type. Править ее вручную не рекомендуется.

Полиморфная связь создается в классе вторичной модели. Для ее установления необходимо объявить там три сущности:

- поле для хранения типа модели, связываемой с записью. Оно должно иметь тип ForeignKey (т. е. внешний ключ для связи "один-со-многими"), устанавливать связь с моделью ContentType из модуля django.contrib.contenttypes.models (там хранится перечень всех моделей проекта) и выполнять каскадное удаление. Обычно такому полю дается имя content_type;
- поле для хранения значения ключа связываемой записи. Оно должно иметь целочисленный тип — обычно PositiveIntegerField. Как правило, этому полю дается имя object_id;
- *поле полиморфной связи*, реализуемое экземпляром класса GenericForeignKey из модуля django.contrib.contenttypes.fields. Вот формат вызова его конструктора:

```
GenericForeignKey([ct_field='content_type'][,][fk_field='object_id']
                  [,][for_concrete_model=True])
```

  Конструктор принимает следующие параметры:

  - ct_field — указывает имя поля, хранящего тип связываемой модели, если это имя отличается от content_type;
  - fk_field — указывает имя поля, хранящего ключ связываемой записи, если это имя отличается от object_id;
  - for_concrete_model — следует дать значение False, если необходимо устанавливать связи, в том числе и с прокси-моделями (о них будет рассказано позже).

Листинг 16.2 содержит код модели Note, хранящей заметки и использующей для связи с соответствующими моделями полиморфную связь.

Листинг 16.2. Пример использования полиморфной связи

```
from django.db import models
from django.contrib.contenttypes.fields import GenericForeignKey
from django.contrib.contenttypes.models import ContentType

class Note(models.Model):
    content = models.TextField()
    content_type = models.ForeignKey(ContentType,
                                     on_delete=models.CASCADE)
    object_id = models.PositiveIntegerField()
    content_object = GenericForeignKey(ct_field='content_type',
                                       fk_field='object_id')
```

Чтобы связать запись модели, содержащей полиморфную связь, с записью другой модели, последнюю следует присвоить *полю полиморфной связи*. Для примера создадим две заметки — для шпильки и тепловоза:

```
>>> from testapp.models import Spare, Machine, Note
>>> m1 = Machine.objects.get(name='Тепловоз')
>>> s1 = Spare.objects.get(name='Шпилька')
>>> n1 = Note.objects.create(content='Самая бесполезная деталь',
                             content_object=s1)
>>> n2 = Note.objects.create(content='В нем не используются шпильки',
                             content_object=m1)
>>> n1.content_object.name
'Шпилька'
>>> n2.content_object.name
'Тепловоз'
```

Из поля, хранящего тип связанной модели, можно получить объект записи, описывающей этот тип. А обратившись к полю `name` этой записи, мы получим сам тип связанной модели, фактически — имя ее класса, приведенное к нижнему регистру:

```
>>> n2.content_type.name
'machine'
```

Переберем в цикле все заметки и выведем их текст вместе с названиями связанных сущностей:

```
>>> for n in Note.objects.all(): print(n.content, n.content_object.name)
...
Шпилька Самая бесполезная деталь
Тепловоз В нем не используются шпильки
```

К сожалению, поле полиморфной связи нельзя использовать в условиях фильтрации. Так что вот такой код вызовет ошибку:

```
>>> notes = Note.objects.filter(content_object=s1)
```

При создании полиморфной связи Django по умолчанию не предоставляет средств для доступа из первичной модели к связанным записям вторичной модели. Такие средства придется создавать самостоятельно.

Доступ из записи первичной модели к связанным записям вторичной модели в случае полиморфной связи предоставляет так называемое *поле обратной связи*. Оно реализуется экземпляром класса `GenericRelation` из модуля `django.contrib.contenttypes.fields`. Конструктор этого класса вызывается в формате:

```
GenericRelation(<вторичная модель>[, content_type_field='content_type'][,
                object_id_field='object_id'][, for_concrete_model=True][,
                related_query_name=None])
```

Вторичная модель указывается либо в виде ссылки на класс, либо в виде строки с именем класса. Параметр `content_type_field` указывает имя поля, хранящего тип связываемой модели, а параметр `object_id_field` — имя поля с ключом связываемой записи, если эти имена отличаются от используемых по умолчанию `content_type` и `object_id` соответственно. Если одна из связываемых моделей является прокси-моделью, то параметру `for_concrete_model` нужно присвоить значение `False`. Назначение параметра `related_query_name` аналогично таковому у поля типа `ForeignKey` (см. *разд. 4.3.1*).

Чтобы из записи модели `Machine` получить список связанных заметок, нам нужно добавить в объявление ее класса такой код:

```
from django.contrib.contenttypes.fields import GenericRelation
class Machine(models.Model):
    . . .
    notes = GenericRelation('Note')
```

Поле обратной связи хранит набор связанных записей вторичной модели. Например, так мы можем перебрать все заметки, оставленные для тепловоза, и вывести на экран их содержимое:

```
>>> for n in m1.notes.all(): print(n.content)
...
В нем не используются шпильки
```

Поля типа `GenericForeingKey` никак не представляются в формах, связанных с моделями (см. *главу 13*). Однако Django позволяет создать встроенный набор форм, позволяющий работать со связанными записями (подробнее о встроенных наборах форм рассказывалось в *разд. 14.5*). Он создается посредством функции `generic_inlineformset_factory()` из модуля `django.contrib.contenttypes.forms`, со следующим форматом вызова:

```
generic_inlineformset_factory(<вторичная модель с полиморфной связью>[,
                      form=<базовая форма, связанная с моделью>][,
                      ct_field='content_type'][, fk_field='object_id'][,
                      for_concrete_model=True][,
                      fields=None][, exclude=None][, extra=3][,
                      can_order=False][, can_delete=True][,
```

```
min_num=None][, validate_min=False][,
max_num=None][, validate_max=False][,
formset=<базовый набор форм, связанных с моделью>])
```

Параметр `ct_field` указывает имя поля, хранящего тип связываемой модели, а параметр `fk_field` — имя поля, в котором сохраняется ключ связываемой записи, если эти имена отличаются от используемых по умолчанию `content_type` и `object_id` соответственно. Если одна из связываемых моделей является прокси-моделью, параметру `for_concrete_model` нужно дать значение `False`. Базовый набор записей, указываемый в параметре `formset`, должен быть производным от класса `BaseGenericInlineFormSet` из модуля `django.contrib.contenttypes.forms`.

16.4. Наследование моделей

Модель Django — это обычный класс Python. Следовательно, мы можем объявить модель, являющуюся подклассом другой модели. Причем фреймворк предлагает нам целых три способа сделать это.

16.4.1. Прямое наследование моделей

В случае *прямого*, или *многотабличного*, один класс модели просто наследуется от другого.

Листинг 16.3 показывает код двух моделей: базовой `Message`, хранящей сообщения, и производной `PrivateMessage`, которая хранит частные сообщения для зарегистрированных пользователей.

Листинг 16.3. Пример прямого (многотабличного) наследования моделей

```
from django.db import models
from django.contrib.auth.models import User

class Message(models.Model):
    content = models.TextField()

class PrivateMessage(Message):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
```

В этом случае Django поступит следующим образом:

- создаст в базе данных две таблицы — для базовой и производной моделей. Каждая из таблиц будет включать только те поля, которые объявлены в соответствующей модели;
- установит между моделями связь "один-с-одним". Базовая модель станет первичной, а производная — вторичной;
- создаст в производной модели поле с именем вида *<имя базовой модели>*`_ptr`, реализующее связь "один-с-одним" с базовой моделью.

Такое служебное поле можно создать вручную, дав ему произвольное имя, указав для него каскадное удаление и обязательно присвоив его параметру `parent_link` значение `True`. Пример:

```
class PrivateMessage(Message):
    . . .
    message = models.OneToOneField(Message, on_delete=models.CASCADE,
                                   parent_link=True)
```

- создаст в каждом объекте записи базовой модели атрибут с именем вида *<имя производной модели>*, хранящий объект записи производной модели;
- при сохранении данных в записи производной модели — сохранит значения полей, унаследованных от базовой модели, в таблице базовой модели, а значения полей производной модели — в таблице производной модели;
- при удалении записи — фактически удалит обе записи: из базовой и производной моделей.

Есть возможность удалить запись производной модели, оставив связанную запись базовой модели. Для этого при вызове у записи производной модели метода `delete()` следует указать в нем параметр `keep_parents` со значением `True`.

Производная модель наследует от базовой все параметры, заданные во вложенном классе `Meta`. При необходимости эти параметры можно переопределить в производной модели.

Для примера добавим в модель `PrivateMessage` запись и получим значения ее полей:

```
>>> from testapp.models import PrivateMessage
>>> from django.contrib.auth.models import User
>>> u = User.objects.get(username='editor')
>>> pm = PrivateMessage.objects.create(content='Привет, editor!', user=u)
>>> pm.content
'Привет, editor!'
>>> pm.user
<User: editor>
```

Получим связанную запись базовой модели `Message`:

```
>>> m = pm.message_ptr
>>> m
<Message: Message object (1)>
```

Получим запись производной модели из записи базовой модели:

```
>>> m.privatemessage
<PrivateMessage: PrivateMessage object (1)>
```

Прямое наследование может выручить, если нужно хранить в базе данных набор сущностей с примерно одинаковым набором полей. Тогда поля, общие для всех типов сущностей, выносятся в базовую модель, а в производных объявляются только поля, уникальные для каждого конкретного типа сущностей.

16.4.2. Абстрактные модели

Второй способ наследовать одну модель от другой — пометить базовую модель как *абстрактную*, задав в ней (т. е. во вложенном классе `Meta`) параметр `abstract` со значением `True`. Пример:

```
class Message(models.Model):
    . . .
    class Meta:
        abstract = True

class PrivateMessage(Message):
    . . .
```

Для абстрактной базовой модели в базе данных не создается никаких таблиц. Напротив, таблица, созданная для производной от нее модели, будет содержать весь набор полей, объявленных как в базовой, так и в производной модели.

Поля, объявленные в базовой абстрактной модели, могут быть переопределены в производной. Можно даже удалить объявленное в базовой модели поле, объявив в производной модели атрибут класса с тем же именем и присвоив ему значение `None`. Пример такого переопределения и удаления полей приведен в листинге 16.4.

Листинг 16.4. Переопределение и удаление полей, объявленных в базовой абстрактной модели

```
class Message(models.Model):
    content = models.TextField()
    name = models.CharField(max_length=20)
    email = models.EmailField()

    class Meta:
        abstract = True

class PrivateMessage(Message):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    # Переопределяем поле name
    name = models.CharField(max_length=40)
    # Удаляем поле email
    email = None
```

Параметры абстрактной базовой модели, объявленные во вложенном классе `Meta`, не наследуются производной моделью автоматически. Однако есть возможность унаследовать их, объявив вложенный класс `Meta` как производный от класса `Meta` базовой модели. Пример:

```
class Message(models.Model):
    . . .
```

```
    class Meta:
        abstract = True
        ordering = ['name']

class PrivateMessage(Message):
    . . .
    class Meta(Message.Meta):
        pass
```

Наследование с применением абстрактной базовой модели применяется в тех же случаях, что и прямое наследование (см. *разд. 16.4.1*). Оно предлагает более гибкие возможности по объявлению набора полей в производных моделях — так, мы можем переопределить или даже удалить какое-либо поле, объявленное в базовой модели. К тому же, поскольку все данные хранятся в одной таблице, работа с ними выполняется быстрее, чем в случае прямого наследования.

16.4.3. Прокси-модели

Третий способ — объявить производную модель как *прокси-модель*. Классы такого типа не предназначены для расширения или изменения набора полей базовой модели, а служат для расширения или изменения ее функциональности.

Прокси-модель объявляется заданием в параметрах производной модели (во вложенном классе `Meta`) параметра `proxy` со значением `True`. В базовой модели при этом никаких дополнительных параметров записывать не нужно.

Для прокси-модели не создается никаких таблиц в базе данных. Все данные хранятся в таблице базовой модели.

Листинг 16.5 содержит код прокси-модели `RevRubric`, производной от модели `Rubric` и задающей сортировку рубрик по убыванию названия.

Листинг 16.5. Пример прокси-модели

```
class RevRubric(Rubric):
    class Meta:
        proxy = True
        ordering = ['-name']
```

Для примера выведем список рубрик из только что созданной прокси-модели:

```
>>> from bboard.models import RevRubric
>>> for r in RevRubric.objects.all(): print(r.name, end=' ')
...
Транспорт Сельхозинвентарь Сантехника Растения Недвижимость Мебель
Бытовая техника
```

16.5. Создание своих диспетчеров записей

Диспетчер записей — это объект, предоставляющий доступ к набору записей, которые хранятся в модели. По умолчанию он представляет собой экземпляр класса `Manager` из модуля `django.db.models` и хранится в атрибуте `objects` модели.

16.5.1. Создание диспетчеров записей

Диспетчеры записей наследуются от класса `Manager`. В них можно как переопределять имеющиеся методы, так и объявлять новые.

Переопределять имеет смысл только метод `get_queryset(self)`, который должен возвращать набор записей текущей модели в виде экземпляра класса `QuerySet` из модуля `django.db.models`. Обычно в теле переопределенного метода сначала вызывают тот же метод базового класса, чтобы получить изначальный набор записей, устанавливают у него фильтрацию, сортировку, добавляют вычисляемые поля и возвращают в качестве результата.

В листинге 16.6 приведен код диспетчера записей `RubricManager`, который возвращает набор рубрик уже отсортированным по полям `order` и `name`. Помимо того, он объявляет дополнительный метод `order_by_bb_count()`, который возвращает набор рубрик, отсортированный по убыванию количества относящихся к ним объявлений.

Листинг 16.6. Пример объявления собственного диспетчера записей

```
from django.db import models

class RubricManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().order_by('order', 'name')

    def order_by_bb_count(self):
        return super().get_queryset().annotate(
                        cnt=models.Count('bb')).order_by('-cnt')
```

Использовать новый диспетчер записей в модели можно трояко:

- в качестве единственного диспетчера записей — объявив в классе модели атрибут `objects` и присвоив ему экземпляр класса диспетчера записей:

  ```
  class Rubric(models.Model):
      . . .
      objects = RubricManager()
  ```

  Теперь, обратившись к атрибуту `objects` модели, мы получим доступ к нашему диспетчеру:

  ```
  >>> from bboard.models import Rubric
  >>> # Получаем набор записей, возвращенный методом get_queryset()
  >>> Rubric.objects.all()
  ```

```
<QuerySet [<Rubric: Транспорт>, <Rubric: Недвижимость>,
<Rubric: Мебель>, <Rubric: Бытовая техника>, <Rubric: Сантехника>,
<Rubric: Растения>, <Rubric: Сельхозинвентарь>]>
>>> # Получаем набор записей, возвращенный вновь добавленным методом
>>> # order_by_bb_count()
>>> Rubric.objects.order_by_bb_count()
<QuerySet [<Rubric: Недвижимость>, <Rubric: Транспорт>,
<Rubric: Бытовая техника>, <Rubric: Сельхозинвентарь>,
<Rubric: Мебель>, <Rubric: Сантехника>, <Rubric: Растения>]>
```

- то же самое, только с использованием атрибута класса с другим именем:

```
class Rubric(models.Model):
    . . .
    bbs = RubricManager()
. . .
>>> # Теперь для доступа к диспетчеру записей используем атрибут
>>> # класса bbs
>>> Rubric.bbs.all()
```

- в качестве дополнительного диспетчера записей — присвоив его другому атрибуту класса модели:

```
class Rubric(models.Model):
    . . .
    objects = models.Manager()
    bbs = RubricManager()
```

Теперь в атрибуте `objects` хранится диспетчер записей, применяемый по умолчанию, а в атрибуте `bbs` — наш диспетчер записей. И мы можем пользоваться сразу двумя диспетчерами записей. Пример:

```
>>> Rubric.objects.all()
<QuerySet [<Rubric: Бытовая техника>, <Rubric: Мебель>,
<Rubric: Недвижимость>, <Rubric: Растения>, <Rubric: Сантехника>,
<Rubric: Сельхозинвентарь>, <Rubric: Транспорт>]>
>>> Rubric.bbs.all()
<QuerySet [<Rubric: Транспорт>, <Rubric: Недвижимость>,
<Rubric: Мебель>, <Rubric: Бытовая техника>, <Rubric: Сантехника>,
<Rubric: Растения>, <Rubric: Сельхозинвентарь>]>
```

Здесь нужно учитывать один момент. Первый объявленный в модели диспетчер записей будет рассматриваться Django как используемый по умолчанию, применяемый при выполнении различных служебных задач (в нашем случае это диспетчер записей `Manager`, присвоенный атрибуту `objects`). Поэтому ни в коем случае нельзя задавать в таком диспетчере данных фильтрацию, иначе записи модели, не удовлетворяющие ее критериям, окажутся необработанными.

На заметку

Можно дать атрибуту, применяемому для доступа к диспетчеру записей, другое имя без задания для него нового диспетчера записей:

```
class Rubric(models.Model):
    . . .
    bbs = models.Manager()
```

16.5.2. Создание диспетчеров обратной связи

Аналогично можно создать свой диспетчер обратной связи, который выдает набор записей вторичной модели, связанный с текущей записью первичной модели. Его класс также объявляется как производный от класса `Manager` из модуля `django.db.models` и также указывается в модели присваиванием его экземпляра атрибуту класса модели.

Листинг 16.7 показывает код диспетчера обратной связи `BbManager`, возвращающий связанные объявления отсортированными по возрастанию цены.

Листинг 16.7. Пример диспетчера обратной связи

```
from django.db import models

class BbManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().order_by('price')
```

Чтобы из записи первичной модели получить набор связанных записей вторичной модели с применением нового диспетчера обратной связи, придется явно указать этот диспетчер. Для этого у объекта первичной записи вызывается метод, имя которого совпадает с именем атрибута, хранящего диспетчер связанных записей. Этому методу передается параметр `manager`, в качестве значения ему присваивается строка с именем атрибута класса вторичной модели, которому был присвоен объект нового диспетчера. Метод вернет в качестве результата набор записей, сформированный этим диспетчером.

Так, указать диспетчер обратной связи `BbManager` в классе модели `Bb` можно следующим образом (на всякий случай не забыв задать диспетчер, который будет использоваться по умолчанию):

```
class Bb(models.Model):
    . . .
    objects = models.Manager()
    by_price = BbManager()
```

Проверим его в деле:

```
>>> from bboard.models import Rubric
>>> r = Rubric.objects.get(name='Недвижимость')
>>> # Используем диспетчер обратной связи по умолчанию. Объявления будут
>>> # выведены отсортированными по умолчанию - по убыванию даты
>>> # их публикации
>>> r.bb_set.all()
```

```
<QuerySet [<Bb: Bb object (6)>, <Bb: Bb object (3)>,
<Bb: Bb object (1)>]>
>>> # Используем свой диспетчер обратной связи. Объявления сортируются
>>> # по возрастанию цены
>>> r.bb_set(manager='by_price').all()
<QuerySet [<Bb: Bb object (6)>, <Bb: Bb object (1)>,
<Bb: Bb object (3)>]>
```

16.6. Создание своих наборов записей

Еще можно объявить свой класс набора записей, сделав его производным от класса `QuerySet` из модуля `django.db.models`. Объявленные в нем методы могут фильтровать и сортировать записи по часто встречающимся критериям, выполнять в них часто применяемые агрегатные вычисления и создавать часто используемые вычисляемые поля.

Листинг 16.8 показывает код набора записей `RubricQuerySet` с дополнительным методом, вычисляющим количество объявлений, имеющихся в каждой рубрике, и сортирующим рубрики по убыванию этого количества.

Листинг 16.8. Пример собственного набора записей

```
from django.db import models

class RubricQuerySet(models.QuerySet):
    def order_by_bb_count(self):
        return self.annotate(cnt=models.Count('bb')).order_by('-cnt')
```

Для того чтобы модель возвращала набор записей, представленный экземпляром объявленного нами класса, понадобится также объявить свой диспетчер записей (как это сделать, было рассказано в *разд. 16.5*). Прежде всего, в методе `get_queryset()` он сформирует и вернет в качестве результата экземпляр нового класса набора записей. Конструктору этого класса в качестве первого позиционного параметра следует передать используемую модель, которую можно извлечь из атрибута `model`, а в качестве параметра `using` — базу данных, в которой хранятся записи модели и которая извлекается из атрибута `_db`.

Помимо этого, нужно предусмотреть вариант, когда объявленные в новом наборе записей дополнительные методы вызываются не у набора записей:

```
rs = Rubric.objects.all().order_by_bb_count()
```

а непосредственно у диспетчера записей:

```
rs = Rubric.objects.order_by_bb_count()
```

Для этого придется объявить одноименные методы еще и в классе набора записей и выполнять в этих методах вызовы соответствующих им методов набора записей.

В листинге 16.9 приведен код диспетчера записей `RubricManager`, призванного обслуживать набор записей `RubricQuerySet` (см. листинг 16.8).

Листинг 16.9. Пример диспетчера записей, обслуживающего набор записей из листинга 16.8

```
from django.db import models

class RubricManager(models.Manager):
    def get_queryset(self):
        return RubricQuerySet(self.model, using=self._db)

    def order_by_bb_count(self):
        return self.get_queryset().order_by_bb_count()
```

Новый диспетчер записей указывается в классе модели описанным в *разд. 16.5* способом:

```
class Rubric(models.Model):
    . . .
    objects = RubricManager()
```

Проверим созданный набор записей в действии (вывод пропущен ради краткости):

```
>>> from bboard.models import Rubric
>>> Rubric.objects.all()
. . .
>>> Rubric.objects.order_by_bb_count()
. . .
```

Писать свой диспетчер записей только для того, чтобы "подружить" модель с новым набором записей, вовсе не обязательно. Можно использовать одну из двух фабрик классов, создающих классы диспетчеров записей на основе наборов записей и реализованных в виде методов:

- `as_manager()` — вызывается у класса набора записей и возвращает обслуживающий его экземпляр класса диспетчера записей:

```
class Rubric(models.Model):
    . . .
    objects = RubricQuerySet.as_manager()
```

- `from_queryset(`*<класс набора записей>*`)` — вызывается у класса диспетчера записей и возвращает ссылку на производный класс диспетчера записей, обслуживающий заданный *набор записей*:

```
class Rubric(models.Model):
    . . .
    objects = models.Manager.from_queryset(RubricQuerySet)()
```

Можно сказать, что обе фабрики классов создают новый класс набора записей и переносят в него методы из базового набора записей. В обоих случаях действуют следующие правила переноса методов:

- обычные методы переносятся по умолчанию;
- псевдочастные методы (имена которых предваряются символом подчеркивания) не переносятся по умолчанию;
- записанный у метода атрибут `queryset_only` со значением `False` указывает перенести метод;
- записанный у метода атрибут `queryset_only` со значением `True` указывает не переносить метод.

Пример:

```
class SomeQuerySet(models.QuerySet):
    # Этот метод будет перенесен
    def method1(self):
        . . .
    # Этот метод не будет перенесен
    def _method2(self):
        . . .

    # Этот метод будет перенесен
    def _method3(self):
        . . .
    _method3.queryset_only = False

    # Этот метод не будет перенесен
    def method4(self):
        . . .
    _method4.queryset_only = True
```

16.7. Управление транзакциями

Django предоставляет удобные инструменты для управления транзакциями, которые пригодятся при программировании сложных решений.

16.7.1. Автоматическое управление транзакциями

Проще всего активизировать автоматическое управление транзакциями, при котором Django самостоятельно запускает транзакции и завершает их — с подтверждением, если все прошло нормально, или с откатом, если в контроллере возникла ошибка.

> ***Внимание!***
>
> Автоматическое управление транзакциями работает только в контроллерах. В других модулях (например, посредниках) управлять транзакциями придется вручную.

Автоматическое управление транзакциями в Django может функционировать в двух режимах.

16.7.1.1. Режим по умолчанию: каждая операция — в отдельной транзакции

В этом режиме каждая отдельная операция с моделью — чтение, добавление, правка и удаление записей — выполняется в отдельной транзакции.

Чтобы активировать этот режим, следует задать такие настройки базы данных (см. *разд. 3.3.2*):

- параметру `ATOMIC_REQUEST` — дать значение `False` (или вообще удалить этот параметр, поскольку `False` — его значение по умолчанию). Тем самым мы предпишем выполнять каждую операцию с базой данных в отдельной транзакции;
- параметру `AUTOCOMMIT` — дать значение `True` (или вообще удалить этот параметр, поскольку `True` — его значение по умолчанию). Так мы включим автоматическое завершение транзакций по окончании выполнения контроллера.

Собственно, во вновь созданном проекте Django база данных изначально настроена на работу в этом режиме.

Режим по умолчанию подходит для случаев, когда в контроллере выполняется не более одной операции с базой данных. В простых сайтах наподобие нашей доски объявлений обычно так и бывает.

> ***На заметку***
>
> Начиная с Django 2.2, операция с базой данных, которая может быть выполнена в один запрос, не заключается в транзакцию. Это сделано для повышения производительности.

16.7.1.2. Режим атомарных запросов

В этом режиме *все* операции с базой данных, происходящие в контроллере (т. е. на протяжении одного HTTP-запроса), выполняются в одной транзакции.

Переключить Django-сайт в такой режим можно, задав следующие настройки у базы данных:

- параметру `ATOMIC_REQUEST` — дать значение `True`, чтобы, собственно, включить режим атомарных запросов;
- параметру `AUTOCOMMIT` — дать значение `True` (или вообще удалить этот параметр).

Пример:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
        'ATOMIC_REQUEST': True,
    }
}
```

Этот режим следует активировать, если в одном контроллере выполняется сразу несколько операций, изменяющих данные в базе. Он гарантирует, что или в базу

будут внесены все требуемые изменения, или, в случае возникновения нештатной ситуации, база останется в своем изначальном состоянии.

16.7.1.3. Режим по умолчанию на уровне контроллера

Если на уровне базы данных включен режим атомарных запросов, то на уровне какого-либо контроллера-функции можно включить режим управления транзакциями по умолчанию (в котором отдельная операция с базой данных выполняется в отдельной транзакции). Для этого достаточно указать перед контроллером-функцией декоратор `non_atomic_requests()` из модуля `django.db.transaction`. Пример:

```
from django.db import transaction

@transaction.non_atomic_requests
def my_view(request):
    # В этом контроллере действует режим обработки транзакций по умолчанию
```

16.7.1.4. Режим атомарных запросов на уровне контроллера

Аналогично, если на уровне базы данных действует режим по умолчанию, то на уровне какого-либо контроллера-функции можно включить режим атомарных запросов. Для этого применяется функция `atomic([savepoint=True])` из модуля `django.db.transaction`. Ее можно использовать как:

- декоратор, указываемый перед контроллером-функцией:

```
from django.db.transaction import atomic

@atomic
def edit(request, pk):
    # В этом контроллере будет действовать режим атомарных запросов
    . . .
```

- менеджер контекста в блоке `with`, в содержимом которого нужно включить режим атомарных запросов:

```
if formset.is_valid():
    with atomic():
        # Выполняем сохранение всех форм набора в одной транзакции
    return redirect('bboard:index')
```

Допускаются вложенные блоки `with`:

```
if formset.is_valid():
    with atomic():
        for form in formset:
            if form.cleaned_data:
                with atomic():
                    # Выполняем сохранение каждой формы набора
                    # в отдельном вложенном блоке with
```

В этом случае при входе во внешний блок `with` будет, собственно, запущена транзакция, а при входе во вложенный блок `with` — создана точка сохранения. При выходе из вложенного блока выполняется подтверждение точки сохранения (если все прошло успешно) или же откат до состояния на момент ее создания (в случае возникновения ошибки). Наконец, после выхода из внешнего блока `with` происходит подтверждение или откат самой транзакции.

Каждая созданная точка сохранения отнимает системные ресурсы. Поэтому предусмотрена возможность отключить их создание, для чего достаточно в вызове функции `atomic()` указать параметр `savepoint` со значением `False`.

16.7.2. Ручное управление транзакциями

Если активно ручное управление транзакциями, то запускать транзакции, как и при автоматическом режиме, будет фреймворк, но завершать их нам придется самостоятельно.

Чтобы активировать ручной режим управления транзакциями, нужно указать следующие настройки базы данных:

- параметру `ATOMIC_REQUEST` — дать значение `False` (если каждая операция с базой данных должна выполняться в отдельной транзакции) или `True` (если все операции с базой данных должны выполняться в одной транзакции);
- параметру `AUTOCOMMIT` — дать значение `False`, чтобы отключить автоматическое завершение транзакций.

Для ручного управления транзакциями применяются следующие функции из модуля `django.db.transaction`:

- `commit()` — завершает транзакцию с подтверждением;
- `rollback()` — завершает транзакцию с откатом;
- `savepoint()` — создает новую точку сохранения и возвращает ее идентификатор в качестве результата;
- `savepoint_commit(`*<идентификатор точки сохранения>*`)` — выполняет подтверждение точки сохранения с указанным *идентификатором*;
- `savepoint_rollback(`*<идентификатор точки сохранения>*`)` — выполняет откат до точки сохранения с указанным *идентификатором*;
- `clean_savepoints()` — сбрасывает счетчик, применяемый для генерирования уникальных идентификаторов точек сохранения;
- `get_autocommit()` — возвращает `True`, если для базы данных включен режим автоматического завершения транзакции, и `False` — в противном случае;
- `set_autocommit(`*<режим>*`)` — включает или отключает режим автоматического завершения транзакции для базы данных. *Режим* указывается в виде логической величины: `True` включает автоматическое завершение транзакций, `False` — отключает.

Пример ручного управления транзакциями при сохранении записи:

```
from django.db import transaction
. . .
if form.is_valid():
    try:
        form.save()
        transaction.commit()
    except:
        transaction.rollback()
```

Пример ручного управления транзакциями при сохранении набора форм с использованием точек сохранения:

```
if formset.is_valid():
    for form in formset:
        if form.cleaned_data:
            sp = transaction.savepoint()
            try:
                form.save()
                transaction.savepoint_commit(sp)
            except:
                transaction.savepoint_rollback(sp)
            transaction.commit()
```

16.7.3. Обработка подтверждения транзакции

Существует возможность обработать момент подтверждения транзакции. Для этого достаточно вызвать функцию `on_commit(<функция-обработчик>)` из модуля `django.db.transaction`, передав ей ссылку на *функцию-обработчик*. Последняя не должна ни принимать параметров, ни возвращать результат. Пример:

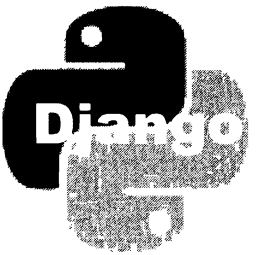
```
from django.db import transaction

def commit_handler():
    # Выполняем какие-либо действия после подтверждения транзакции

transaction.on_commit(commit_handler)
```

Указанная функция будет вызвана только после подтверждения транзакции, но не после ее отката, подтверждения или отката точки сохранения. Если в момент вызова функции `on_commit()` активная транзакция отсутствует, то функция-обработчик выполнена не будет.

ГЛАВА 17



Формы и наборы форм: расширенные инструменты и дополнительная библиотека

Помимо форм и наборов форм, связанных с моделями, Django предлагает аналогичные инструменты, которые не связаны с моделями. Они могут применяться для указания как данных, не предназначенных для занесения в базу данных (например, ключевого слова для поиска), так и сведений, которые должны быть сохранены в базе после дополнительной обработки.

Кроме того, фреймворк предлагает расширенные инструменты для вывода форм и наборов форм на экран. А дополнительная библиотека Django Simple Captcha позволяет обрабатывать CAPTCHA.

17.1. Формы, не связанные с моделями

Формы, не связанные с моделями, создаются и обрабатываются так же, как их "коллеги", которые связаны с моделями, за несколькими исключениями:

- форма, не связанная с моделью, объявляется как подкласс класса `Form` из модуля `django.forms`;
- все поля, которые должны присутствовать в форме, необходимо объявлять в виде атрибутов класса формы;
- вложенный класс `Meta` в такой форме не объявляется (что вполне понятно — ведь в этом классе задаются сведения о модели, с которой связана форма);
- средства для сохранения введенных в форму данных в базе, включая метод `save()` и параметр `instance` конструктора, не поддерживаются.

В листинге 17.1 приведен код не связанной с моделью формы, предназначенной для указания искомого ключевого слова и рубрики, в которой будет осуществляться поиск объявлений.

Листинг 17.1. Форма, не связанная с моделью

```
from django import forms

class SearchForm(forms.Form):
    keyword = forms.CharField(max_length=20, label='Искомое слово')
    rubric = forms.ModelChoiceField(queryset=Rubric.objects.all(),
                                    label='Рубрика')
```

В листинге 17.2 приведен код контроллера, который извлекает данные из этой формы и использует их для указания параметров фильтрации объявлений (предполагается, что форма пересылает данные методом POST).

Листинг 17.2. Контроллер, который использует форму, не связанную с моделью

```
def search(request):
    if request.method == 'POST':
        sf = SearchForm(request.POST)
        if sf.is_valid():
            keyword = sf.cleaned_data['keyword']
            rubric_id = sf.cleaned_data['rubric'].pk
            bbs = Bb.objects.filter(title__icontains=keyword,
                                    rubric=rubric_id)
            context = {'bbs': bbs}
            return render(request, 'bboard/search_results.html', context)
    else:
        sf = SearchForm()
    context = {'form': sf}
    return render(request, 'bboard/search.html', context)
```

17.2. Наборы форм, не связанные с моделями

Наборы форм, не связанные с моделями, создаются с применением функции `formset_factory()` из модуля `django.forms`:

```
formset_factory(form=<форма>[, extra=1][,
                can_order=False][, can_delete=False][,
                min_num=None][, validate_min=False][,
                max_num=None][, validate_max=False][,
                formset=<базовый набор форм>])
```

Здесь *форма*, на основе которой создается набор форм, является обязательным параметром. *Базовый набор форм* должен быть производным от класса `BaseFormSet` из модуля `django.forms.formsets`. Назначение остальных параметров было описано в *разд. 14.1*.

Пример создания набора форм на основе формы `SearchForm`, код которой приведен в листинге 17.1:

```
from django.forms import formset_factory
fs = formset_factory(SearchForm, extra=3, can_delete=True)
```

Опять же, нужно иметь в виду, что набор форм, не связанный с моделью, не поддерживает средств для сохранения в базе занесенных в него данных, включая атрибуты `new_objects`, `changed_objects` и `deteled_objects`. Код, сохраняющий введенные данные, придется писать самостоятельно.

В остальном же работа с такими наборами форм протекает аналогично работе с наборами форм, связанными с моделями. Мы можем перебирать формы, содержащиеся в наборе, и извлекать из них данные для обработки.

Если набор форм поддерживает переупорядочение форм (т. е. при его создании в вызове функции `formset_factory()` был указан параметр `can_order` со значением `True`), то в составе каждой формы появится поле `ORDER` типа `IntegerField`, хранящее порядковый номер текущей формы. Мы можем использовать его, чтобы выстроить в нужном порядке какие-либо сущности, или иным образом.

Если набор форм поддерживает удаление отдельных форм (добавить эту поддержку можно, записав в вызове функции `formset_factory()` параметр `can_delete` со значением `True`), то в составе каждой формы появится поле `DELETE` типа `BooleanField`. Это поле будет хранить значение `True`, если форма была помечена на удаление, и `False` — в противном случае.

Класс набора форм, не связанного с моделью, поддерживает два полезных атрибута:

- `ordered_forms` — последовательность форм, которые были переупорядочены;
- `deleted_forms` — последовательность удаленных форм.

В листинге 17.3 приведен код контроллера, который обрабатывает набор форм, не связанный с моделью.

Листинг 17.3. Контроллер, который обрабатывает набор форм, не связанный с моделью

```
def formset_processing(request):
    FS = formset_factory(SearchForm, extra=3, can_order=True,
                                              can_delete=True)
    if request.method == 'POST':
        formset = FS(request.POST)
        if formset.is_valid():
            for form in formset:
                if form.cleaned_data and not form.cleaned_data['DELETE']:
                    keyword = form.cleaned_data['keyword']
                    rubric_id = form.cleaned_data['rubric'].pk
                    order = form.cleaned_data['ORDER']
                    # Выполняем какие-либо действия над полученными
                    # данными
            return render(request, 'bboard/process_result.html')
```

```
    else:
        formset = FS()
    context = {'formset': formset}
    return render(request, 'bboard/formset.html', context)
```

17.3. Расширенные средства для вывода форм и наборов форм

Эти средства поддерживаются обеими разновидностями форм: и связанными с моделями, и не связанными с ними.

17.3.1. Указание CSS-стилей у форм

Для указания CSS-стилей, которые будут применены к отдельным элементам выводимой формы, классы форм поддерживают два атрибута класса:

- `required_css_class` — имя стилевого класса, которым будут помечаться элементы управления, обязательные для заполнения;
- `error_css_class` — имя стилевого класса, которым будут помечаться элементы управления с некорректными данными.

Эти стилевые классы будут привязываться к тегам `<p>`, `<li>` или `<tr>`, в зависимости от того, посредством каких HTML-тегов форма была выведена на экран.

Пример:

```
class SearchForm(forms.Form):
    error_css_class = 'error'
    required_css_class = 'required'
```

17.3.2. Настройка выводимых форм

Некоторые настройки, затрагивающие выводимые на экран формы, указываются в виде именованных параметров конструктора класса формы. Вот эти параметры:

- `field_order` — задает порядок следования полей формы при ее выводе на экран. В качестве значения указывается последовательность имен полей, представленных в виде строк. Если задать `None`, поля будут следовать друг за другом в том же порядке, в котором они были объявлены в классе формы. Значение по умолчанию — `None`. Пример:

  ```
  bf = BbForm(field_order=('rubric', 'rubric', 'price', 'content'))
  ```

- `label_suffix` — строка с суффиксом, который будет добавлен к тексту надписи при выводе. Значение по умолчанию — символ двоеточия;
- `auto_id` — управляет формированием якорей элементов управления, которые указываются в атрибутах `id` формирующих их тегов и тегов `<label>`, создающих надписи. В качестве значения параметра можно указать:

- строку формата — идентификаторы будут формироваться согласно ей. Символ-заменитель `%s` указывает местоположение в строке формата имени поля, соответствующего элементу управления. Пример:

  ```
  sf = SearchForm(auto_id='id_for_%s')
  ```

  Для поля `keyword` будет сгенерирован якорь `id_for_keyword`, а для поля `rubric` — якорь `id_for_rubric`;

- `True` — в качестве якорей будут использоваться имена полей формы, соответствующих элементам управления;

- `False` — якоря вообще не будут формироваться. Также не будут формироваться теги `<label>`, а надписи будут представлять собой простой текст.

Значение параметра по умолчанию: `"id_%s"`;

- `use_required_attribute` — если `True`, то в теги, формирующие обязательные для заполнения элементы управления, будут помещены атрибуты `required`, если `False`, то этого не произойдет (по умолчанию — `True`);
- `prefix` — строковый префикс для имен полей в выводимой форме. Применяется, если в один тег `<form>` нужно поместить несколько форм. По умолчанию — `None` (префикс отсутствует).

17.3.3. Настройка наборов форм

Конструкторы классов наборов форм поддерживают именованные параметры `auto_id` и `prefix`, описанные в *разд. 17.3.2*:

```
formset = FS(auto_id=False)
```

Поле порядкового номера `ORDER`, посредством которого выполняется переупорядочивание форм, и поле удаления формы `DELETE` доступны через одноименные элементы. Это можно использовать, чтобы вывести служебные поля отдельно от остальных. Пример:

```
<h2>Рубрики</h2>
<form method="post">
    {% csrf_token %}
    {{ formset.management_form }}
    <table>
        {% for form in formset %}
        <tr><td colspan="2">{{ form.name }}</td></tr>
        <tr><td>{{ form.ORDER }}</td><td>{{ form.DELETE }}</td></tr>
        {% endfor %}
    </table>
    <p><input type="submit" value="Сохранить"></p>
</form>
```

Начиная с Django 3.0, классы наборов форм поддерживают атрибут класса `ordering_widget`. В нем указывается класс элемента управления, посредством кото-

рого выводится поле порядкового номера формы ORDER. По умолчанию используется класс NumberInput (поле для ввода целого числа).

Пример переупорядочивания форм в наборе с помощью обычного поля ввода:

```
from django.forms import modelformset_factory, BaseModelFormSet
from django.forms.widgets import TextInput

class BaseRubricFormSet(BaseModelFormSet):
    ordering_widget = TextInput

RubricFormSet = modelformset_factory(Rubric, fields=('name',),
                                     can_order=True, can_delete=True,
                                     formset=BaseRubricFormSet)
```

Эта возможность пригодится при необходимости задействовать какую-либо дополнительную библиотеку, содержащую подходящий элемент управления.

17.4. Библиотека Django Simple Captcha: поддержка CAPTCHA

Если планируется дать пользователям-гостям возможность добавлять какие-либо данные в базу (например, оставлять комментарии), не помешает как-то обезопасить форму, в которую вводятся эти данные, от программ-роботов. Одно из решений — применение *CAPTCHA* (Completely Automated Public Turing test to tell Computers and Humans Apart, полностью автоматизированный публичный тест Тьюринга для различения компьютеров и людей).

CAPTCHA выводится на веб-страницу в виде графического изображения, содержащего сильно искаженный или зашумленный текст, который нужно прочитать и занести в расположенное рядом поле ввода. Если результат оказался верным, то, скорее всего, данные занесены человеком, поскольку программам такие сложные задачи пока еще не по плечу.

Для Django существует довольно много библиотек, реализующих в формах поддержку CAPTCHA. Одна из них — Django Simple Captcha.

> ***На заметку***
>
> Полная документация по библиотеке Django Simple Captcha находится здесь: **https://django-simple-captcha.readthedocs.io/**.

17.4.1. Установка Django Simple Captcha

Установка этой библиотеки выполняется отдачей в командной строке следующей команды:

```
pip install django-simple-captcha
```

Вместе с Django Simple Captcha будут установлены библиотеки Pillow, django-ranged-response и six, необходимые ей для работы.

Чтобы задействовать библиотеку после установки, необходимо:

- добавить входящее в ее состав приложение `captcha` в список приложений проекта (параметр `INSTALLED_APPS` настроек проекта, подробнее — в *разд. 3.3.3*):

```
INSTALLED_APPS = [
    . . .
    'captcha',
]
```

- выполнить миграции, входящие в состав библиотеки:

```
manage.py migrate
```

- в списке маршрутов уровня проекта (в модуле urls.py пакета конфигурации) создать маршрут, связывающий префикс `captcha` и вложенный список маршрутов из модуля `captcha.urls`:

```
urlpatterns = [
    . . .
    path('captcha/', include('captcha.urls')),
]
```

> ***На заметку***
>
> Для своих нужд приложение `captcha` создает в базе данных таблицу `captcha_captchastore`. Она хранит сведения о сгенерированных на данный момент CAPTCHA, включая временну́ю отметку их устаревания.

17.4.2. Использование Django Simple Captcha

В форме, в которой должна присутствовать CAPTCHA, следует объявить поле типа `CaptchaField` из модуля `captcha.fields`. Это может быть как форма, связанная с моделью:

```
from django import forms
from captcha.fields import CaptchaField

class CommentForm(forms.ModelForm):
    . . .
    captcha = CaptchaField()
    class Meta:
        model = Comment
```

так и несвязанная форма:

```
class SomeForm(forms.Form):
    . . .
    captcha = CaptchaField(label='Введите текст с картинки',
                           error_messages={'invalid': 'Неправильный текст'})
```

На веб-странице элемент управления, представляющий CAPTCHA, выглядит так, как показано на рис. 17.1. Если в параметре `label` конструктора не была указана надпись для него, то он получит надпись по умолчанию — **Captcha**.

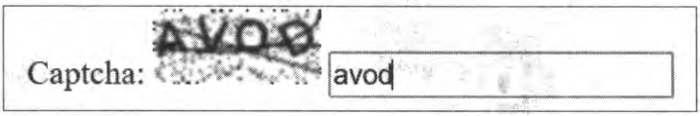


Рис. 17.1. CAPTCHA на веб-странице

Проверка правильности ввода CAPTCHA будет выполняться при валидации формы. Если был введен неправильный текст, форма не пройдет валидацию и будет повторно выведена на экран с указанием сообщения об ошибке.

Как было продемонстрировано в примере ранее, поле `CaptchaField` поддерживает параметры, единые для всех типов полей (см. *разд. 13.1.3.2*), и, кроме того, еще два:

- `generator` — полное имя функции, генерирующей текст для CAPTCHA, в виде строки. В библиотеке доступны следующие функции:
  - `captcha.helpers.random_char_challenge` — классическая CAPTCHA в виде случайного набора из четырех букв. Нечувствительна к регистру;
  - `captcha.helpers.math_challenge` — математическая CAPTCHA, в которой посетителю нужно вычислить результат арифметического выражения и ввести получившийся результат;
  - `captcha.helpers.word_challenge` — словарная CAPTCHA, представляющая собой случайно выбранное слово из заданного словаря (как задать этот словарь, будет рассказано в *разд. 17.4.3*).

  Пример:

```
class CommentForm(forms.ModelForm):
    . . .
    captcha = CaptchaField(generator='captcha.helpers.math_challenge')
    class Meta:
        model = Comment
```

  Значение параметра по умолчанию берется из параметра `CAPTCHA_CHALLENGE_FUNCT` (см. *разд. 17.4.3*);
- `id_prefix` — строковый префикс, добавляемый к якорю, который указывается в атрибутах `id` тегов, формирующих элементы управления для ввода CAPTCHA. Необходим, если в одной форме нужно вывести несколько CAPTCHA. По умолчанию — `None` (префикс отсутствует).

17.4.3. Настройка Django Simple Captcha

Параметры библиотеки указываются в настройках проекта — в модуле settings.py пакета конфигурации. Далее перечислены наиболее полезные из них (полный список параметров приведен в документации по библиотеке):

- `CAPTCHA_CHALLENGE_FUNCT` — полное имя функции, генерирующей текст для CAPTCHA, в виде строки. Поддерживаемые функции перечислены в *разд. 17.4.2*, в описании параметра `generator` поля `CaptchaField`. По умолчанию: `"captcha.helpers.random_char_challenge"`;

- `CAPTCHA_LENGTH` — длина CAPTCHA в символах текста. Принимается во внимание только при использовании классической CAPTCHA. По умолчанию: `4`;
- `CAPTCHA_MATH_CHALLENGE_OPERATOR` — строка с символом, обозначающим оператор умножения. Принимается во внимание только при использовании математической CAPTCHA. По умолчанию: `"*"`. Пример указания крестика в качестве оператора умножения:

  ```
  CAPTCHA_MATH_CHALLENGE_OPERATOR = 'x'
  ```

- `CAPTCHA_WORDS_DICTIONARY` — полный путь к файлу со словарем, используемым в случае выбора словарной CAPTCHA. Словарь должен представлять собой текстовый файл, в котором каждое слово находится на отдельной строке;
- `CAPTCHA_DICTIONARY_MIN_LENGTH` — минимальная длина слова, взятого из словаря, в символах. Применяется в случае выбора словарной CAPTCHA. По умолчанию: `0`;
- `CAPTCHA_DICTIONARY_MAX_LENGTH` — максимальная длина слова, взятого из словаря, в символах. Применяется в случае выбора словарной CAPTCHA. По умолчанию: `99`;
- `CAPTCHA_TIMEOUT` — промежуток времени в минутах, в течение которого сгенерированная CAPTCHA останется действительной. По умолчанию: `5`;
- `CAPTCHA_FONT_PATH` — полный путь к файлу шрифта, используемого для вывода текста. По умолчанию — путь `"<папка, в которой установлен Python>\Lib\site-packages\captcha\fonts\Vera.ttf"` (шрифт Vera, хранящийся в файле по этому пути, является свободным для распространения).

  Также можно указать последовательность путей к шрифтам — в этом случае шрифты будут выбираться случайным образом;
- `CAPTCHA_FONT_SIZE` — кегль шрифта текста в пикселах. По умолчанию: `22`;
- `CAPTCHA_LETTER_ROTATION` — диапазон углов поворота букв в тексте CAPTCHA в виде кортежа, элементы которого укажут предельные углы поворота в градусах. По умолчанию: `(-35, 35)`;
- `CAPTCHA_FOREGROUND_COLOR` — цвет текста на изображении CAPTCHA в любом формате, поддерживаемом CSS. По умолчанию: `"#001100"` (очень темный, практически черный цвет);
- `CAPTCHA_BACKGROUND_COLOR` — цвет фона изображения CAPTCHA в любом формате, поддерживаемом CSS. По умолчанию: `"#ffffff"` (белый цвет);
- `CAPTCHA_IMAGE_SIZE` — геометрические размеры изображения в виде кортежа, первым элементом которого должна быть ширина, вторым — высота. Размеры исчисляются в пикселах. Если указать `None`, то размер изображения будет устанавливаться самой библиотекой. По умолчанию — `None`.

Элемент управления для ввода CAPTCHA выводится согласно шаблону, хранящемуся по пути *<папка, в которой установлен Python>*\Lib\site-packages\captcha\templates\captcha\widgets\captcha.html. Мы можем сделать копию этого шаблона, поместив ее

в папке templates\captcha\widgets пакета приложения, и исправить соответственно своим нуждам. После этого CAPTCHA на странице будет выводиться с применением исправленного шаблона.

17.4.4. Дополнительные команды *captcha_clean* и *captcha_create_pool*

Библиотека Django Simple Captcha добавляет утилите manage.py поддержку двух дополнительных команд.

Команда `captcha_clean` удаляет из хранилища устаревшие CAPTCHA. Ее формат очень прост:

```
manage.py captcha_clean
```

Команда `captcha_create_pool` создает набор готовых CAPTCHA для дальнейшего использования, что позволит потом сэкономить время и системные ресурсы на их вывод. Формат команды:

```
manage.py captcha_create_pool
[--pool-size <количество создаваемых CAPTCHA>] [--cleanup-expired]
```

Дополнительные ключи:

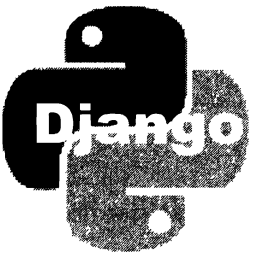
- `--pool-size` — задает количество предварительно создаваемых CAPTCHA. Если не указан, будет создано 1000 CAPTCHA;
- `--cleanup-expired` — заодно удаляет из хранилища устаревшие CAPTCHA.

17.5. Дополнительные настройки проекта, имеющие отношение к формам

Осталось рассмотреть пару параметров, указываемых в настройках проекта и влияющих на обработку форм:

- `DATA_UPLOAD_MAX_MEMORY_SIZE` — максимально допустимый объем полученных от посетителя данных в виде числа в байтах. Если этот объем был превышен, то генерируется исключение `SuspiciousOperation` из модуля `django.core.exceptions`. Если указать значение `None`, то проверка на превышение допустимого объема выполняться не будет. По умолчанию: `2621440` (2,5 Мбайт);
- `DATA_UPLOAD_MAX_NUMBER_FIELDS` — максимально допустимое количество POST-параметров в полученном запросе (т. е. полей в выведенной форме). Если это количество было превышено, то генерируется исключение `SuspiciousOperation`. Если указать значение `None`, то проверка на превышение допустимого количества значений выполняться не будет. По умолчанию: `1000`.

Ограничения, налагаемые этими параметрами, предусмотрены для предотвращения сетевых атак *DOS* (Denial Of Service, отказ от обслуживания). Увеличивать значения параметров следует, только если сайт должен обрабатывать данные большого объема.



ГЛАВА 18

Поддержка баз данных PostgreSQL и библиотека django-localflavor

Django предоставляет расширенные средства для работы с СУБД PostgreSQL. Помимо этого, существует библиотека django-localflavor, предоставляющая дополнительные поля моделей, форм и элементы управления.

18.1. Дополнительные инструменты для поддержки PostgreSQL

Эти инструменты реализованы в приложении `django.contrib.postgres`, поставляемом в составе фреймворка. Чтобы они успешно работали, приложение следует добавить в список зарегистрированных в проекте (параметр `INSTALLED_APPS` настроек проекта — см. *разд. 3.3.3*):

```
INSTALLED_APPS = [
    . . .
    'django.contrib.postgres',
]
```

> ***Внимание!***
>
> **Полная русскоязычная документация по PostgreSQL 12 находится по интернет-адресу https://postgrespro.ru/docs/postgresql/12/index.**

18.1.1. Объявление моделей для работы с PostgreSQL

18.1.1.1. Поля, специфические для PostgreSQL

Классы всех этих полей объявлены в модуле `django.contrib.postgres.fields`:

- `IntegerRangeField` — *поле диапазона*, хранящее диапазон целочисленных значений обычной длины (32-разрядное) в виде его начального и конечного значений.
- `BigIntegerRangeField` — поле диапазона, хранящее диапазон целочисленных значений двойной длины (64-разрядное).

- `DecimalRangeField` (начиная с Django 2.2) — поле диапазона, хранящее диапазон чисел фиксированной точности в виде объектов типа `Decimal` из модуля `decimal` Python.
- `DateRangeField` — поле диапазона, хранящее диапазон значений даты в виде объектов типа `date` из модуля `datetime`.
- `DateTimeRangeField` — поле, хранящее диапазон временны́х отметок в виде объектов типа `datetime` из модуля `datetime`.

  Пример объявления модели `PGSRoomReserving` с полем типа `DateTimeRangeField`:

```
from django.contrib.postgres.fields import DateTimeRangeField

class PGSRoomReserving(models.Model):
    name = models.CharField(max_length=20, verbose_name='Помещение')
    reserving = DateTimeRangeField(
                            verbose_name='Время резервирования')
    cancelled = models.BooleanField(default=False,
                        verbose_name='Отменить резервирование')
    . . .
```

- `ArrayField` — *поле списка*, хранящее список Python, элементы которого обязательно должны принадлежать одному типу. Дополнительные параметры:
  - `base_field` — тип элементов, сохраняемых в поле списков. Указывается в виде объекта (не класса!) соответствующего поля модели;
  - `size` — максимальный размер сохраняемых в поле списков в виде целого числа. При попытке записать в поле список большего размера возникнет ошибка. По умолчанию — `None` (максимальный размер списков не ограничен).

  Пример объявления в модели рубрик `PGSRubric` поля списка `tags`, хранящего строковые величины:

```
from django.contrib.postgres.fields import ArrayField

class PGSRubric(models.Model):
    name = models.CharField(max_length=20, verbose_name='Имя')
    description = models.TextField(verbose_name='Описание')
    tags = ArrayField(base_field=models.CharField(max_length=20),
                                                verbose_name='Теги')
    . . .
```

  Пример объявления в модели `PGSProject` поля списка `platforms`, хранящего поля списка, которые, в свою очередь, хранят строковые значения (фактически создается поле, способное хранить двумерные списки):

```
class PGSProject(models.Model):
    name = models.CharField(max_length=40, verbose_name='Название')
    platforms = ArrayField(base_field=ArrayField(
```

```
        models.CharField(max_length=20)),
        verbose_name='Использованные платформы')
    . . .
```

- `HStoreField` — *поле словаря*, хранящее словарь Python. Ключи элементов такого словаря должны быть строковыми, а значения могут быть строками или `None`.

> ***Внимание!***
> Для успешной работы поля `HStoreField` следует добавить в базу данных расширение `hstore`. Как добавить в базу данных PostgreSQL расширение, будет рассказано позже.

Пример объявления в модели `PGSProject2` поля словаря `platforms`:

```
from django.contrib.postgres.fields import HStoreField

class PGSProject2(models.Model):
    name = models.CharField(max_length=40, verbose_name='Название')
    platforms = HStoreField(verbose_name='Использованные платформы')
    . . .
```

- `JSONField` — поле, хранящее данные в формате JSON.
- `CICharField` — то же самое, что и `CharField` (см. *разд. 4.2.2*), только при выполнении поиска по этому полю не учитывается регистр символов:

```
from django.contrib.postgres.fields import CICharField, JSONField

class PGSProject3(models.Model):
    name = CICharField(max_length=40, verbose_name='Название')
    data = JSONField()
```

- `CITextField` — то же самое, что и `TextField`, только при выполнении поиска по этому полю не учитывается регистр символов.
- `CIEmailField` — то же самое, что и `EmailField`, только при выполнении поиска по этому полю не учитывается регистр символов.

> ***Внимание!***
> Чтобы поля `CICharField`, `CITextField` и `CIEmailField` успешно работали, в базу данных следует добавить расширение `citext`.

18.1.1.2. Индексы PostgreSQL

Индекс, создаваемый указанием в конструкторе поля параметров `unique`, `db_index`, `primary_key` (см. *разд. 4.2.1*) или посредством класса `Index` (см. *разд. 4.4*), получит тип B-Tree, подходящий для большинства случаев.

Начиная с Django 2.2, при создании индексов посредством класса `Index` можно указать у индексируемых полей классы операторов PostgreSQL. Для этого применяется необязательный параметр `opclasses` конструктора класса. Ему нужно присвоить последовательность имен классов операторов, представленных строками: первый класс оператора будет применен к первому индексированному полю, второй — ко второму полю и т. д.

Пример создания индекса по полям `name` и `description` с указанием у поля `name` класса оператора `varchar_pattern_ops`, а у поля `description` — класса оператора `bpchar_pattern_ops`:

```
class PGSRubric(models.Model):
    . . .
    class Meta:
        indexes = [
            models.Index(fields=('name', 'description'),
                         name='i_pgsrubric_name_description',
                         opclasses=('varchar_pattern_ops',
                                    'bpchar_pattern_ops'))
        ]
```

Для создания индексов других типов, поддерживаемых PostgreSQL, следует применять следующие классы из модуля `django.contrib.postgres.indexes`:

- `BTreeIndex` — индекс формата B-Tree.

  Дополнительный параметр `fillfactor` указывает степень заполнения индекса в процентах в виде целого числа от 10 до 100. По умолчанию — `None` (90%).

- `GistIndex` — индекс формата GiST. Дополнительные параметры:

  - `buffering` — если `True`, то при построении индекса будет включен механизм буферизации, если `False` — не будет включен, если `None` — включать или не включать буферизацию, решает СУБД (по умолчанию — `None`);
  - `fillfactor` — степень заполнения индекса в процентах в виде целого числа от 10 до 100. По умолчанию — `None` (90%).

  Пример создания такого индекса по полю `reserving` модели `PGSRoomReserving` с применением класса оператора `range_ops`:

```
from django.contrib.postgres.indexes import GistIndex

class PGSRoomReserving(models.Model):
    . . .
    class Meta:
        indexes = [
            GistIndex(fields=['reserving'], name='i_pgsrr_reserving',
                opclasses=('range_ops',), fillfactor=50)
        ]
```

  Внимание!

  При индексировании таким индексом полей, отличных от `IntegerRangeField`, `BigIntegerRangeField`, `DecimalRangeField`, `DateTimeRangeField` **и** `DateRangeField`, **следует установить в базе данных расширение** `btree_gist`.

- `SpGistIndex` (начиная с Django 2.2) — индекс формата SP-GiST.

  Дополнительный параметр `fillfactor` указывает степень заполнения индекса в процентах в виде целого числа от 10 до 100. По умолчанию — `None` (90%).

- `HashIndex` (начиная с Django 2.2) — индекс на основе хэшей.

  Дополнительный параметр `fillfactor` указывает степень заполнения индекса в процентах в виде целого числа от 10 до 100. По умолчанию — `None` (90%).

- `BrinIndex` — индекс формата BRIN. Дополнительные параметры:
  - `autosummarize` — если `True`, то СУБД будет подсчитывать и сохранять в индексе итоговую информацию по каждому блоку записей, если `False` или `None` — не будет делать этого (по умолчанию — `None`). Такая итоговая информация позволяет ускорить фильтрацию и сортировку, однако увеличивает объем индекса;
  - `pages_per_range` — количество страниц в блоке записей, по которому будет подсчитываться итоговая информация, в виде целого числа. По умолчанию — `None` (128 страниц).
- `GinIndex` — индекс формата GIN. Дополнительные параметры:
  - `fastupdate` — если `True` или `None`, то будет задействован механизм быстрого обновления индекса, при котором обновляемые записи сначала добавляются во временный список, а уже потом, при выполнении обслуживания базы данных, переносятся непосредственно в индекс. Если `False`, то механизм быстрого обновления будет отключен. По умолчанию — `None`;
  - `gin_pending_list_limit` — объем временного списка обновляемых записей в виде целого числа в байтах. По умолчанию — `None` (4 Мбайт).

Внимание!

При индексировании таким индексом полей, отличных от `ArrayField` и `JSONField`, следует установить в базе данных расширение `btree_gin`.

18.1.1.3. Специфическое условие PostgreSQL

В параметре `constraints` модели (см. *разд. 4.4*) можно указать условие `ExclusionConstraint` из модуля `django.contrib.postgres.constraints`, поддержка которого появилась в Django 3.0. Оно указывает критерий, которому НЕ должны удовлетворять записи, добавляемые в модель. Если запись удовлетворяет этим критериям, то она не будет добавлена в модель, и сгенерируется исключение `IntegrityError` из модуля `django.db`.

Внимание!

Для успешной работы этого условия следует установить в базе данных расширение `btree_gist`.

Формат конструктора класса `ExclusionConstraint`:

```
ExclusionConstraint(name=<имя условия>, expressions=<критерии условия>[,
                    index_type=None][, condition=None])
```

Имя условия задается в виде строки и должно быть уникальным в пределах проекта.

Критерии условия указываются в виде списка или кортежа, каждый элемент которого задает один критерий и должен представлять собой список или кортеж из двух элементов:

- имени поля в виде строки или экземпляра класса `F` (см. *разд. 7.3.8*). Значение этого поля из добавляемой записи будет сравниваться со значениями полей из записей, уже имеющихся в базе данных;
- оператора сравнения, поддерживаемого языком SQL, в виде строки. Посредством этого оператора будет выполняться сравнение значений поля, заданного в первом элементе.

  Для задания операторов сравнения также можно применять следующие атрибуты класса `RangeOperators` из модуля `django.contrib.postgres.fields`:

  - `EQUAL` — `=` ("равно");
  - `NOT_EQUAL` — `<>` ("не равно");
  - `CONTAINS` — `@>` ("содержит" — диапазон из имеющейся записи содержит диапазон или значение из добавляемой записи);
  - `CONTAINED_BY` — `<@` ("содержится" — диапазон или значение из имеющейся записи содержится в диапазоне из добавляемой записи);
  - `OVERLAPS` — `@@` ("пересекаются" — оба диапазона пересекаются);
  - `FULLY_LT` — `<<` ("строго слева" — диапазон из имеющейся записи меньше, или находится левее, диапазона из добавляемой записи);
  - `FULLY_GT` — `>>` ("строго справа" — диапазон из имеющейся записи больше, или находится правее, диапазона из добавляемой записи);
  - `NOT_LT` — `&>` ("не левее" — все значения диапазона из имеющейся записи меньше нижней границы диапазона из добавляемой записи);
  - `NOT-GT` — `&<` ("не правее" — все значения диапазона из имеющейся записи больше верхней границы диапазона из добавляемой записи);
  - `ADJACENT_TO` — `-|-` ("примыкают" — диапазоны примыкают друг к другу, т. е. имеют общую границу).

Параметр `index_type` указывает тип создаваемого индекса в виде строки `'GIST'` (GiST) или `'SPGIST'` (SP-GiST). Если не указан, то будет создан индекс GiST.

Параметр `condition` позволяет задать критерий фильтрации, ограничивающий набор записей, к которым будет применяться создаваемое условие. Критерий указывается в виде экземпляра класса `Q` (см. *разд. 7.3.9*). Если не задан, то условие будет применяться ко всем записям в таблице.

Пример создания в модели `PGSRubric` условия, запрещающего добавлять в базу рубрики с совпадающими названиями и описаниями:

```
from django.contrib.postgres.constraints import ExclusionConstraint
from django.contrib.postgres.fields import RangeOperators
```

```
class PGSRubric(models.Model):
    . . .
    class Meta:
        . . .
        constraints = [
            ExclusionConstraint(name='c_pgsrubric_name_description',
                expressions=[('name', RangeOperators.EQUAL),
                             ('description', RangeOperators.EQUAL)])
        ]
```

Пример создания в модели `PGSRoomReserving` условия, запрещающего дважды резервировать одно и то же помещение на тот же или частично совпадающий промежуток времени, но не мешающее отменить резервирование помещения:

```
class PGSRoomReserving(models.Model):
    . . .
    class Meta:
        . . .
        constraints = [
            ExclusionConstraint(name='c_pgsrr_reserving',
                expressions=[('name', RangeOperators.EQUAL),
                             ('reserving', RangeOperators.OVERLAPS)],
                condition=models.Q(cancelled=False))
        ]
```

18.1.1.4. Расширения PostgreSQL

Расширение PostgreSQL — это эквивалент библиотеки Python. Оно упаковано в компактный пакет, включает в себя объявление новых типов полей, индексов, операторов, функций и устанавливается в базе данных подачей специальной SQL-команды.

Установка расширений PostgreSQL выполняется в миграциях. Класс миграции `Migration`, объявленный в модуле каждой миграции, содержит атрибут `operations`, хранящий последовательность операций, которые будут выполнены с базой данных при осуществлении миграции. В начале этой последовательности и записываются выражения, устанавливающие расширения:

```
class Migration(migrations.Migration):
    . . .
    operations = [
        # Выражения, устанавливающие расширения, пишутся здесь
        migrations.CreateModel( . . . ),
        . . .
    ]
```

Эти выражения должны создавать экземпляры классов, представляющих миграции. Все эти классы, объявленные в модуле `django.contrib.postgres.operations`, перечислены далее:

- `CreateExtension` — устанавливает в базе данных расширение с заданным в виде строки *именем*:

```
CreateExtension(name=<имя расширения>)
```

Пример установки расширения `hstore`:

```
from django.contrib.postgres.operations import CreateExtension

class Migration(migrations.Migration):
    . . .
    operations = [
        CreateExtension(name='hstore'),
        migrations.CreateModel( . . . ),
        . . .
    ]
```

- `BtreeGinExtension` — устанавливает расширение `btree_gin`;
- `BtreeGistExtension` — устанавливает расширение `btree_gist`;
- `CITextExtension` — устанавливает расширение `citext`. Пример:

```
from django.contrib.postgres.operations import BtreeGistExtension, \
                                               CITextExtension

class Migration(migrations.Migration):
    . . .
    operations = [
        BtreeGistExtension(),
        CITextExtension(),
        . . .
    ]
```

- `CryptoExtension` — устанавливает расширение `pgcrypto`;
- `HStoreExtension` — устанавливает расширение `hstore`;
- `TrigramExtension` — устанавливает расширение `pg_trgm`;
- `UnaccentExtension` — устанавливает расширение `unaccent`.

Расширения можно добавить как в обычной миграции, вносящей изменения в базу данных, так и в "пустой" миграции (создание "пустой" миграции описано в *разд. 5.1*).

18.1.1.5. Валидаторы PostgreSQL

Специфические для PostgreSQL валидаторы, объявленные в модуле `django.contrib.postgres.validators`, перечислены далее:

- `RangeMinValueValidator` — проверяет, не меньше ли нижняя граница диапазона, сохраняемого в поле диапазона, заданной в параметре `limit_value` величины. Формат конструктора:

```
RangeMinValueValidator(limit_value=<минимальная нижняя граница>[,
                                   message=None])
```

Параметр `message` задает сообщение об ошибке; если он не указан, то используется стандартное. Код ошибки: `"min_value"`.

Начиная с Django 2.2, в качестве значения параметра `limit_value` можно указать функцию, не принимающую параметров и возвращающую минимальную нижнюю границу;

- `RangeMaxValueValidator` — проверяет, не превышает ли верхняя граница диапазона, сохраняемого в поле диапазона, заданную в параметре `limit_value` величину. Формат конструктора:

```
RangeMaxValueValidator(limit_value=<максимальная верхняя граница>[,
                                   message=None])
```

Параметр `message` задает сообщение об ошибке; если он не указан, используется стандартное. Код ошибки: `"max_value"`.

Начиная с Django 2.2, в качестве значения параметра `limit_value` можно указать функцию, не принимающую параметров и возвращающую максимальную верхнюю границу.

Пример использования валидаторов `RangeMinValueValidator` и `RangeMaxValueValidator`:

```
from django.contrib.postgres.validators import \
                    RangeMinValueValidator, RangeMaxValueValidator
from datetime import datetime

class PGSRoomReserving(models.Model):
    . . .
    reserving = DateTimeRangeField(
        verbose_name='Время резервирования',
        validators=[
            RangeMinValueValidator(limit_value=datetime(2000, 1, 1)),
            RangeMaxValueValidator(limit_value=datetime(3000, 1, 1)),
        ])
    . . .
```

- `ArrayMinLengthValidator` — проверяет, не меньше ли размер заносимого списка заданного в первом параметре минимума. Формат конструктора:

```
ArrayMinLengthValidator(<минимальный размер>[, message=None])
```

Параметр `message` задает сообщение об ошибке — если он не указан, то выдается стандартное сообщение. Код ошибки: `"min_length"`.

Начиная с Django 2.2, в качестве значения первого параметра можно указать функцию, не принимающую параметров и возвращающую минимальный размер списка в виде целого числа;

- `ArrayMaxLengthValidator` — проверяет, не превышает ли размер заносимого списка заданный в первом параметре максимум. Используется полем типа `ArrayField` с указанным параметром `size`. Формат конструктора:

  ```
  ArrayMaxLengthValidator(<максимальный размер>[, message=None])
  ```

  Параметр `message` задает сообщение об ошибке — если он не указан, то используется стандартное. Код ошибки: `"max_length"`.

  Начиная с Django 2.2, в качестве значения первого параметра можно указать функцию, не принимающую параметров и возвращающую максимальный размер списка в виде целого числа;

- `KeysValidator` — проверяет, содержит ли словарь, записанный в поле словаря, элементы с заданными ключами. Формат конструктора:

  ```
  KeysValidator(<ключи>[, strict=False][, messages=None])
  ```

  Первым параметром указывается последовательность строк с ключами, наличие которых в словаре следует проверить.

  Если параметру `strict` дать значение `True`, то будет выполняться проверка, не присутствуют ли в словаре какие-либо еще ключи, помимо перечисленных в первом параметре. Если значение параметра `strict` равно `False` или этот параметр вообще не указан, то такая проверка проводиться не будет.

  Параметру `message`, задающему сообщения об ошибках, следует присвоить словарь с элементами `missing_keys` и `extra_keys`; первый задаст сообщение об отсутствии в словаре ключей, перечисленных в первом параметре, а второй — сообщение о наличии в словаре "лишних" ключей. Если сообщения об ошибках не заданы, будут выводиться сообщения по умолчанию.

  Коды ошибки: `"missing_keys"` — при отсутствии в словаре требуемых элементов, `"extra_keys"` — при наличии в словаре "лишних" элементов.

  Пример проверки наличия в сохраняемом в поле `platforms` словаре элемента с ключом `client`:

  ```
  from django.contrib.postgres.validators import KeysValidator

  class PGSProject2(models.Model):
      . . .
      platforms = HStoreField(verbose_name='Использованные платформы',
                              validators=[KeysValidator(('client',))])
      . . .
  ```

  Пример проверки наличия в сохраняемом в поле `platforms` словаре элементов с ключами `client`, `server` и отсутствия там других элементов:

  ```
  class PGSProject2(models.Model):
      . . .
      platforms = HStoreField(verbose_name='Использованные платформы',
          validators=[KeysValidator(('client', 'server'), strict=True)])
      . . .
  ```

18.1.2. Запись и выборка данных в PostgreSQL

18.1.2.1. Запись и выборка значений полей в PostgreSQL

Запись и выборка отдельных значений из полей типов, специфичных для PostgreSQL, имеет следующие особенности:

- поля диапазона — сохраняемый диапазон должен представлять собой список или кортеж из двух элементов — нижней и верхней границы диапазона. Если в качестве одной из границ указать `None`, то диапазон не будет ограничен с соответствующей стороны. Примеры:

```
>>> from testapp.models import PGSRoomReserving
>>> from datetime import datetime
>>> rr = PGSRoomReserving()
>>> rr1.name = 'Большой зал'
>>> rr1.reserving = (datetime(2019, 12, 31, 12),
                     datetime(2019, 12, 31, 16))
>>> rr1.save()
>>> rr2 = PGSRoomReserving()
>>> rr2.name = 'Малый зал'
>>> rr2.reserving = (datetime(2019, 12, 30, 10, 15),
                     datetime(2019, 12, 30, 11, 20))
>>> rr2.save()
```

Заносимый в поле диапазон также может быть представлен экземпляром класса `NumericRange` (для поля типа `IntegerRangeField`, `BigIntegerRangeField` и `DecimalRangeField`), `DateRange` (для поля `DateRangeField`) или `DateTimeTZRange` (для поля `DateTimeRangeField`). Все эти классы объявлены в модуле `psycopg2.extras`. Формат их конструкторов:

```
<Класс диапазона>([lower=None][,][ upper=None][,][ bounds='[)'])
```

Параметры `lower` и `upper` задают соответственно нижнюю и верхнюю границу диапазона. Если какой-либо из этих параметров не указан, то соответствующая граница получит значение `None`, и диапазон не будет ограничен с этой стороны.

Параметр `bounds` указывает, будут нижняя и верхняя границы входить в диапазон или нет. Его значением должна быть строка из двух символов, первый из которых задает вхождение или невхождение в диапазон нижней границы, второй — верхней. Символ `[` обозначает вхождение соответствующей границы в диапазон, символ `(` — невхождение.

Пример занесения в поле диапазона, у которого обе границы входят в диапазон:

```
>>> from psycopg2.extras import DateTimeTZRange
>>> PGSRoomReserving.objects.create(name='Столовая',
        reserving=DateTimeTZRange(
                  lower=datetime(2019, 12, 31, 20, 30),
                  upper=datetime(2020, 1, 1, 2), bounds='[]'))
```

Значение диапазона извлекается из поля также в виде экземпляра одного из перечисленных ранее классов диапазонов. Для работы с ним следует применять следующие атрибуты:

- `lower` — нижняя граница диапазона или `None`, если диапазон не ограничен снизу;
- `upper` — верхняя граница диапазона или `None`, если диапазон не ограничен сверху;
- `lower_inf` — `True`, если диапазон не ограничен снизу, и `False` — в противном случае;
- `upper_inf` — `True`, если диапазон не ограничен сверху, и `False` — в противном случае;
- `lower_inc` — `True`, если нижняя граница не входит в диапазон, и `False` — если входит;
- `upper_inc` — `True`, если верхняя граница не входит в диапазон, и `False` — если входит.

Примеры:

```
>>> rr = PGSRoomReserving.objects.get(pk=1)
>>> rr.name
'Большой зал'
>>> rr.reserving
DateTimeTZRange(datetime.datetime(2019, 12, 31, 12, 0, tzinfo=<UTC>),
datetime.datetime(2019, 12, 31, 16, 0, tzinfo=<UTC>), '[)')
>>> rr.reserving.lower
datetime.datetime(2019, 12, 31, 12, 0, tzinfo=<UTC>)
>>> rr.reserving.lower_inc
True
>>> rr.reserving.upper_inc
False
>>> rr = PGSRoomReserving.objects.get(pk=3)
>>> rr.name
'Столовая'
>>> rr.reserving.lower_inc, rr3.reserving.upper_inc
(True, True)
```

- поле списка (`ArrayField`) — сохраняемое значение можно указать в виде списка или кортежа Python:

```
>>> from testapp.models import PGSRubric
>>> r = PGSRubric()
>>> r.name = 'Недвижимость'
>>> r.description = 'Помещения, жилые и нежилые'
>>> r.tags = ('дом', 'дача', 'склад')
>>> r.save()
```

```
>>> PGSRubric.objects.create(name='Транспорт',
                             description='Транспортные средства',
                             tags=('автомобиль', 'мотоцикл'))
>>> PGSRubric.objects.create(name='Дорогие вещи',
                             description='Всякая всячина',
                             tags=('автомобиль', 'дом', 'склад'))

>>> r = PGSRubric.objects.get(pk=1)
>>> r.name
'Недвижимость'
>>> r.tags
['дом', 'дача', 'склад']
>>> r.tags[2]
'склад'
```

- **поле словаря** (`HStoreField`):

```
>>> from testapp.models import PGSProject2
>>> p = PGSProject2()
>>> p.name = 'Сайт магазина'
>>> p.platforms = {'client': 'HTML, CSS, JavaScript'}
>>> p.save()
>>> PGSProject2.objects.create(name='Сайт новостей',
               platforms={'client': 'HTML, CSS, TypeScript, VueJS',
                          'server': 'NodeJS, Sails.js, MongoDB'})
>>> PGSProject2.objects.create(name='Видеочат',
               platforms={'client': 'HTML, CSS, JavaScript',
                          'server': 'NodeJS'})

>>> p = PGSProject2.objects.get(pk=2)
>>> p.name
'Сайт новостей'
>>> p.platforms
{'client': 'HTML, CSS, TypeScript, VueJS',
'server': 'NodeJS, Sails.js, MongoDB'}
>>> p.platforms['server']
'NodeJS, Sails.js, MongoDB'
```

- `JSONField` — **значение может быть числом, строкой, логической величиной, последовательностью, словарем Python или** `None`:

```
>>> from testapp.models import PGSProject3
>>> p = PGSProject3()
>>> p.name = 'Сайт госучреждения'
>>> p.data = {'client': 'HTML, CSS', 'server': 'PHP, MySQL'}
>>> p.save()
>>> p.data = {'client': ('HTML', 'CSS'), 'server': ('PHP', 'MySQL')}
>>> p.save()
```

```
>>> PGSProject3.objects.create(name='Фотогалерея',
                data={'client': ('HTML', 'CSS', 'JavaScript', 'Zurb'),
                      'server': ('Python', 'Django', 'PostgreSQL')})
>>> PGSProject3.objects.create (name='Видеохостинг',
                data={'client': ('VueJS', 'Vuex', 'Zurb'),
                      'server': ('NodeJS', 'MongoDB'),
                      'balancer': 'nginx'})

>>> p = PGSProject3.objects.get(name='Фотогалерея')
>>> p.data
{'client': ['HTML', 'CSS', 'JavaScript', 'Zurb'],
'server': ['Python', 'Django', 'PostgreSQL']}
>>> p.data['server'][1]
'Django'
```

Значения в поля `CICharField`, `CITextField` и `CIEmailField` заносятся в том же виде, что и в аналогичные поля `CharField`, `TextField` и `EmailField` (см. *разд. 4.2.2*).

18.1.2.2. Фильтрация записей в PostgreSQL

Для фильтрации записей по значениям полей специфических типов PostgreSQL предоставляется ряд особых модификаторов:

- поля диапазона:
  - `contains` — хранящийся в поле диапазон должен содержать указанный диапазон:

```
>>> dtr = DateTimeTZRange(lower=datetime(2019, 12, 31, 13),
                          upper=datetime(2019, 12, 31, 14))
>>> PGSRoomReserving.objects.filter(reserving__contains=dtr)
<QuerySet [<PGSRoomReserving: Большой зал>]>
```

  - `contained_by` — хранящийся в поле диапазон должен содержаться в указанном диапазоне:

```
>>> dtr = DateTimeTZRange(lower=datetime(2019, 12, 31, 12),
                          upper=datetime(2020, 1, 1, 12))
>>> PGSRoomReserving.objects.filter(reserving__contained_by=dtr)
<QuerySet [<PGSRoomReserving: Большой зал>,
<PGSRoomReserving: Столовая>]>

>>> dtr = DateTimeTZRange(lower=datetime(2019, 12, 31, 12),
                          upper=datetime(2020, 1, 1, 4))
>>> PGSRoomReserving.objects.filter(reserving__contained_by=dtr)
<QuerySet [<PGSRoomReserving: Большой зал>,
<PGSRoomReserving: Столовая>]>
```

    Модификатор `contained_by` может применяться не только к полям диапазонов, но и к полям типов `IntegerField`, `BigIntegerField`, `FloatField`, `DateField` и `DateTimeField`. В этом случае будет проверяться, входит ли хранящееся в поле значение в указанный диапазон;

- `overlaps` — **хранящийся в поле диапазон должен пересекаться с заданным:**

```
>>> dtr = DateTimeTZRange(lower=datetime(2019, 12, 31, 15),
                          upper=datetime(2019, 12, 31, 19))
>>> PGSRoomReserving.objects.filter(reserving__overlap=dtr)
<QuerySet [<PGSRoomReserving: Большой зал>]>
```

- `fully_lt` — **сохраненный в поле диапазон должен быть меньше (находиться левее) указанного диапазона:**

```
>>> dtr = DateTimeTZRange(lower=datetime(2019, 12, 31, 8),
                          upper=datetime(2019, 12, 31, 9))
>>> PGSRoomReserving.objects.filter(reserving__fully_lt=dtr)
<QuerySet [<PGSRoomReserving: Малый зал>]>
```

- `fully_gt` — **сохраненный в поле диапазон должен быть больше (находиться правее) указанного диапазона:**

```
>>> PGSRoomReserving.objects.filter(reserving__fully_gt=dtr)
<QuerySet [<PGSRoomReserving: Большой зал>,
<PGSRoomReserving: Столовая>]>
```

- `not_lt` — **ни одна точка сохраненного в поле диапазона не должна быть меньше нижней границы указанного диапазона:**

```
>>> dtr = DateTimeTZRange(lower=datetime(2019, 12, 31, 14),
                          upper=datetime(2019, 12, 31, 16))
>>> PGSRoomReserving.objects.filter(reserving__not_lt=dtr)
<QuerySet [<PGSRoomReserving: Столовая>]>
```

- `not_gt` — **ни одна точка сохраненного в поле диапазона не должна быть больше верхней границы указанного диапазона:**

```
>>> PGSRoomReserving.objects.filter(reserving__not_gt=dtr)
<QuerySet [<PGSRoomReserving: Большой зал>,
<PGSRoomReserving: Малый зал>]>
```

- `adjacent_to` — **граничное значение сохраненного в поле диапазона должно совпадать с граничным значением указанного диапазона:**

```
>>> dtr = DateTimeTZRange(lower=datetime(2019, 12, 30, 11, 20),
                          upper=datetime(2019, 12, 31, 12))
>>> PGSRoomReserving.objects.filter(reserving__adjacent_to=dtr)
<QuerySet [<PGSRoomReserving: Большой зал>,
<PGSRoomReserving: Малый зал>]>
```

- `startswith` — **сохраненный в поле диапазон должен иметь указанную нижнюю границу:**

```
>>> from datetime import timezone
>>> PGSRoomReserving.objects.filter(
                    reserving__startswith=datetime(2019, 12, 30,
                                10, 15, 0, 0, timezone.utc))
<QuerySet [<PGSRoomReserving: Малый зал>]>
```

- `startswith` — сохраненный в поле диапазон должен иметь указанную верхнюю границу:

```
>>> PGSRoomReserving.objects.filter(
                    reserving__endswith=datetime(2020, 1, 1,
                                2, 0, 0, 0, timezone.utc))
<QuerySet [<PGSRoomReserving: Столовая>]>
```

❒ `ArrayField`:

- *<индекс элемента>* — из сохраненного в поле списка извлекается элемент с заданным *индексом*, и дальнейшее сравнение выполняется с извлеченным элементом. В качестве *индекса* можно использовать лишь неотрицательные целые числа. Пример:

```
>>> PGSRubric.objects.filter(tags__1='дача')
<QuerySet [<PGSRubric: Недвижимость>]>
```

- *<индекс первого элемента>_<индекс последнего элемента>* — из сохраненного в поле списка берется срез, расположенный между элементами с указанными *индексами*, и дальнейшее сравнение выполняется с полученным срезом. В качестве *индексов* допустимы лишь неотрицательные целые числа. Пример:

```
>>> PGSRubric.objects.filter(tags__0_2=('автомобиль', 'дом'))
<QuerySet [<PGSRubric: Дорогие вещи>]>
>>> PGSRubric.objects.filter(tags__0_2__overlap=('автомобиль',
                                                 'дом'))
<QuerySet [<PGSRubric: Недвижимость>, <PGSRubric: Транспорт>,
<PGSRubric: Дорогие вещи>]>
```

- `contains` — сохраненный в поле список должен содержать все элементы, перечисленные в заданной последовательности:

```
>>> PGSRubric.objects.filter(tags__contains=('автомобиль',
                                             'мотоцикл'))
<QuerySet [<PGSRubric: Транспорт>]>
>>> PGSRubric.objects.filter(tags__contains=('автомобиль',))
<QuerySet [<PGSRubric: Транспорт>]>
>>> PGSRubric.objects.filter(tags__contains=('автомобиль',
                     'склад', 'мотоцикл', 'дом', 'дача'))
<QuerySet []>
```

- `contained_by` — все элементы сохраненного в поле списка должны присутствовать в заданной последовательности:

```
>>> PGSRubric.objects.filter(tags__contained_by=('автомобиль',))
<QuerySet []>
>>> PGSRubric.objects.filter(tags__contained_by=('автомобиль',
                           'склад', 'мотоцикл', 'дом', 'дача'))
<QuerySet [<PGSRubric: Недвижимость>, <PGSRubric: Транспорт>,
<PGSRubric: Дорогие вещи>]>
```

- `overlap` — **сохраненный в поле список должен содержать хотя бы один из элементов, перечисленных в заданной последовательности:**

```
>>> PGSRubric.objects.filter(tags__overlap=('автомобиль',))
<QuerySet [<PGSRubric: Транспорт>, <PGSRubric: Дорогие вещи>]>
>>> PGSRubric.objects.filter(tags__overlap=('автомобиль',
                                            'склад'))
<QuerySet [<PGSRubric: Недвижимость>, <PGSRubric: Транспорт>,
<PGSRubric: Дорогие вещи>]>
```

- `len` — **определяется размер хранящегося в поле списка, и дальнейшее сравнение выполняется с ним:**

```
>>> PGSRubric.objects.filter(tags__len__lt=3)
<QuerySet [<PGSRubric: Транспорт>]>
```

❒ `HStoreField`:

- `<ключ>` — **из сохраненного в поле словаря извлекается элемент с заданным *ключом*, и дальнейшее сравнение выполняется с ним:**

```
>>> PGSProject2.objects.filter(
                platforms__server='NodeJS, Sails.js, MongoDB')
<QuerySet [<PGSProject2: Сайт новостей>]>
>>> PGSProject2.objects.filter(
                platforms__client__icontains='javascript')
<QuerySet [<PGSProject2: Сайт магазина>,
<PGSProject2: Видеочат>]>
```

- `contains` — **сохраненный в поле словарь должен содержать все элементы, перечисленные в заданном словаре:**

```
>>> PGSProject2.objects.filter(
    platforms__contains={'client': 'HTML, CSS, JavaScript'})
<QuerySet [<PGSProject2: Сайт магазина>,
<PGSProject2: Видеочат>]>
>>> PGSProject2.objects.filter(
    platforms__contains={'client': 'HTML, CSS, JavaScript',
                         'server': 'NodeJS'})
<QuerySet [<PGSProject2: Видеочат>]>
```

- `contained_by` — **все элементы сохраненного в поле словаря должны присутствовать в заданном словаре:**

```
>>> PGSProject2.objects.filter(
    platforms__contained_by={'client': 'HTML, CSS, JavaScript'})
<QuerySet [<PGSProject2: Сайт магазина>]>
```

- `has_key` — **сохраненный в поле словарь должен содержать элемент с указанным ключом:**

```
>>> PGSProject2.objects.filter(platforms__has_key='server')
<QuerySet [<PGSProject2: Сайт новостей>,
<PGSProject2: Видеочат>]>
```

- has_any_keys — сохраненный в поле словарь должен содержать хотя бы один элемент с ключом, присутствующим в заданной последовательности:

```
>>> PGSProject2.objects.filter(
                platforms__has_any_keys=('server', 'client'))
<QuerySet [<PGSProject2: Сайт магазина>,
<PGSProject2: Сайт новостей>, <PGSProject2: Видеочат>]>
```

- has_keys — сохраненный в поле словарь должен содержать все элементы с ключами, присутствующими в заданной последовательности:

```
>>> PGSProject2.objects.filter(
                platforms__has_keys=('server', 'client'))
<QuerySet [<PGSProject2: Сайт новостей>,
<PGSProject2: Видеочат>]>
```

- keys — из сохраненного в поле словаря извлекаются ключи всех элементов, из ключей формируется список, и дальнейшее сравнение выполняется с ним:

```
>>> PGSProject2.objects.filter(
                platforms__keys__contains=('server',))
<QuerySet [<PGSProject2: Сайт новостей>,
<PGSProject2: Видеочат>]>
```

- values — из сохраненного в поле словаря извлекаются значения всех элементов, из значений формируется список, и дальнейшее сравнение выполняется с ним:

```
>>> PGSProject2.objects.filter(
                platforms__values__contains=('NodeJS',))
<QuerySet [<PGSProject2: Видеочат>]>
```

- JSONField — можно обращаться к элементам и атрибутам хранящихся в поле объектов, перечисляя индексы, ключи, имена элементов или атрибутов через двойное подчеркивание (__):

```
>>> # Ищем запись, хранящую в поле data словарь с элементом server,
>>> # содержащим список, первый элемент которого хранит слово "Python"
>>> PGSProject3.objects.filter(data__server__0='Python')
<QuerySet [<PGSProject3: Фотогалерея>]>
```

Также поддерживаются модификаторы contains, contained_by, has_key, has_any_keys и has_keys, описанные ранее:

```
>>> # Ищем запись, хранящую в поле data словарь с элементом client,
>>> # значение которого содержит слово "Zurb"
>>> PGSProject3.objects.filter(data__client__contains=('Zurb',))
<QuerySet [<PGSProject3: Фотогалерея>, <PGSProject3: Видеохостинг>]>
>>> # Ищем запись, хранящую в поле data словарь с элементом balancer
>>> PGSProject3.objects.filter(data__has_key='balancer')
<QuerySet [<PGSProject3: Видеохостинг>]>
```

Чтобы найти запись с JSON-данными, в которых отсутствует какой-либо элемент или атрибут, следует воспользоваться модификатором `isnull` со значением `True`:

```
>>> # Ищем записи, в которых словарь из поля data не содержит элемент
>>> # balancer
>>> PGSProject3.objects.filter(data__balancer__isnull=True)
<QuerySet [<PGSProject3: Сайт госучреждения>,
<PGSProject3: Фотогалерея>]>
```

Два следующих специфических модификатора PostgreSQL помогут при фильтрации записей по значениям строковых и текстовых полей:

- `trigram_similar` — хранящееся в поле значение должно быть похоже на заданное.

  Внимание!

  Для использования модификатора `trigram_similar` следует установить расширение `pg_trgm`.

  Пример:

```
>>> PGSRoomReserving.objects.filter(name__trigram_similar='Стловая')
<QuerySet [<PGSRoomReserving: Столовая>]>
```

- `unaccent` — выполняет сравнение хранящегося в поле и заданного значений без учета диакритических символов.

  Внимание!

  Для использования модификатора `unaccent` следует установить расширение `unaccent`.

  Примеры:

```
>>> PGSProject2.objects.create(name='México Travel',
                               platforms={'client': 'HTML, CSS'})
<PGSProject2: México Travel>

>>> PGSProject2.objects.filter(name__unaccent='Mexico Travel')
<QuerySet [<PGSProject2: México Travel>]>
>>> PGSProject2.objects.filter(name__unaccent__icontains='mexico')
<QuerySet [<PGSProject2: México Travel>]>
```

18.1.3. Агрегатные функции PostgreSQL

Все классы специфических агрегатных функций PostgreSQL объявлены в модуле `django.contrib.postgres.aggregates`:

- `StringAgg` — возвращает строку, составленную из значений, извлеченных из указанного *поля* и отделенных друг от друга заданным *разделителем*:

```
StringAgg(<имя поля или выражение>, <разделитель>[, distinct=False][,
                                    filter=None][, ordering=()])
```

Первым параметром указывается *имя поля*, значения которого составят результирующую строку, или *выражение*, представленное экземпляром класса `F` (см. *разд. 7.3.8*). Вторым параметром задается строка с *разделителем*.

Если параметру `distinct` задано значение `False` (по умолчанию), то в результирующую строку войдут все значения заданного *поля или выражения*, если `False` — только уникальные значения. Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q` (см. *разд. 7.3.9*); если он не задан, фильтрация не выполняется.

Параметр `ordering` поддерживается, начиная с Django 2.2. Он указывает последовательность полей, по которым будет выполняться сортировка элементов в результирующей строке. Каждое поле задается в виде строки с именем (по умолчанию сортировка выполняется по возрастанию; чтобы задать сортировку по убыванию, следует предварить имя поля дефисом) или экземпляра класса `F`.

Пример выборки всех названий комнат из модели `PGSRoomReserving` с сортировкой в обратном алфавитном порядке:

```
>>> from django.contrib.postgres.aggregates import StringAgg
>>> PGSRoomReserving.objects.aggregate(rooms=StringAgg('name',
          delimiter=', ', distinct=True, ordering=('-name',)))
{'rooms': 'Столовая, Малый зал, Большой зал'}
```

- `ArrayAgg` — возвращает список из значений, извлеченных из указанного *поля*:

```
ArrayAgg(<имя поля или выражение>[, distinct=False][, filter=None][,
                                                    ordering=()])
```

Первым параметром указывается *имя поля*, значения которого войдут в список, или *выражение*, представленное экземпляром класса `F` (см. *разд. 7.3.8*). Если параметру `distinct` задано значение `False` (по умолчанию), то в результирующий список войдут все значения заданного *поля или выражения*, если `False` — только уникальные значения. Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q` (см. *разд. 7.3.9*); если он не задан, фильтрация не выполняется.

Параметр `ordering` поддерживается, начиная с Django 2.2. Он указывает последовательность полей, по которым будет выполняться сортировка элементов в результирующей последовательности. Каждое поле может быть указано в виде строки с именем (по умолчанию сортировка выполняется по возрастанию; чтобы задать сортировку по убыванию, следует предварить имя поля дефисом) или экземпляра класса `F`.

Пример создания списка с именами рубрик из модели `PGSRubric`, отсортированных в обратном алфавитном порядке:

```
>>> from django.contrib.postgres.aggregates import ArrayAgg
>>> PGSRubric.objects.aggregate(names=ArrayAgg('name',
                                ordering=('-name',)))
{'names': ['Транспорт', 'Недвижимость', 'Дорогие вещи']}
```

- `JSONBAgg` — возвращает значения указанного *поля* в виде JSON-объекта:

  ```
  JSONBAgg(<имя поля или выражение>[, filter=None])
  ```

  Первым параметром указывается *имя поля*, значения которого войдут в объект, или *выражение*, представленное экземпляром класса `F` (см. *разд. 7.3.8*). Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q` (см. *разд. 7.3.9*); если он не задан, фильтрация не выполняется.

  Пример:

  ```
  >>> from django.contrib.postgres.aggregates import JSONBAgg
  >>> PGSProject3.objects.aggregate(result=JSONBAgg('data'))
  {'result': [{'client': ['HTML', 'CSS'], 'server': ['PHP', 'MySQL']},
  {'client': ['HTML', 'CSS', 'JavaScript', 'Zurb'],
  'server': ['Python', 'Django', 'PostgreSQL']},
  {'client': ['VueJS', 'Vuex', 'Zurb'], 'server': ['NodeJS', 'MongoDB'],
  'balancer': 'nginx'}]}
  ```

- `BitAnd` — возвращает результат побитового умножения (И) всех значений заданного поля, отличных от `None`, или `None`, если все значения поля равны `None`:

  ```
  BitAnd(<имя поля или выражение>[, filter=None])
  ```

  Первым параметром указывается *имя поля*, значения которого будут перемножаться, или *выражение*, представленное экземпляром класса `F` (см. *разд. 7.3.8*). Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q` (см. *разд. 7.3.9*); если он не задан, фильтрация не выполняется.

- `BitOr` — возвращает результат побитового сложения (ИЛИ) всех значений заданного поля, отличных от `None`, или `None`, если все значения поля равны `None`:

  ```
  BitOr(<имя поля или выражение>[, filter=None])
  ```

  Параметры указываются в том же формате, что и у функции `BitAnd` (см. ранее).

- `BoolAnd` — возвращает `True`, если все значения заданного поля равны `True`, `None` — если все значения равны `None` или в таблице нет записей, и `False` — в противном случае:

  ```
  BoolAnd(<имя поля или выражение>[, filter=None])
  ```

  Параметры указываются в том же формате, что и у функции `BitAnd` (см. ранее).

- `BoolOr` — возвращает `True`, если хотя бы одно значение заданного поля равно `True`, `None` — если все значения равны `None` или в таблице нет записей, и `False` — в противном случае:

  ```
  BoolOr(<имя поля или выражение>[, filter=None])
  ```

  Параметры указываются в том же формате, что и у функции `BitAnd` (см. ранее).

- `Corr` — возвращает коэффициент корреляции, вычисленный на основе значений из *поля Y* и *поля X*, в виде вещественного числа (тип `float`) или `None`, если в наборе нет ни одной записи:

  ```
  Corr(<имя поля или выражение Y>, <имя поля или выражение X>[, filter=None])
  ```

Первыми двумя параметрами указываются *имена числовых полей* или *выражения*, представленные экземплярами класса `F` (см. *разд. 7.3.8*). Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q` (см. *разд. 7.3.9*); если он не задан, фильтрация не выполняется.

- `CovarPop` — возвращает ковариацию населения, вычисленную на основе значений из *поля Y* и *поля X*, в виде вещественного числа (тип `float`) или `None`, если в наборе нет ни одной записи:

```
CovarPop(<имя поля или выражение Y>, <имя поля или выражение X>[,
                                     sample=False][, filter=None])
```

Первыми двумя параметрами указываются *имена числовых полей* или *выражения*, представленные экземплярами класса `F` (см. *разд. 7.3.8*). Если параметру `sample` дать значение `False`, то будет возвращена выборочная ковариация населения, если дать значение `True` — ковариация населения в целом. Параметр `filter` указывает условие фильтрации записей в виде экземпляра класса `Q` (см. *разд. 7.3.9*); если он не задан, фильтрация не выполняется.

- `RegrAvgX` — возвращает среднее арифметическое, вычисленное на основе значений из *поля X*, в виде вещественного числа (тип `float`) или `None`, если в наборе нет ни одной записи:

```
RegrAvgX(<имя поля или выражение Y>, <имя поля или выражение X>[,
                                     filter=None])
```

Параметры указываются в том же формате, что и у функции `Corr` (см. ранее).

- `RegrAvgY` — возвращает среднее арифметическое, вычисленное на основе значений из *поля Y*, в виде вещественного числа (тип `float`) или `None`, если в наборе нет ни одной записи:

```
RegrAvgY(<имя поля или выражение Y>, <имя поля или выражение X>[,
                                     filter=None])
```

Параметры указываются в том же формате, что и у функции `Corr` (см. ранее).

- `RegrCount` — возвращает количество записей, в которых `поле Y` и `поле X` хранят величины, отличные от `None`, в виде целого числа (тип `int`) или `None`, если в наборе нет ни одной записи:

```
RegrCount(<имя поля или выражение Y>, <имя поля или выражение X>[,
                                     filter=None])
```

Параметры указываются в том же формате, что и у функции `Corr` (см. ранее).

- `RegrIntercept` — возвращает величину точки пересечения оси Y и линии регрессии, рассчитанную методом наименьших квадратов на основе значений *поля Y* и *поля X*, в виде вещественного числа (тип `float`) или `None`, если в наборе нет ни одной записи:

```
RegrIntercept(<имя поля или выражение Y>, <имя поля или выражение X>[,
                                     filter=None])
```

Параметры указываются в том же формате, что и у функции `Corr` (см. ранее).

- `RegrR2` — возвращает квадрат коэффициента корреляции, вычисленный на основе значений из *поля Y* и *поля X*, в виде вещественного числа (тип `float`) или `None`, если в наборе нет ни одной записи:

```
RegrR2(<имя поля или выражение Y>, <имя поля или выражение X>[,
                                   filter=None])
```

  Параметры указываются в том же формате, что и у функции `Corr` (см. ранее).

- `RegrSlope` — возвращает величину наклона линии регрессии, рассчитанную методом наименьших квадратов на основе значений *поля Y* и *поля X*, в виде вещественного числа (тип `float`) или `None`, если в наборе нет ни одной записи:

```
RegrSlope(<имя поля или выражение Y>, <имя поля или выражение X>[,
                                      filter=None])
```

  Параметры указываются в том же формате, что и у функции `Corr` (см. ранее).

- `RegrSXX` — возвращает сумму площадей, рассчитанную на основе значений *поля X*, в виде вещественного числа (тип `float`) или `None`, если в наборе нет ни одной записи:

```
RegrSXX(<имя поля или выражение Y>, <имя поля или выражение X>[,
                                    filter=None])
```

  Параметры указываются в том же формате, что и у функции `Corr` (см. ранее).

- `RegrSXY` — возвращает сумму производных, рассчитанную на основе значений *поля Y* и *поля X*, в виде вещественного числа (тип `float`) или `None`, если в наборе нет ни одной записи:

```
RegrSXY(<имя поля или выражение Y>, <имя поля или выражение X>[,
                                    filter=None])
```

  Параметры указываются в том же формате, что и у функции `Corr` (см. ранее).

- `RegrSYY` — возвращает сумму площадей, рассчитанную на основе значений *поля Y*, в виде вещественного числа (тип `float`) или `None`, если в наборе нет ни одной записи:

```
RegrSYY(<имя поля или выражение Y>, <имя поля или выражение X>[,
                                    filter=None])
```

  Параметры указываются в том же формате, что и у функции `Corr` (см. ранее).

18.1.4. Функции СУБД, специфичные для PostgreSQL

Классы этих функций объявлены в модуле `django.contrib.postgres.functions`:

- `RandomUUID` — возвращает сгенерированный случайным образом уникальный универсальный идентификатор.

> ***Внимание!***
>
> **Для использования функции `RandomUUID` следует установить расширение `pgcrypto`.**

- `TransactionNow` — возвращает временну́ю отметку (дату и время) запуска текущей транзакции или, если транзакция не была запущена, текущую временну́ю отметку.

  Если существует несколько вложенных друг в друга транзакций, будет возвращена временна́я отметка запуска наиболее "внешней" транзакции.

  Пример:

```
from django.contrib.postgres.functions import TransactionNow
Bb.objects.update(uploaded=TransactionNow())
```

18.1.5. Полнотекстовая фильтрация PostgreSQL

Полнотекстовая фильтрация — это фильтрация записей по наличию в строковых или текстовых значениях, сохраненных в их полях, заданных слов. Подобного рода фильтрация используется в поисковых интернет-службах.

18.1.5.1. Модификатор *search*

Модификатор `search` выполняет полнотекстовую фильтрацию по одному полю:

```
>>> # Отбираем записи, у которых в значениях поля name содержится
>>> # слово "зал"
>>> PGSRoomReserving.objects.filter(name__search='зал')
<QuerySet [<PGSRoomReserving: Большой зал>,
<PGSRoomReserving: Малый зал>]>
```

18.1.5.2. Функции СУБД для полнотекстовой фильтрации

Классы этих функций объявлены в модуле `django.contrib.postgres.search`:

- `SearchVector` — задает поля, по которым будет осуществляться фильтрация. Искомое значение должно содержаться, по крайней мере, в одном из заданных полей:

```
SearchVector(<имя поля или выражение 1>,
             <имя поля или выражение 2> . . .
             <имя поля или выражение n>[, config=None][, weight=None])
```

  Можно указать либо *имена полей* в виде строк, либо *выражения* в виде экземпляров класса `F` (см. *разд. 7.3.8*).

  Созданный экземпляр класса `SearchVector` следует указать в вызове метода `annotate()` в именованном параметре, создав тем самым вычисляемое поле (подробности — в *разд. 7.6*). После этого можно выполнить фильтрацию по значению этого поля обычным способом — с помощью метода `filter()` (см. *разд. 7.3.5*).

  Пример:

```
>>> from django.contrib.postgres.search import SearchVector
>>> # Отбираем записи, содержащие в поле name или элементе client
>>> # поля словаря platforms слово "TypeScript"
```

```
>>> cr = SearchVector('name', 'platforms__client')
>>> PGSProject2.objects.annotate(
          search_vector=cr).filter(search_vector='TypeScript')
<QuerySet [<PGSProject2: Сайт новостей>]>
>>> # Аналогично, но ищем слово "javascript"
>>> PGSProject2.objects.annotate(search_vector=cr).filter(
                                  search_vector='javascript')
<QuerySet [<PGSProject2: Сайт магазина>, <PGSProject2: Видеочат>]>
```

Экземпляры класса `SearchVector` также можно объединять оператором `+`:

```
cr = SearchVector('name') + SearchVector('platforms__client')
```

Параметр `config` позволяет указать специфическую конфигурацию поиска в формате PostgreSQL в виде либо строки, либо экземпляра класса `F` с именем поля, в котором хранится эта конфигурация:

```
cr = SearchVector('name', 'platforms__client', config='english')
```

Параметр `weight` задает уровень значимости для поля (полей). Запись, в которой искомое значение присутствует в поле с большей значимостью, станет более релевантной. Уровень значимости указывается в виде предопределенных строковых значений `'A'` (максимальный уровень релевантности), `'B'`, `'C'` или `'D'` (минимальный уровень). Пример:

```
cr = SearchVector('name', weight='A') + \
                 SearchVector('platforms__client', weight='C')
```

- `SearchQuery` — задает искомое значение:

```
SearchQuery(<искомое значение>[, config=None][, search_type='plain'])
```

Параметр `search_type` указывает режим поиска в виде одной из строк:

- `'plain'` — будут отбираться записи, содержащие все слова, из которых состоит *искомое значение*, в произвольном порядке (поведение по умолчанию):

```
>>> from django.contrib.postgres.search import SearchQuery
>>> # Отбираем записи, содержащие в поле name или элементе client
>>> # поля словаря platforms слова "видеочат" и "javascript",
>>> # при этом порядок слов значения не имеет
>>> cr = SearchVector('name', 'platforms__client')
>>> q = SearchQuery('видеочат javascript')
>>> PGSProject2.objects.annotate(search_vector=cr).filter(
                                    search_vector=q)
<QuerySet [<PGSProject2: Видеочат>]>
```

- `'phrase'` — будут отбираться записи, содержащие заданное *искомое значение*:

```
>>> # Отбираем записи, содержащие в поле name или элементе client
>>> # поля словаря platforms фразу "видеочат javascript"
>>> q = SearchQuery('видеочат javascript', search_type='phrase')
```

```
>>> PGSProject2.objects.annotate(search_vector=cr).filter(
                                    search_vector=q)
<QuerySet []>
```

- `'raw'` — *искомое значение* обрабатывается согласно правилам записи логических выражений PostgreSQL. В таком значении строковые величины берутся в одинарные кавычки, используются логические операторы `&` (И), `|` (ИЛИ) и круглые скобки:

```
>>> # Отбираем записи, содержащие в поле name или элементе client
>>> # поля словаря platforms слово "сайт" и либо слово
>>> # "javascript", либо "typescript"
>>> q = SearchQuery("'сайт' & ('javascript' | 'typescript')",
                    search_type='raw')
>>> PGSProject2.objects.annotate(search_vector=cr).filter(
                                    search_vector=q)
<QuerySet [<PGSProject2: Сайт магазина>,
<PGSProject2: Сайт новостей>]>
```

Искомое значение может быть указано в виде комбинации экземпляров класса `SearchQuery`, содержащих одно слово и объединенных логическими операторами `&` (И), `|` (ИЛИ) и `~` (НЕ). Также можно применять круглые скобки. Пример:

```
>>> # Отбираем записи, не содержащие в поле name или элементе
>>> # client поля словаря platforms слово "сайт" и содержащие
>>> # слово "javascript" или "typescript"
>>> q = ~SearchQuery('сайт') & \
        (SearchQuery('javascript') | SearchQuery('typescript'))
>>> PGSProject2.objects.annotate(search_vector=cr).filter(
                                    search_vector=q)
<QuerySet [<PGSProject2: Видеочат>]>
```

Параметр `config` указывает специфическую конфигурацию поиска в формате PostgreSQL.

- `SearchRank` — создает вычисляемое поле релевантности. Релевантность вычисляется на основе того, сколько раз искомое значение присутствует в содержимом записи, насколько близко отдельные слова искомого значения находятся друг к другу и т. п., и представляется в виде вещественного числа от 1.0 (наиболее подходящие записи) до 0.0 (записи, вообще не удовлетворяющие заданным критериям фильтрации).

  Формат конструктора:

```
SearchRank(<поля, по которым выполняется фильтрация>,
           <искомое значение>[, weights=None])
```

Поле, по которому выполняется фильтрация, задается экземпляром класса `SearchVector`, а *искомое значение* — экземпляром класса `SearchQuery`. Примеры:

```
>>> from django.contrib.postgres.search import SearchRank
>>> # Вычисляем релевантность записей, содержащих в поле name или
>>> # элементе client поля словаря platforms слово "магазин" или
>>> # "javascript"
>>> cr = SearchVector('name', 'platforms__client')
>>> q = SearchQuery("'магазин' | 'javascript'", search_type='raw')
>>> result = PGSProject2.objects.annotate(
                        rank=SearchRank(cr, q)).order_by('-rank')
>>> for r in result: print(r.name, ': ', r.rank)
...
Сайт магазина :  0.030396355
Видеочат :  0.030396355
Сайт новостей :  0.0
México Travel :  0.0

>>> # Отбираем лишь релевантные записи
>>> result = PGSProject2.objects.annotate(
            rank=SearchRank(cr, q)).filter(rank__gt=0).order_by('-rank')
>>> for r in result: print(r.name, ': ', r.rank)
...
Сайт магазина :  0.030396355
Видеочат :  0.030396355
```

Параметр `weights` позволит задать другие величины уровней значимости полей для предопределенных значений `'A'`, `'B'`, `'C'` и `'D'` (см. описание класса `SearchVector`). Параметру присваивается список из четырех величин уровней значимости: первая задаст уровень для предопределенного значения `'D'`, вторая — для значения `'C'`, третья и четвертая — для `'B'` и `'A'`. Каждая величина уровня значимости указывается в виде вещественного числа от 0.0 (наименее значимое поле) до 1.0 (максимальная значимость). Пример:

```
cr = SearchVector('name', weight='A') + \
                 SearchVector('platforms__client', weight='C')
result = PGSProject2.objects.annotate(
         rank=SearchRank(cr, q, weights=[0.2, 0.5, 0.8, 1.0])).filter(
         rank__gt=0).order_by('-rank')
```

18.1.5.3. Функции СУБД для фильтрации по похожим словам

Следующие две функции СУБД, классы которых объявлены в модуле `django.contrib.postgres.search`, выполняют фильтрацию записей, которые содержат слова, похожие на заданное.

> ***Внимание!***
>
> **Для использования этих функций следует установить расширение `pg_trgm`.**

- `TrigramSimilarity` — создает вычисляемое поле, хранящее степень похожести слов, присутствующих в содержимом записи, на заданное *искомое значение*.

Степень похожести представляется вещественным числом от 1.0 (точное совпадение) до 0.0 (совпадение отсутствует). Формат конструктора:

```
TrigramSimilarity(<поле, по которому выполняется фильтрация>,
                  <искомое значение>)
```

Поле, по которому выполняется фильтрация, можно указать в виде либо строки с его именем, либо экземпляра класса `F` (см. *разд. 7.3.8*).

Примеры:

```
>>> from django.contrib.postgres.search import TrigramSimilarity
>>> result = PGSProject2.objects.annotate(
                         simil=TrigramSimilarity('name', 'видео'))
>>> for r in result: print(r.name, ': ', r.simil)
...
Сайт магазина :  0.0
Сайт новостей :  0.0
Видеочат :  0.5
México Travel :  0.0

>>> result = PGSProject2.objects.annotate(
             simil=TrigramSimilarity('name', 'видео')).filter(
                                                    simil__gte=0.5)
>>> for r in result: print(r.name, ': ', r.simil)
...
Видеочат :  0.5
```

- `TrigramDistance` — создает вычисляемое поле, хранящее степень расхождения слов, присутствующих в содержимом записи, с заданным *искомым значением*. Степень расхождения представляется вещественным числом от 1.0 (полное расхождение) до 0.0 (расхождение отсутствует). Формат конструктора:

```
TrigramDistance(<поле, по которому выполняется фильтрация>,
                <искомое значение>)
```

Поле, по которому выполняется фильтрация, можно указать в виде либо строки с его именем, либо экземпляра класса `F` (см. *разд. 7.3.8*).

Примеры:

```
>>> from django.contrib.postgres.search import TrigramDistance
>>> result = PGSProject2.objects.annotate(
                         dist=TrigramDistance('name', 'новость'))
>>> for r in result: print(r.name, ': ', r.dist)
...
Сайт магазина :  1.0
Сайт новостей :  0.625
Видеочат :  1.0
México Travel :  1.0

>>> result = PGSProject2.objects.annotate(
             dist=TrigramDistance('name', 'новость')).filter(
                                                   dist__lte=0.7)
```

```
>>> for r in result: print(r.name, ': ', r.dist)
...
Сайт новостей :  0.625
```

18.1.6. Объявление форм для работы с PostgreSQL

18.1.6.1. Поля форм, специфические для PostgreSQL

Все классы перечисленных далее полей объявлены в модуле `django.contrib.postgres.forms`:

- `IntegerRangeField` — поле диапазона, служащее для указания диапазона целочисленных значений.
- `DecimalRangeField` (начиная с Django 2.2) — поле диапазона, служащее для указания диапазона чисел фиксированной точности в виде объектов типа `Decimal` из модуля `decimal` Python.
- `DateRangeField` — поле диапазона, служащее для указания диапазона значений даты в виде объектов типа `date` из модуля `datetime`.
- `DateTimeRangeField` — поле диапазона, служащее для ввода диапазона временны́х отметок в виде объектов типа `datetime` из модуля `datetime`.
- `SimpleArrayField` — поле списка. Выводит элементы списка в одном поле ввода, отделяя их друг от друга заданным разделителем. Дополнительные параметры:
  - `base_field` — тип элементов списков, сохраняемых в поле. Указывается в виде объекта (не класса!) соответствующего поля формы;
  - `delimiter` — строка с разделителем, которым отделяются друг от друга отдельные элементы списка (по умолчанию — запятая);
  - `min_length` — минимальный размер списка в виде целого числа (по умолчанию — не ограничен);
  - `max_length` — максимальный размер списка в виде целого числа (по умолчанию — не ограничен).

  Пример:

```
from django.contrib.postgres.forms import SimpleArrayField

class PGSRubricForm(forms.ModelForm):
    tags = SimpleArrayField(base_field=forms.CharField(max_length=20))
    . . .
```

- `SplitArrayField` — *разделенное поле списка*. Каждый элемент списка выводится в отдельном поле ввода. Дополнительные параметры:
  - `base_field` — тип элементов списков, сохраняемых в поле. Указывается в виде объекта (не класса!) соответствующего поля формы;
  - `size` — количество выводимых на экране элементов списка в виде целого числа. Если превышает количество элементов, имеющихся в списке, то на

экран будут выведены пустые поля ввода. Если меньше количества элементов в списке, то на экран будут выведены все элементы;

- `remove_trailing_nulls` — если `False`, в связанном поле модели будут сохранены все элементы занесенного пользователем списка, даже пустые; если `True` — пустые элементы в конце списка будут удалены (по умолчанию — `False`).

Пример:

```
from django.contrib.postgres.forms import SplitArrayField

class PGSRubricForm2(forms.ModelForm):
    tags = SplitArrayField(
           base_field=forms.CharField(max_length=20), size=4)
```

- `HStoreField` — поле словаря. Выводит словарь в виде исходного кода на языке Python в области редактирования.
- `JSONField` — поле JSON. Выводит содержимое поля в виде его JSON-нотации в области редактирования.

В табл. 18.1 перечислены классы полей модели, специфические для PostgreSQL, и соответствующие им классы полей формы, используемые по умолчанию.

***Таблица 18.1.** Специфические для PostgreSQL классы полей модели и соответствующие им классы полей формы, используемые по умолчанию*

Классы полей модели	Классы полей формы
`IntegerRangeField`	`IntegerRangeField`
`BigIntegerRangeField`	
`DecimalRangeField`	`DecimalRangeField`
`DateRangeField`	`DateRangeField`
`DateTimeRangeField`	`DateTimeRangeField`
`ArrayField`	`SimpleArrayField`
`HStoreField`	`HStoreField`
`JSONField`	`JSONField`
`CICharField`	`CharField`. Параметр `max_length` получает значение от параметра `max_length` конструктора поля модели. Если параметр `null` конструктора поля модели имеет значение `True`, то параметр `empty_value` конструктора поля формы получит значение `None`
`CITextField`	`CharField`, у которого в качестве элемента управления указана область редактирования (параметр `widget` имеет значение `Textarea`)
`CIEmailField`	`EmailField`

18.1.6.2. Элементы управления, специфические для PostgreSQL

Класс `RangeWidget` из модуля `django.contrib.postgres.forms` представляет элемент управления для ввода диапазона значений какого-либо типа. Параметр `base_widget` задает элемент управления, используемый для ввода каждого из граничных значений диапазона, в виде экземпляра класса нужного элемента управления или ссылки на сам этот класс.

Пример указания для ввода граничных значений диапазона, хранящегося в поле `reserving` модели `PGSRoomReserving`, элемента управления `SplitDateTimeWidget`:

```
from django.contrib.postgres.forms import DateTimeRangeField, RangeWidget
from django.forms.widgets import SplitDateTimeWidget

class PGSRRForm(forms.ModelForm):
    reserving = DateTimeRangeField(
                widget=RangeWidget(base_widget=SplitDateTimeWidget))
```

В табл. 18.2 перечислены классы полей формы, специфические для PostgreSQL, и соответствующие им классы элементов управления, используемые по умолчанию.

***Таблица 18.2.** Классы полей формы, специфические для PostgreSQL, и соответствующие им классы элементов управления, используемые по умолчанию*

Классы полей формы	Классы элементов управления
`IntegerRangeField`	`RangeWidget` с полями ввода `NumberInput`
`DecimalRangeField`	
`DateRangeField`	`RangeWidget` с полями ввода `DateInput`
`DateTimeRangeField`	`RangeWidget` с полями ввода `DateTimeInput`
`SimpleArrayField`	Элементы управления, соответствующие типу хранящихся в списке значений
`SplitArrayField`	
`HStoreField`	`TextInput`
`JSONField`	

18.2. Библиотека django-localflavor: дополнительные поля для моделей и форм

Библиотека django-localflavor предоставляет два поля модели, предназначенные для хранения банковских сведений, и несколько классов полей формы, в которые заносятся сведения, специфические для разных стран, включая Россию (коды субъектов федерации, номера паспортов и пр.).

На заметку

Полная документация по библиотеке находится по интернет-адресу **https://django-localflavor.readthedocs.io/en/latest/**.

Здесь будут описаны лишь поля формы и элементы управления, специфические для Российской Федерации. За описанием полей формы и элементов управления для других стран обращайтесь к документации по библиотеке.

18.2.1. Установка django-localflavor

Установка библиотеки выполняется подачей команды:

```
pip install django-localflavor
```

Помимо django-localflavor, будет установлена библиотека python-stdnum, необходимая для работы.

Далее следует включить присутствующее в составе библиотеки приложение `localflavor` в список зарегистрированных в проекте (параметр `INSTALLED_APPS` настроек проекта, подробнее — в *разд. 3.3.3*):

```
INSTALLED_APPS = [
    . . .
    'localflavor',
]
```

18.2.2. Поля модели, предоставляемые django-localflavor

Оба класса полей объявлены в модуле `localflavor.generic.models`:

- `IBANField` — хранит номер международного банковского счета (IBAN) в строковом виде. Дополнительные параметры:
  - `include_countries` — указывает перечень стран, международные банковские номера которых допускается заносить в поле. Значением параметра должен быть кортеж из кодов стран в формате ISO 3166-1 alpha 2, записанных в виде строк (пример: `('GB', 'US', 'FR', 'DE')`). Коды стран в формате ISO 3166-1 alpha 2 можно найти по интернет-адресу **https://ru.wikipedia.org/wiki/ISO_3166-1**.

    Чтобы разрешить заносить в поле коды всех стран, использующих международные номера IBAN, следует присвоить этому параметру значение переменной `IBAN_SEPA_COUNTRIES` из модуля `localflavor.generic.countries.sepa`.

    Значение по умолчанию — `None` (в поле разрешается заносить значения международных банковских номеров из любых стран, даже не использующих международные номера IBAN);
  - `use_nordea_extensions` — если `True`, то в поле также можно сохранять номера счета в банке Nordea, если `False` — нет (по умолчанию — `False`).
- `BICField` — хранит банковский идентификационный код (БИК) в строковом виде.

18.2.3. Поля формы, предоставляемые django-localflavor

Следующие два класса полей объявлены в модуле `localflavor.generic.forms`:

- `IBANFormField` — номер международного банковского счета (IBAN) в виде строки. Поддерживает дополнительные параметры `include_countries` и `use_nordea_extensions` (см. *разд. 18.2.2*). Используется полем модели `IBANField` по умолчанию.
- `BICFormField` — банковский идентификационный код (БИК) в виде строки. Используется полем модели `BICField` по умолчанию.

А эти три класса полей объявлены в модуле `localflavor.ru.forms`:

- `RUPassportNumberField` — номер внутреннего паспорта РФ в виде 11-значной строки формата:

  <серия из четырех цифр> <номер из шести цифр>

- `RUAlienPassportNumberField` — номер заграничного паспорта РФ в виде 10-значной строки формата:

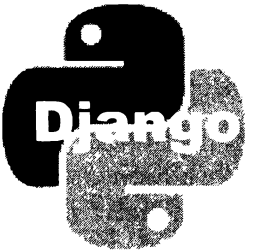
  <серия из двух цифр> <номер из семи цифр>

- `RUPostalCodeField` — почтовый индекс РФ в виде строки из 6 цифр.

18.2.4. Элементы управления, предоставляемые django-localflavor

Следующие классы элементов управления объявлены в модуле `localflavor.ru.forms`:

- `RUCountySelect` — список для выбора федерального округа РФ. Выбранный федеральный округ представляется строкой с его англоязычным названием: "Central Federal County" (Центральный), "South Federal County" (Южный), "North-West Federal County" (Северо-Западный), "Far-East Federal County" (Дальневосточный), "Siberian Federal County" (Сибирский), "Ural Federal County" (Уральский), "Privolzhsky Federal County" (Приволжский), "North-Caucasian Federal County" (Северо-Кавказский).
- `RURegionSelect` — список для выбора субъекта РФ. Выбранный субъект представляется строкой с его двузначным кодом, записанным в Конституции РФ: `'77'` (Москва), `'78'` (Санкт-Петербург), `'34'` (Волгоградская обл.) и т. д.



ГЛАВА 19

Шаблоны: расширенные инструменты и дополнительная библиотека

Существует ряд дополнительных библиотек, расширяющих возможности шаблонизатора Django, добавляющих ему поддержку новых тегов и фильтров. Две таких библиотеки рассматриваются в этой главе. Также будет рассказано, как самостоятельно добавить в шаблонизатор новые теги и фильтры.

19.1. Библиотека django-precise-bbcode: поддержка BBCode

BBCode (Bulletin Board Code, код досок объявлений) — это язык разметки, который используется для форматирования текста на многих форумах и блогах. Форматирование выполняется с помощью тегов, схожих с тегами языка HTML, но заключаемых в квадратные скобки. При выводе такие теги преобразуются в обычный HTML-код.

Обрабатывать BBCode в Django-сайтах удобно с помощью дополнительной библиотеки django-precise-bbcode. Она поддерживает все широко употребляемые теги, позволяет добавить поддержку новых тегов, просто записав их в базу данных сайта, и может выводить графические смайлики. Кроме того, при выводе она заменяет символы перевода строк HTML-тегами `<br>`, в результате чего разбитый на абзацы текст выводится на страницу также разделенным на абзацы, а не в одну строку.

> ***НА ЗАМЕТКУ***
>
> Полную документацию по django-precise-bbcode можно найти здесь: **https://django-precise-bbcode.readthedocs.io/en/stable/index.html**.

19.1.1. Установка django-precise-bbcode

Для установки библиотеки необходимо подать команду:

```
pip install django-precise-bbcode
```

Помимо самой django-precise-bbcode, будет установлена библиотека обработки графики Pillow, которая необходима для вывода графических смайликов.

Перед использованием библиотеки следует выполнить следующие шаги:

- добавить приложение `precise_bbcode` — программное ядро библиотеки — в список зарегистрированных в проекте (параметр `INSTALLED_APPS` настроек проекта, подробнее — в *разд. 3.3.3*):

```
INSTALLED_APPS = [
    . . .
    'precise_bbcode',
]
```

- выполнить миграции.

> ***На заметку***
>
> Для своих нужд приложение `precise_bbcode` создает в базе данных таблицы `precise_bbcode_bbcodetag` и `precise_bbcode_smileytag`. Первая хранит список BBCode-тегов, добавленных разработчиком сайта, а вторая — список смайликов.

19.1.2. Поддерживаемые BBCode-теги

Изначально django-precise-bbcode поддерживает лишь общеупотребительные BBCode-теги, которые уже стали стандартом де-факто в Интернете:

- `[b]`*<текст>*`[/b]` — **полужирный** *текст*;
- `[i]`*<текст>*`[/i]` — *курсивный* *текст*;
- `[u]`*<текст>*`[/u]` — подчеркнутый *текст*;
- `[s]`*<текст>*`[/s]` — ~~зачеркнутый~~ *текст*;
- `[img]`*<интернет-адрес>*`[/img]` — изображение с заданного *интернет-адреса*;
- `[url]`*<интернет-адрес>*`[/url]` — *интернет-адрес* в виде гиперссылки;
- `[url=`*<интернет-адрес>*`]`*<текст>*`[/url]` — *текст* в виде гиперссылки, указывающей на заданный *интернет-адрес*;
- `[color=<цвет>]`*<текст>*`[/color]` — *текст* указанного *цвета*, который может быть задан в любом формате, поддерживаемом CSS:

```
[color=green]Зеленый текст[/color]
[color=#cccccc]Серый текст[/color]
```

- `[center]`*<текст>*`[/center]` — выравнивает *текст* по середине;
- `[list]`*<набор пунктов>*`[/list]` — маркированный список с указанным *набором пунктов*;
- `[list=`*<тип нумерации>*`]`*<набор пунктов>*`[/list]` — нумерованный список с указанным *набором пунктов*. В качестве *типа нумерации* можно указать:
  - `1` — арабские цифры;
  - `01` — арабские цифры с начальным нулем;

- `I` — римские цифры в верхнем регистре;
- `i` — римские цифры в нижнем регистре;
- `A` — латинские буквы в верхнем регистре;
- `a` — латинские буквы в нижнем регистре;

- `[*]<текст пункта списка>` — отдельный пункт списка, формируемого тегом `[list]`:

```
[list]
[*]Python
[*]Django
[*]django-precise-bbcode
[/list]
```

- `[quote]<текст>[/quote]` — *текст* в виде цитаты, выводится с отступом слева;
- `[code]<текст>[/code]` — *текст*, выведенный моноширинным шрифтом.

19.1.3. Обработка BBCode

19.1.3.1. Обработка BBCode при выводе

Чтобы выполнить преобразование BBCode в HTML-код в шаблоне, нужно предварительно загрузить библиотеку тегов с псевдонимом `bbcode_tags`:

```
{% load bbcode_tags %}
```

После этого можно воспользоваться одним из двух следующих инструментов:

- тегом `bbcode` *<выводимый текст>*:

```
{% bbcode bb.content %}
```

- фильтром `bbcode`:

```
{{ bb.content|bbcode|safe }}
```

Недостаток этого фильтра — необходимость его использования совместно с фильтром `safe`. Если фильтр `safe` не указать, то все содержащиеся в выводимом тексте недопустимые знаки HTML будут преобразованы в специальные символы, на экран будет выведен непосредственно сам HTML-код, а не результат его обработки.

Также можно выполнить преобразование BBCode в HTML-код прямо в контроллере. Для этого потребуется два шага:

- вызов функции `get_parser()` из модуля `precise_bbcode.bbcode`. В качестве результата она вернет объект преобразователя;
- вызов метода `render(<текст BBCode>)` полученного преобразователя. Метод вернет HTML-код, полученный преобразованием заданного *текста BBCode*.

Пример:

```
from precise_bbcode.bbcode import get_parser
def detail(request, pk):
    parser = get_parser()
```

```
    bb = Bb.objects.get(pk=pk)
    parsed_content = parser.render(bb.content)
    . . .
```

19.1.3.2. Хранение BBCode в модели

Поле типа `BBCodeTextField` модели, объявленное в модуле `precise_bbcode.fields`, служит для хранения текста, отформатированного с применением BBCode. Оно поддерживает все параметры, общие для всех классов полей (см. *разд. 4.2.1*), и дополнительные параметры конструктора класса `TextField` (поскольку является производным от этого класса).

Пример:

```
from precise_bbcode.fields import BBCodeTextField

class Bb(models.Model):
    . . .
    content = BBCodeTextField(null=True, blank=True, verbose_name='Описание')
    . . .
```

Для вывода содержимого такого поля, преобразованного в HTML-код, в шаблоне следует воспользоваться атрибутом `rendered`:

```
{{ bb.content.rendered }}
```

Фильтр `safe` здесь указывать не нужно.

> ***На заметку***
>
> При выполнении миграции на каждое поле типа `BBCodeTextField`, объявленное в модели, создаются два поля таблицы. Первое поле хранит изначальный текст, занесенный в соответствующее поле модели, а его имя совпадает с именем этого поля модели. Второе поле хранит HTML-код, полученный в результате преобразования изначального текста, а его имя имеет вид `_<имя первого поля>_rendered`. При выводе содержимого поля типа `BBCodeTextField` в шаблоне путем обращения к атрибуту `rendered` выводится HTML-код из второго поля.

> ***Внимание!***
>
> Поле типа `BBCodeTextField` стоит объявлять только в том случае, если в модели нет ни одной записи. Если же его добавить в модель, содержащую записи, то после выполнения миграции второе поле, хранящее полученный в результате преобразования HTML-код, окажется пустым, и при попытке вывести его содержимое на экран мы ничего не увидим. Единственный способ заполнить второе поле — выполнить правку и сохранение каждой записи модели.

19.1.4. Создание дополнительных BBCode-тегов

Дополнительные BBCode-теги, поддержку которых следует добавить в django-precise-bbcode, записываются в базе данных сайта. Работа с ними выполняется в административном сайте Django, в приложении **Precise BBCode** под графой **BBCode tags**. Изначально перечень дополнительных тегов пуст.

Для каждого вновь создаваемого дополнительного тега необходимо ввести следующие сведения:

- **Tag definition** — сам BBCode-тег.

  Тег может иметь содержимое, помещенное между открывающим и закрывающим тегами (внутри тега), и параметр, указанный в открывающем теге после его имени и отделенный от него знаком =. Для их указания применяются следующие специальные символы:

  - `{TEXT}` — обозначает произвольный фрагмент текста:

    ```
    [right]{TEXT}[/right]
    [spoiler={TEXT}]{TEXT}[/spoiler]
    ```

  - `{SIMPLETEXT}` — текст из букв латиницы, цифр, пробелов, точек, запятых, знаков "плюс", дефисов и подчеркиваний;
  - `{COLOR}` — цвет в любом из форматов, поддерживаемых CSS:

    ```
    [color={COLOR}]{TEXT}[/color]
    ```

  - `{URL}` — интернет-адрес;
  - `{EMAIL}` — адрес электронной почты;
  - `{NUMBER}` — число;
  - `{RANGE=`*<минимум>*`,`*<максимум>*`}` — число в диапазоне от *минимума* до *максимума*:

    ```
    [digit]{RANGE=0,9}[/digit]
    ```

  - `{CHOICE=`*<перечень значений, разделенных запятыми>*`}` — любое из указанных *значений*:

    ```
    [platform]{CHOICE=Python,Django,SQLite}[/platform]
    ```

- **Replacement HTML code** — эквивалентный HTML-код. Для указания в нем мест, куда следует вставить содержимое и параметр создаваемого BBCode-тега, используются те же самые специальные символы, что были приведены ранее. Несколько примеров можно увидеть в табл. 19.1;

***Таблица 19.1.** Примеры объявлений тегов BBCode и написания эквивалентного HTML-кода*

Объявление BBCode-тега	Эквивалентный HTML-код

`[right]{TEXT}[/right]`	`<div style="text-align:right;">`
	`  {TEXT}`
	`</div>`

`[spoiler={TEXT}]{TEXT}[/spoiler]`	`<div class="spoiler">`
	`  <div class="header">`
	`  {TEXT}`
	`  </div>`
	`  <div class="content">`
	`  {TEXT}`
	`  </div>`
	`</div>`

Таблица 19.1 (окончание)

Объявление BBCode-тега	Эквивалентный HTML-код

`[color={COLOR}]{TEXT}[/color]`	`<span style="color:{COLOR};">`
	`  {TEXT}`
	`</span>`

- **Standalone tag** — установить флажок, если создается одинарный тег, и сбросить, если создаваемый тег — парный (изначально сброшен);

Остальные параметры находятся под спойлером **Advanced options**:

- **Newline closing** — установка флажка предпишет закрывать тег перед началом новой строки (изначально сброшен);
- **Same tag closing** — установка этого флажка предпишет закрывать тег, если в тексте далее встретится такой же тег (изначально сброшен);
- **End tag closing** — установка этого флажка предпишет закрывать тег перед окончанием содержимого тега, в который он вложен (изначально сброшен);
- **Transform line breaks** — установка флажка вызовет преобразование переводов строк в соответствующие HTML-теги (изначально установлен);
- **Render embedded tags** — если флажок установлен, все BBCode-теги, вложенные в текущий тег, будут соответственно обработаны (изначально установлен);
- **Escape HTML characters** — установка флажка укажет преобразовывать любые недопустимые знаки HTML ("меньше", "больше", амперсанд) в соответствующие специальные символы (изначально установлен);
- **Replace links** — если флажок установлен, то все интернет-адреса, присутствующие в содержимом текущего тега, будут преобразованы в гиперссылки (изначально установлен);
- **Strip leading and trailing whitespace** — установка флажка приведет к тому, что начальные и конечные пробелы в содержимом тега будут удалены (изначально сброшен);
- **Swallow trailing newline** — будучи установленным, флажок предпишет удалять первый из следующих за тегом переводов строки (изначально сброшен).

На заметку

В составе сведений о создаваемом BBCode-теге также присутствуют поле ввода **Help text for this tag** и флажок **Display on editor**, не описанные в документации по библиотеке. В поле ввода заносится, судя по всему, необязательное описание тега. Назначение флажка неясно.

На заметку

Библиотека django-precise-bbcode также позволяет создавать более сложные BBCode-теги, однако для этого требуется программирование. Необходимые инструкции находятся на домашнем сайте библиотеки.

19.1.5. Создание графических смайликов

Библиотека django-precise-bbcode может выводить вместо текстовых смайликов, присутствующих в тексте, указанные для них графические изображения.

> ***Внимание!***
>
> Для работы с графическими смайликами задействуется подсистема Django, обрабатывающая выгруженные пользователями файлы, которую нужно предварительно настроить. Как это сделать, описывается в *главе 20*.

Список графических смайликов также хранится в базе данных сайта, а работа с ним выполняется на административном сайте Django, в приложении **Precise BBCode** под графой **Smilies**. Изначально набор графических смайликов пуст.

Для каждого графического смайлика нужно указать следующие сведения:

- **Smiley code** — текстовое представление смайлика (примеры: `":-)"`, `":-("`);
- **Smiley icon** — графическое изображение смайлика, выводящееся вместо его текстового представления;
- **Smiley icon width** — необязательная ширина смайлика в пикселах. Если не указана, то смайлик при выводе будет иметь свою изначальную ширину;
- **Smiley icon height** — необязательная высота смайлика в пикселах. Если не указана, то смайлик при выводе будет иметь свою изначальную высоту.

> ***На заметку***
>
> В составе сведений о создаваемом графическом смайлике также присутствуют поле ввода **Related emotions** и флажок **Display on editor**, не описанные в документации. Их назначение неясно.

19.1.6. Настройка django-precise-bbcode

Настройки этой библиотеки указываются в модуле settings.py пакета конфигурации:

- `BBCODE_NEWLINE` — строка с HTML-кодом, которым при выводе будет заменен перевод строки (по умолчанию: `"<br>"`);
- `BBCODE_ESCAPE_HTML` — набор знаков, которые при выводе должны заменяться специальными символами HTML. Указывается в виде последовательности, каждым элементом которой должна быть последовательность из двух элементов: самого заменяемого знака и соответствующего ему специального символа. По умолчанию задана последовательность:

```
( ('&', '&amp;'), ('<', '&lt;'), ('>', '&gt;'), ('"', '&quot;'),
  ('\'', '&#39;'), )
```

- `BBCODE_DISABLE_BUILTIN_TAGS` — если `True`, BBCode-теги, поддержка которых встроена в библиотеку (см. *разд. 19.1.2*), не будут обрабатываться, если `False` — будут (по умолчанию — `False`).

  Этот параметр может пригодиться, если стоит задача полностью изменить набор поддерживаемых библиотекой тегов. В таком случае следует присвоить ему зна-

чение `True` и создать все нужные теги самостоятельно, пользуясь инструкциями из *разд. 19.1.4*;

- `BBCODE_ALLOW_CUSTOM_TAGS` — если `True`, то дополнительные теги, созданные разработчиком сайта, будут обрабатываться библиотекой, если `False` — не будут (по умолчанию — `True`).

  Установка этого параметра в `False` может сэкономить немного системных ресурсов, но лишь при условии, что будут использоваться только теги, поддержка которых встроена в саму библиотеку (см. *разд. 19.1.2*);

- `BBCODE_ALLOW_CUSTOM_TAGS` — если `True`, то все символы `\n`, присутствующие в тексте, будут заменяться последовательностями символов `\r\n`, если `False` — не будут (по умолчанию — `True`);

- `BBCODE_ALLOW_SMILIES` — если `True`, то библиотека будет выводить графические смайлики, если `False` — не будет (по умолчанию — `True`).

  Установка этого параметра в `False` сэкономит немного системных ресурсов, если функциональность по выводу графических смайликов не используется;

- `SMILIES_UPLOAD_TO` — путь к папке для сохранения выгруженных файлов с изображениями смайликов. Данная папка должна располагаться в папке, хранящей выгруженные файлы, поэтому путь к ней указывается относительно этой папки. По умолчанию: `"precise_bbcode/smilies"` (о настройке подсистемы обработки выгруженных файлов будет рассказано в *главе 20*).

19.2. Библиотека django-bootstrap4: интеграция с Bootstrap

Bootstrap — популярный CSS-фреймворк для быстрой верстки веб-страниц и создания всевозможных интерфейсных элементов наподобие меню, спойлеров и пр. Использовать Bootstrap для оформления Django-сайта позволяет дополнительная библиотека django-bootstrap4.

С помощью этой библиотеки можно оформлять веб-формы, пагинатор, предупреждения и всплывающие сообщения (о них разговор пойдет в *главе 23*). Также она включает средства для привязки к формируемым веб-страницам необходимых таблиц стилей и файлов веб-сценариев.

> ***На заметку***
>
> Документацию по фреймворку Bootstrap можно найти здесь: **https://getbootstrap.com/**, а полную документацию по библиотеке django-bootstrap4 — здесь: **https://django-bootstrap4.readthedocs.io/en/latest/index.html**.

19.2.1. Установка django-bootstrap4

Для установки библиотеки необходимо набрать команду:

```
pip install django-bootstrap4
```

Помимо django-bootstrap4, будут установлены библиотеки beautifulsoup4 и soupsieve, необходимые ей для работы.

Далее нужно добавить приложение `bootstrap4`, входящее в состав библиотеки, в список приложений проекта (параметр `INSTALLED_APPS` настроек проекта, подробнее — в *разд. 3.3.3*):

```
INSTALLED_APPS = [
    . . .
    'bootstrap4',
]
```

19.2.2. Использование django-bootstrap4

Перед использованием django-bootstrap4 следует загрузить библиотеку тегов с псевдонимом `bootstrap4`:

```
{% load bootstrap4 %}
```

Далее приведены все теги, поддерживаемые этой библиотекой:

- `bootstrap_css` — вставляет в шаблон HTML-код, привязывающий к странице таблицу стилей Bootstrap;
- `bootstrap_javascript [jquery=<редакция jQuery>]` — вставляет в шаблон HTML-код, привязывающий к странице файлы веб-сценариев Bootstrap и jQuery (эта библиотека требуется для реализации некоторых эффектов и к тому же может оказаться полезной при программировании веб-сценариев). В качестве *редакции jQuery*, привязываемой к странице, можно указать одно из следующих значений:
  - `False` — вообще не привязывать jQuery (поведение по умолчанию). Однако при этом не будут работать сложные элементы страниц, создаваемые средствами Bootstrap, наподобие раскрывающихся меню;
  - `"slim"` — привязать сокращенную редакцию jQuery, из которой исключены средства для работы с AJAX и анимацией. Тем не менее сложные элементы страниц, создаваемые с помощью Bootstrap, работать будут;
  - `True` — привязать полную редакцию jQuery.

  Пример:

```
<html>
    <head>
        . . .
        {% bootstrap_css %}
        {% bootstrap_javascript jquery=True %}
        . . .
    </head>
    <body>
        . . .
    </body>
</html>
```

- `bootstrap_form` *<форма>* `[exclude=`*<список имен полей, которые не должны выводиться на экран>*`]` `[`*<параметры оформления>*`]` — выводит указанную *форму*:

```
{% load bootstrap4 %}

<form method="post">
    {% csrf_token %}
    {% bootstrap_form form %}
    {% buttons submit='Добавить' %}{% endbuttons %}
</form>
```

В необязательном параметре `exclude` можно указать *список имен полей, которые не должны выводиться на экран*, приведя их через запятую:

```
{% bootstrap_form form exclude='price,rubric' %}
```

Параметры оформления будут действовать на все поля формы, выводящиеся на экран (т. е. не упомянутые в параметре `exclude`):

- `layout` — разметка полей формы в виде строки:
  - `"vertical"` — надписи выводятся над элементами управления (разметка по умолчанию);
  - `"horizontal"` — надписи выводятся слева от элементов управления;
  - `"inline"` — надписи выводятся непосредственно в элементах управления. Поддерживается только у полей ввода и областей редактирования, у остальных элементов управления надписи не выводятся вообще;
- `form_group_class` — имя стилевого класса, что будет привязан к блокам (тегам `<div>`), в которые заключается каждый элемент управления вместе с относящейся к нему надписью. По умолчанию: `"form_group"`;
- `field_class` — имя стилевого класса, что будет привязан к блокам, в которые заключается каждый элемент управления (но не относящаяся к нему надпись). По умолчанию — "пустая" строка (т. е. никакой стилевой класс не будет привязан к блокам);
- `label_class` — имя стилевого класса, что будет привязан к тегам `<label>`, создающим надписи. По умолчанию — "пустая" строка;
- `show_help` — если `True`, для полей будет выведен дополнительный поясняющий текст (разумеется, если он задан), если `False` — не будет выведен. По умолчанию — `True`;
- `show_label` — управляет выводом надписей у элементов управления:
  - `True` — надписи будут выводиться (поведение по умолчанию);
  - `False` или `'sr-only'` — надписи выводиться не будут, однако в HTML-коде будут присутствовать создающие их теги, благодаря этому программы чтения с экрана смогут прочитать их;
  - `'skip'` — вообще не формировать надписи;

- `size` — размер элемента управления и его надписи в виде строки `"small"` (маленький), `"medium"` (средний) и `"large"` (большой). По умолчанию: `"medium"`;
- `horizontal_label_class` — имя стилевого класса, который будет привязан к тегам `<label>` надписей, если используется разметка `"horizontal"`. По умолчанию: `"col-md-3"` (может быть изменено в настройках библиотеки);
- `horizontal_field_class` — имя стилевого класса, который будет привязан к тегам `<div>` с элементами управления, если используется разметка `"horizontal"`. По умолчанию: `"col-md-9"` (может быть изменено в настройках библиотеки);
- `required_css_class` — имя стилевого класса, что будет привязан к блоку, охватывающему надпись и элемент управления, в который обязательно следует занести значение. По умолчанию — "пустая" строка;
- `bound_css_class` — имя стилевого класса, что будет привязан к блоку, охватывающему надпись и элемент управления, в который занесено корректное значение. По умолчанию: `"has-success"`;
- `error_css_class` — имя стилевого класса, что будет привязан к блоку, охватывающему надпись и элемент управления, в который занесены некорректные данные. По умолчанию: `"has-error"`.

Пример:

```
{% bootstrap_form form layout='horizontal' show_help=False size='small' %}
```

- `bootstrap_form_errors` *<форма>* `[type=`*<тип ошибок>*`]` — выводит список ошибок, допущенных посетителем при занесении данных в указанную *форму*. В качестве *типа ошибок* можно указать одну из следующих строк:
  - `"all"` — все ошибки (значение по умолчанию);
  - `"fields"` — ошибки, относящиеся к полям формы;
  - `"non_fields"` — ошибки, относящиеся к форме в целом.

  Применяется, если нужно вывести сообщения об ошибках в каком-то определенном месте страницы. В противном случае можно положиться на тег `bootstrap_form`, который выводит такие сообщения непосредственно в форме;
- `bootstrap_formset` *<набор форм>* `[`*<параметры оформления>*`]` — выводит указанный *набор форм*:

```
{% load bootstrap4 %}

<form method="post">
    {% csrf_token %}
    {% bootstrap_formset formset %}
    {% buttons submit='Сохранить' %}{% endbuttons %}
</form>
```

Поддерживаются те же *параметры оформления*, которые были рассмотрены в описании тега `bootstrap_form`;

- `bootstrap_formset_errors` *<набор форм>* — выводит список ошибок, допущенных посетителем при занесении данных в указанный *набор форм*.

  Применяется, если нужно вывести сообщения об ошибках в определенном месте страницы. В противном случае можно положиться на тег `bootstrap_formset`, который выводит такие сообщения непосредственно в наборе форм;

- `buttons submit=`*<текст надписи>* `[reset=`*<текст надписи>*`] . . . endbuttons` — выводит кнопку отправки данных и, возможно, кнопку сброса формы. Параметр `submit` задает текст надписи для кнопки отправки данных. Необязательный параметр `reset` задает текст надписи для кнопки сброса формы; если он не указан, то такая кнопка не будет создана. Пример:

  ```
  {% buttons submit='Добавить объявление' %}{% endbuttons %}
  ```

- `bootstrap_button` *<параметры кнопки>* — выводит кнопку. Поддерживаются следующие *параметры* создаваемой кнопки:

  - `content` — текст надписи для кнопки;
  - `button_type` — тип кнопки. Доступны строковые значения `"submit"` (кнопка отправки данных), `"reset"` (кнопка сброса формы), `"button"` (обычная кнопка) и `"link"` (кнопка-гиперссылка);
  - `href` — интернет-адрес для кнопки-гиперссылки;
  - `size` — размер кнопки в виде строки `"xs"` (самый маленький), `"sm"`, `"small"` (маленький), `"md"`, `"medium"` (средний), `"lg"`, `"large"` (большой). По умолчанию создается кнопка среднего размера;
  - `button_class` — имя стилевого класса, который будет привязан к кнопке (по умолчанию: `"btn-default"`);
  - `extra_classes` — имена дополнительных стилевых классов, которые следует привязать к кнопке, перечисленные через пробел (по умолчанию — "пустая" строка);
  - `name` — значение для атрибута `name` тега `<button>`, создающего кнопку;
  - `value` — значение для атрибута `value` тега `<button>`, создающего кнопку.

  Пример:

  ```
  {% bootstrap_button content='Сохранить' button_type='submit' ↳
  button_class='btn-primary' %}
  ```

- `bootstrap_field` *<поле формы>* `[`*<параметры оформления>*`]` `[`*<дополнительные параметры оформления>*`]` — выводит указанное *поле формы*:

  ```
  {% bootstrap_field form.title %}
  ```

  Поддерживаются те же *параметры оформления*, которые были рассмотрены в описании тега `bootstrap_form`, и, помимо них, *дополнительные параметры оформления*:

  - `placeholder` — текст, который будет выводиться непосредственно в элементе управления, если используется разметка, отличная от `"inline"`. Поддерживается только у полей ввода и областей редактирования;

- `addon_before` — текст, который будет помещен перед элементом управления (по умолчанию — "пустая" строка);
- `addon_before_class` — имя стилевого класса, который будет привязан к тегу `<span>`, охватывающему текст, помещаемый перед элементом управления. Если указать значение `None`, то тег `<span>` создаваться не будет. По умолчанию: `"input-group-text"`;
- `addon_after` — текст, который будет помещен после элемента управления (по умолчанию — "пустая" строка);
- `addon_after_class` — имя стилевого класса, который будет привязан к тегу `<span>`, охватывающему текст, помещаемый после элемента управления. Если указать значение `None`, то тег `<span>` создаваться не будет. По умолчанию: `"input-group-text"`.

Пример:

```
{% bootstrap_field form.title placeholder='Товар' show_label=False %}
```

- `bootstrap_messages` — выводит всплывающие сообщения (см. *главу 23*):

```
{% bootstrap_messages %}
```

- `bootstrap_alert` *<параметры предупреждения>* — выводит предупреждение с заданными *параметрами*:
  - `content` — HTML-код, создающий содержимое предупреждения;
  - `alert_type` — тип предупреждения в виде строки `"info"` (простое сообщение), `"warning"` (предупреждение о некритической ситуации), `"danger"` (предупреждение о критической ситуации) и `"success"` (сообщение об успехе выполнения какой-либо операции). По умолчанию: `"info"`;
  - `dismissable` — если `True`, то в сообщении будет присутствовать кнопка закрытия в виде крестика, щелкнув на которой посетитель уберет предупреждение со страницы. Если `False`, то кнопка закрытия не выведется, и предупреждение будет присутствовать на странице постоянно. По умолчанию — `True`.

Пример:

```
{% bootstrap_alert content='<p>Рубрика добавлена</p>' ↳
alert_type='success' %}
```

- `bootstrap_label` *<параметры надписи>* — выводит надпись со следующими *параметрами*:
  - `content` — текст надписи;
  - `label_for` — значение, которое будет присвоено атрибуту `for` тега `<label>`, создающего надпись;
  - `label_class` — имя стилевого класса, который будет привязан к тегу `<label>`, создающему надпись;
  - `label_title` — текст всплывающей подсказки для надписи;

- bootstrap_pagination *<часть пагинатора>* [*<параметры пагинатора>*] — выводит пагинатор на основе заданной *части* (представленной экземпляром класса Page, описанного в *разд. 12.2*). Поддерживаются дополнительные *параметры пагинатора*:
  - pages_to_show — количество гиперссылок, указывающих на части пагинатора, которые будут выведены на страницу (остальные будут скрыты). По умолчанию: 11 (текущая часть плюс по 5 частей предыдущих и следующих);
  - url — интернет-адрес, на основе которого будут формироваться интернет-адреса отдельных частей пагинатора. Если указать значение None, то будет использован текущий интернет-адрес. По умолчанию — None;
  - size — размер гиперссылок, ведущих на части пагинатора, в виде строки "small" (маленький), None (средний) или "large" (большой). По умолчанию — None;
  - parameter_name — имя GET-параметра, через который передается номер текущей части (по умолчанию: "page").

  Пример:

```
{% bootstrap_pagination page size="small" %}
```

19.2.3. Настройка django-bootstrap4

Настройки библиотеки django-bootstrap4 записываются в параметре BOOTSTRAP4 модуля settings.py пакета конфигурации. Значением этого параметра должен быть словарь, отдельные элементы которого представляют отдельные параметры библиотеки. Пример:

```
BOOTSTRAP4 = {
    'required_css_class': 'required',
    'success_css_class': 'has-success',
    'error_css_class': 'has-error',
}
```

Список наиболее полезных параметров приведен далее.

- horizontal_label_class — имя стилевого класса, который будет привязан к тегам <label>, создающим надписи, если используется разметка "horizontal" (по умолчанию: "col-md-3");
- horizontal_field_class — имя стилевого класса, который будет привязан к тегам , заключающим в себе элементы управления, если используется разметка "horizontal" (по умолчанию: "col-md-9");
- required_css_class — имя стилевого класса, что будет привязан к блоку, охватывающему надпись и элемент управления, в который обязательно следует занести значение (по умолчанию — "пустая" строка);
- success_css_class — имя стилевого класса, что будет привязан к блоку, охватывающему надпись и элемент управления, в который занесено корректное значение (по умолчанию: "has-success");

- `error_css_class` — имя стилевого класса, что будет привязан к блоку, охватывающему надпись и элемент управления, в который занесены некорректные данные. Значение по умолчанию: `"has-error"`.

Библиотека использует следующие шаблоны, хранящиеся по пути *<папка, в которой установлен Python>*\Lib\site-packages\bootstrap4\templates\bootstrap4:

- field_help_text.html — выводит дополнительный поясняющий текст. Строка с этим текстом хранится в переменной `field_help` контекста шаблона;
- field_errors.html — выводит перечень ошибок, допущенных пользователем при занесении значения в какой-либо элемент управления. Список строк с сообщениями об ошибках хранится в переменной `field_errors` контекста шаблона;
- form_errors.html — выводит перечень ошибок, относящихся к форме целиком. Список строк с сообщениями об ошибках хранится в переменной `errors` контекста шаблона;
- messages.html — выводит всплывающие сообщения. Список строк с этими сообщениями хранится в переменной `messages` контекста шаблона.

Мы можем сделать копию этих шаблонов, поместив их в папке templates\bootstrap4 пакета приложения, и исправить их в соответствии со своими нуждами.

19.3. Написание своих фильтров и тегов

Здесь рассказывается, как написать свой собственный фильтр и самую простую разновидность тега (более сложные разновидности приходится разрабатывать много реже).

19.3.1. Организация исходного кода

Модули с кодом, объявляющим фильтры и теги шаблонизатора, должны находиться в пакете templatetags пакета приложения. Поэтому сразу же создадим в пакете приложения папку с именем templatetags, а в ней — "пустой" модуль __init__.py.

> ***Внимание!***
>
> Если отладочный веб-сервер Django запущен, после создания пакета *templatetags* его следует перезапустить, чтобы он перезагрузил обновленный код сайта с вновь созданными модулями.

Вот схема организации исходного кода для приложения `bboard` (предполагается, что код фильтров и тегов хранится в модуле filtersandtags.py):

```
<папка проекта>
    bboard
        __.init__.py
        . . .
        templatetags
            __.init__.py
            filtersandtags.py
        . . .
```

Каждый модуль, объявляющий фильтры и теги, становится библиотекой тегов. Псевдоним этой библиотеки совпадает с именем модуля.

19.3.2. Написание фильтров

Проще всего написать фильтр, который принимает какое-либо значение и, возможно, набор параметров, после чего возвращает то же самое значение в преобразованном виде.

19.3.2.1. Написание и использование простейших фильтров

Фильтр — это обычная функция Django, которая:

- в качестве первого параметра принимает обрабатываемое значение;
- в качестве последующих параметров принимает значения параметров, указанных у фильтра. Эти параметры могут иметь значения по умолчанию;
- возвращает в качестве результата преобразованное значение.

Объявленную функцию нужно зарегистрировать в шаблонизаторе в качестве фильтра.

Сначала необходимо создать экземпляр класса `Library` из модуля `django.template`. Потом у этого экземпляра класса нужно вызвать метод `filter()` в следующем формате:

```
filter(<имя регистрируемого фильтра>,
       <ссылка на функцию, реализующую фильтр>)
```

Зарегистрированный фильтр будет доступен в шаблоне под указанным *именем*.

В листинге 19.1 приведен пример объявления и регистрации фильтра `currency`. Он принимает числовое значение и, в качестве необязательного параметра, обозначение денежной единицы. В качестве результата он возвращает строку с числовым значением, отформатированным как денежная сумма.

Листинг 19.1. Пример создания фильтра

```
from django import template

register = template.Library()

def currency(value, name='руб.'):
    return '%1.2f %s' % (value, name)

register.filter('currency', currency)
```

Вызов метода `filter()` можно оформить как декоратор. В таком случае он указывается у функции, реализующей фильтр, и вызывается без параметров. Пример:

```
@register.filter
def currency(value, name='руб.'):
    . . .
```

В необязательном параметре `name` декоратора `filter()` можно указать другое имя, под которым фильтр будет доступен в шаблоне:

```
@register.filter(name='cur')
def currency(value, name='руб.'):
    . . .
```

Может случиться так, что фильтр в качестве обрабатываемого должен принимать значение исключительно строкового типа, но ему было передано значение, тип которого отличается от строки (например, число). В этом случае при попытке обработать такое значение как строку (скажем, при вызове у него метода, который поддерживается только строковым типом) возникнет ошибка. Но мы можем указать Django предварительно преобразовать нестроковое значение в строку. Для этого достаточно задать для функции, реализующей фильтр, декоратор `stringfilter` из модуля `django.template.defaultfilters`. Пример:

```
from django.template.defaultfilters import stringfilter
. . .
@register.filter
@stringfilter
def somefilter(value):
    . . .
```

Если фильтр в качестве обрабатываемого значения принимает дату и время, то мы можем указать, чтобы это значение было автоматически преобразовано в местное время в текущей временно́й зоне. Для этого нужно задать у декоратора `filter` параметр `expects_localtime` со значением `True`. Пример:

```
@register.filter(expects_localtime=True)
def datetimefilter(value):
    . . .
```

Объявленный фильтр можно использовать в шаблонах. Ранее говорилось, что модуль, объявляющий фильтры, становится библиотекой тегов, псевдоним которой совпадает с именем модуля. Следовательно, чтобы задействовать фильтр, нужно предварительно загрузить нужную библиотеку тегов с помощью тега `load`. Пример (предполагается, что фильтр `currency` объявлен в модуле **filtersandtags.py**):

```
{% load filtersandtags %}
```

Объявленный нами фильтр используется так же, как и любой из встроенных в Django:

```
{{ bb.price|currency }}
{{ bb.price|currency:'p.' }}
```

19.3.2.2. Управление заменой недопустимых знаков HTML

Если в выводимом на страницу значении присутствует какой-либо из недопустимых знаков HTML: символ "меньше", "больше", двойная кавычка, амперсанд, то он должен быть преобразован в соответствующий ему специальный символ. В коде фильтров для этого можно использовать две функции из модуля `django.utils.html`:

- `escape(<строка>)` — выполняет замену всех недопустимых знаков в *строке* и возвращает обработанную строку в качестве результата;
- `conditional_escape(<строка>)` — то же самое, что `escape()`, но выполняет замену только в том случае, если в переданной ему *строке* такая замена еще не производилась.

Результат, возвращаемый фильтром, должен быть помечен как строка, в которой была выполнена замена недопустимых знаков. Сделать это можно, вызвав функцию `mark_safe(<помечаемая строка>)` из модуля `django.utils.safestring` и вернув из фильтра возвращенный ей результат. Пример:

```
from django.utils.safestring import mark_safe
. . .
@register.filter
def somefilter(value):
    . . .
    return mark_safe(result_string)
```

Строка, помеченная как прошедшая замену недопустимых знаков, представляется экземпляром класса `SafeText` из того же модуля `django.utils.safestring`. Так что мы можем проверить, проходила ли полученная фильтром строка процедуру замены или еще нет. Пример:

```
from django.utils.html import escape
from django.utils.safestring import SafeText
. . .
@register.filter
def somefilter(value):
    if not isinstance(value, SafeText):
        # Полученная строка не прошла замену. Выполняем ее сами
        value = escape(value)
    . . .
```

Еще нужно учитывать тот факт, что разработчик может отключить автоматическую замену недопустимых знаков в каком-либо фрагменте кода шаблона, заключив его в тег шаблонизатора `autoescape . . . endautoescape`. Чтобы в коде фильтра выяснить, была ли отключена автоматическая замена, следует указать в вызове декоратора `filter()` параметр `needs_autoescape` со значением `True` и добавить в список параметров функции, реализующей фильтр, параметр `autoescape` со значением по умолчанию `True`. В результате последний получит значение `True`, если автоматическая замена активна, и `False`, если она была отключена. Пример:

```
@register.filter(needs_autoescape=True)
def somefilter(value, autoescape=True):
```

```
    if autoescape:
        value = escape(value)
    . . .
```

И наконец, можно уведомить Django, что он сам должен выполнять замену в значении, возвращенном фильтром. Для этого достаточно в вызове декоратора `filter()` указать параметр `is_safe` со значением `True`. Пример:

```
@register.filter(is_safe=True)
def currency(value, name='руб.'):
    . . .
```

19.3.3. Написание тегов

Объявить простейший одинарный тег шаблонизатора, вставляющий какие-либо данные в то место, где он находится, немногим сложнее, чем создать фильтр.

19.3.3.1. Написание тегов, выводящих элементарные значения

Если объявляемый тег должен выводить какое-либо элементарное значение: строку, число или дату, — нужно лишь объявить функцию, которая реализует этот тег.

Функция, реализующая тег, может принимать произвольное количество параметров, как обязательных, так и необязательных. В качестве результата она должна возвращать выводимое значение в виде строки.

Подобного рода тег регистрируется созданием экземпляра класса `Library` из модуля `template` и вызовом у этого экземпляра метода `simple_tag([name=None][,][takes_context=False])`, причем вызов нужно оформить в виде декоратора у функции, реализующей тег.

Листинг 19.2 иллюстрирует объявление тега `lst`. Он принимает произвольное количество параметров, из которых первый — строка-разделитель — является обязательным, выводит на экран значения остальных параметров, отделяя их друг от друга строкой-разделителем, а в конце ставит количество выведенных значений, взятое в скобки.

Листинг 19.2. Пример объявления тега, выводящего элементарное значение

```
from django import template

register = template.Library()

@register.simple_tag
def lst(sep, *args):
    return '%s (итого %s)' % (sep.join(args), len(args))
```

По умолчанию созданный таким образом тег доступен в коде шаблона под своим изначальным именем, которое совпадает с именем функции, реализующей тег.

В вызове метода `simple_tag()` мы можем указать два необязательных именованных параметра:

- `name` — имя, под которым тег будет доступен в коде шаблона. Используется, если нужно указать для тега другое имя;
- `takes_context` — если `True`, то первым параметром в функцию, реализующую тег, будет передан контекст шаблона:

```
@register.simple_tag(takes_context=True)
def lst(context, sep, *args):
    . . .
```

Значение, возвращенное таким тегом, подвергается автоматической замене недопустимых знаков HTML на специальные символы. Так что нам самим это делать не придется.

Если же замену недопустимых знаков проводить не нужно (например, написанный нами тег должен выводить фрагмент HTML-кода), то возвращаемое функцией значение можно "пропустить" через функцию `mark_safe()`, описанную в *разд. 19.3.2.2*.

Объявив тег, мы можем использовать его в шаблоне (не забыв загрузить модуль, в котором он реализован):

```
{% load filtersandtags %}
. . .
{% lst ', ' '1' '2' '3' '4' '5' '6' '7' '8' '9' %}
```

19.3.3.2. Написание шаблонных тегов

Если тег должен выводить фрагмент HTML-кода, то мы можем, как говорилось ранее, "пропустить" возвращаемую строку через функцию `mark_safe()`. Вот пример подобного рода тега:

```
@register.simple_tag
def lst(sep, *args):
    return mark_safe('%s (итого <strong>%s</strong>)' % ↵
                     (sep.join(args), len(args)))
```

Но если нужно выводить более сложные фрагменты HTML-кода, то удобнее объявить *шаблонный тег*. Возвращаемое им значение формируется так же, как и обычная веб-страница Django-сайта, — рендерингом на основе шаблона.

Функция, реализующая такой тег, должна возвращать в качестве результата контекст шаблона. В качестве декоратора, указываемого для этой функции, нужно поместить вызов метода `inclusion_tag()` экземпляра класса `Library`. Вот формат вызова этого метода:

```
inclusion_tag(<путь к шаблону>[, name=None][, takes_context=False])
```

Листинг 19.3 объявляет шаблонный тег `ulist`. Он аналогичен объявленному в листинге 19.2 тегу `lst`, но выводит перечень переданных ему позиций в виде маркированного списка HTML, а количество позиций помещает под списком и выделяет курсивом.

Листинг 19.3. Пример шаблонного тега

```
from django import template

register = template.Library()

@register.inclusion_tag('tags/ulist.html')
def ulist(*args):
    return {'items': args}
```

Шаблон такого тега ничем не отличается от шаблонов веб-страниц и располагается непосредственно в папке templates пакета приложения. Код шаблона tags\ulist.html тега `ulist` приведен в листинге 19.4.

Листинг 19.4. Шаблон для тега из листинга 19.3

```
<ul>
    {% for item in items %}
    <li>{{ item }}</li>
    {% endfor %}
</ul>
<p>Итого <em>{{ items|length }}</em></p>
```

Метод `inclusion_tag()` поддерживает необязательные параметры `name` и `takes_context`, описанные в *разд. 19.3.3.1*. При указании значения `True` для параметра `takes_context` контекст шаблона страницы также будет доступен в шаблоне тега.

Используется шаблонный тег так же, как и обычный, возвращающий элементарное значение:

```
{% load filtersandtags %}
. . .
{% ulist '1' '2' '3' '4' '5' '6' '7' '8' '9' %}
```

> ***На заметку***
>
> Django также поддерживает объявление более сложных тегов, являющихся парными и выполняющих над своим содержимым различные манипуляции (в качестве примера можно привести тег `for . . . endfor`). Поскольку потребность в разработке новых тегов такого рода возникает нечасто, их объявление не описывается в этой книге. Интересующиеся могут найти руководство по этой теме на странице: **https://docs.djangoproject.com/en/3.0/howto/custom-template-tags/**.

19.3.4. Регистрация фильтров и тегов

Если мы, согласно принятым в Django соглашениям, сохранили модуль с объявлениями фильтров и тегов в пакете templatetags пакета приложения, ничего более делать не нужно. Фреймворк превратит этот модуль в библиотеку тегов, имя модуля станет псевдонимом этой библиотеки, и нам останется лишь загрузить ее, воспользовавшись тегом `load` шаблонизатора.

Но если мы не последовали этим соглашениям или хотим указать у библиотеки тегов другой псевдоним, то придется внести исправления в настройки проекта, касающиеся обработки шаблонов. Эти настройки описывались в *разд. 11.1*.

Чтобы зарегистрировать модуль с фильтрами и тегами как загружаемую библиотеку тегов (т. е. требующую загрузки тегом `load` шаблонизатора), ее нужно добавить в список загружаемых библиотек. Этот список хранится в дополнительных настройках шаблонизатора, задаваемых параметром `OPTIONS` в параметре `libraries`.

Предположим, что модуль filtersandtags.py, хранящий фильтры и теги, находится непосредственно в пакете приложения (что нарушает соглашения Django). Тогда зарегистрировать его мы можем, записав в модуле settings.py пакета конфигурации такой код (выделен полужирным шрифтом):

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        . . .
        'OPTIONS': {
            . . .
            'libraries': {
                'filtersandtags': 'bboard.filtersandtags',
            }
        },
    },
]
```

Нам даже не придется переделывать код шаблонов, т. к. написанная нами библиотека тегов будет доступна под тем же псевдонимом `filtersandtags`.

Мы можем изменить псевдоним этой библиотеки тегов, скажем, на `ft`:

```
'libraries': {
    'ft': 'bboard.filtersandtags',
}
```

и сможем использовать для ее загрузки новое, более короткое имя:

```
{% load ft %}
```

Если же у нас нет желания писать в каждом шаблоне тег `load`, чтобы загрузить библиотеку тегов, то мы можем оформить ее как встраиваемую — и объявленные в ней фильтры и теги станут доступными без каких бы то ни было дополнительных действий. Для этого достаточно указать путь к модулю библиотеки тегов в списке дополнительного параметра `builtins`. Вот пример (добавленный код выделен полужирным шрифтом):

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        . . .
        'OPTIONS': {
            . . .
```

```
            'builtins': [
                'bboard.filtersandtags',
            ],
        },
    },
]
```

После этого мы сможем просто использовать все объявленные в библиотеке теги, когда и где нам заблагорассудится.

19.4. Переопределение шаблонов

Предположим, нужно реализовать выход с сайта с выводом страницы с сообщением о выходе. Для этого мы решаем использовать стандартный контроллер-класс `LogoutView`, описанный в *разд. 15.4.2*, и указанный для него по умолчанию шаблон registration\logged_out.html. Мы пишем шаблон logged_out.html, помещаем его в папку registration, вложенную в папку templates пакета приложения, запускаем отладочный веб-сервер, выполняем вход на сайт, переходим на страницу выхода... и наблюдаем на экране не нашу страницу, а какую-то другую, судя по внешнему виду, принадлежащую административному сайту Django...

Дело в том, что Django в поисках нужного шаблона просматривает папки templates, находящиеся в пакетах *всех* приложений, которые зарегистрированы в проекте, и прекращает поиски, как только найдет первый подходящий шаблон. В нашем случае таким оказался шаблон registration\logged_out.html, принадлежащий стандартному приложению `django.contrib.admin`, т. е. административному сайту.

Мы можем избежать этой проблемы, просто задав для контроллера-класса шаблон с другим именем. Это можно сделать либо в маршруте, вставив нужный параметр в вызов метода `as_view()` контроллера-класса, либо в его подклассе, в соответствующем атрибуте. Но существует и другой способ — использовать *переопределение шаблонов*, "подсунув" Django наш шаблон до того, как он доберется до стандартного.

Есть два способа реализовать переопределение шаблонов:

- приложения в поисках шаблонов просматриваются в том порядке, в котором они указаны в списке зарегистрированных приложений из параметра `INSTALLED_APPS` настроек проекта. Следовательно, мы можем просто поместить наше приложение перед стандартным. Пример (предполагается, что нужный шаблон находится в приложении `bboard`):

  ```
  INSTALLED_APPS = [
      'bboard',
      'django.contrib.admin',
      . . .
  ]
  ```

  После этого, поместив шаблон registration\logged_out.html в папку templates пакета приложения `bboard`, мы можем быть уверены, что наш шаблон будет найден раньше, чем стандартный;

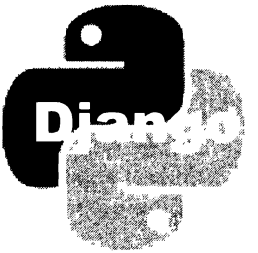
- папки, пути к которым приведены в списке параметра DIRS настроек шаблонизатора (см. *разд. 11.1*), просматриваются *перед* папками templates пакетов приложений. Мы можем создать в папке проекта папку main_templates и поместить шаблон registration\logged_out.html в нее. После чего нам останется изменить настройки шаблонизатора следующим образом:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'main_templates')],
        . . .
    },
]
```

  Значение переменной BASE_DIR вычисляется в модуле settings.py ранее и представляет собой полный путь к папке проекта.

  Здесь мы указали в списке параметра DIRS единственный элемент — путь к только что созданной нами папке. И опять же, мы можем быть уверены, что контроллер-класс будет использовать наш, а не "чужой" шаблон.

Однако при переопределении шаблонов нужно иметь в виду один весьма неприятный момент. Если мы переопределим какой-либо шаблон, задействуемый стандартным приложением, то это стандартное приложение будет использовать переопределенный нами шаблон, а не свой собственный. Например, если мы переопределим шаблон registration\logged_out.html, принадлежащий стандартному приложению django.contrib.admin, то последнее будет использовать именно переопределенный шаблон. Так что в некоторых случаях, возможно, будет целесообразнее воздержаться от переопределения шаблонов стандартных приложений и написать свой шаблон.



ГЛАВА 20

Обработка выгруженных файлов

Django предлагает удобные инструменты для обработки файлов, выгруженных посетителями: как высокоуровневые, в стиле "раз настроил — и забыл", так и низкоуровневые, для специфических случаев.

20.1. Подготовка подсистемы обработки выгруженных файлов

Чтобы успешно обрабатывать в своем сайте выгруженные посетителями файлы, следует выполнить некоторые подготовительные действия.

20.1.1. Настройка подсистемы обработки выгруженных файлов

Настройки этой подсистемы записываются в модуле settings.py пакета конфигурации. Вот наиболее интересные из них:

- `MEDIA_URL` — префикс, добавляемый к интернет-адресу выгруженного файла. Встретив в начале интернет-адреса этот префикс, Django поймет, что это выгруженный файл и его нужно передать для обработки подсистеме выгруженных файлов. По умолчанию — "пустая" строка;
- `MEDIA_ROOT` — полный путь к папке, в которой будут храниться выгруженные файлы. По умолчанию — "пустая" строка.

  Это единственные обязательные для указания параметры подсистемы выгруженных файлов. Вот пример их задания:

  ```
  MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
  MEDIA_URL = '/media/'
  ```

  Значение переменной `BASE_DIR` вычисляется в том же модуле settings.py и представляет собой полный путь к папке проекта;

- FILE_UPLOAD_HANDLERS — последовательность имен классов обработчиков выгрузки (*обработчик выгрузки* извлекает файл из отправленных посетителем данных и временно сохраняет его на диске серверного компьютера или в оперативной памяти). В Django доступны два класса обработчика выгрузки, объявленные в модуле django.core.files.uploadhandler:
  - MemoryFileUploadHandler — сохраняет выгруженный файл в оперативной памяти и задействуется, если размер файла не превышает 2,5 Мбайт (это значение настраивается в другом параметре);
  - TemporaryFileUploadHandler — сохраняет выгруженный файл на диске серверного компьютера в папке для временных файлов. Задействуется, если размер выгруженного файла больше 2,5 Мбайт.

  Значение по умолчанию — список с именами обоих классов, которые выбираются автоматически, в зависимости от размера выгруженного файла;
- FILE_UPLOAD_MAX_MEMORY_SIZE — максимальный размер выгруженного файла, сохраняемого в оперативной памяти, в байтах. Если размер превышает это значение, то файл будет сохранен на диске. По умолчанию: 2621440 байтов (2,5 Мбайт);
- FILE_UPLOAD_TEMP_DIR — полный путь к папке, в которой будут сохраняться файлы, размер которых превышает указанный в параметре FILE_UPLOAD_MAX_MEMORY_SIZE. Если задано значение None, то будет использована стандартная папка для хранения временных файлов в операционной системе. По умолчанию — None;
- FILE_UPLOAD_PERMISSIONS — числовой код прав доступа, даваемых выгруженным файлам. Если задано значение None, то права доступа даст сама операционная система, — в большинстве случаев это будут права 0o600 (владелец может читать и записывать файлы, его группа и остальные пользователи вообще не имеют доступа к файлам). По умолчанию — 0o644 (владелец может читать и записывать файлы, его группа и остальные пользователи — только читать их);

  На заметку

  В версиях Django, предшествующих 3.0, параметр FILE_UPLOAD_PERMISSIONS имел значение по умолчанию None.

- FILE_UPLOAD_DIRECTORY_PERMISSIONS — числовой код прав доступа, даваемых папкам, которые создаются при сохранении выгруженных файлов. Если задано значение None, то права доступа даст сама операционная система, — в большинстве случаев это будут права 0o600 (владелец может читать и записывать файлы в папках, его группа и остальные пользователи вообще не имеют доступа к папкам). По умолчанию — None;
- DEFAULT_FILE_STORAGE — имя класса файлового хранилища, используемого по умолчанию, в виде строки (*файловое хранилище* обеспечивает сохранение файла в выделенной для этого папке, получение его интернет-адреса, параметров и пр.).

По умолчанию: "django.core.files.storage.FileSystemStorage" (это единственное файловое хранилище, поставляемое в составе Django).

20.1.2. Указание маршрута для выгруженных файлов

Практически всегда посетители выгружают файлы на сайт для того, чтобы показать их другим посетителям. Следовательно, на веб-страницах сайта позже будут выводиться сами эти файлы или указывающие на них гиперссылки. Для того чтобы посетители смогли их просмотреть или загрузить, нужно создать соответствующий маршрут (о маршрутах и маршрутизации рассказывалось в *главе 8*).

Маршрут, указывающий на выгруженный файл, записывается в списке уровня проекта, т. е. в модуле urls.py пакета конфигурации. Для его указания используется функция `static()` из модуля `django.conf.urls.static`, которая в этом случае вызывается в следующем формате:

`static(`*<префикс>*`, document_root=`*<путь к папке с выгруженными файлами>*`)`

Она создает маршрут, связывающий:

- шаблонный путь формата:

  <префикс, заданный в вызове функции>/<путь к выгруженному файлу>

- контроллер-функцию `serve()` из модуля `django.views.static`, выдающую файл с заданным путем из указанной папки;
- папку с *путем*, заданным в параметре `document_root`, в которой хранятся выгруженные файлы.

В качестве результата возвращается список с единственным элементом — описанным ранее маршрутом. Если сайт работает в эксплуатационном режиме (подробности — в *разд. 3.3.1*), то возвращается пустой список.

В нашем случае в качестве префикса следует указать значение параметра `MEDIA_URL`, а в качестве пути к папке — значение параметра `MEDIA_ROOT` настроек проекта (их можно получить из модуля settings.py пакета конфигурации):

```
from django.conf.urls.static import static
from django.conf import settings
. . .
urlpatterns = [
    . . .
]
urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Функция `static()` создает маршрут только при работе в отладочном режиме. После перевода сайта в эксплуатационный режим для формирования маршрута придется использовать другие средства (чем мы и займемся в *главе 30*).

20.2. Хранение файлов в моделях

Предоставляемые Django высокоуровневые средства для обработки выгруженных файлов предполагают хранение таких файлов в полях моделей и подходят в большинстве случаев.

20.2.1. Типы полей модели, предназначенные для хранения файлов

Для хранения файлов в моделях Django предусматривает два типа полей, представляемые описанными далее классами из модуля `django.db.models`:

- `FileField` — файл любого типа. Фактически хранит путь к выгруженному файлу, указанный относительно папки, путь к которой задан в параметре `MEDIA_ROOT` настроек проекта.

  Необязательный параметр `max_length` указывает максимальную длину заносимого в поле пути в виде целого числа в символах (по умолчанию: `100`).

  Необязательный параметр `upload_to` задает папку, в которой будет сохранен выгруженный файл и которая должна находиться в папке, чей путь задан в параметре `MEDIA_ROOT`. В качестве значения параметра можно указать:

  - строку с путем, заданным относительно пути из параметра `MEDIA_ROOT`, — файл будет выгружен во вложенную папку, находящуюся по этому пути:

    ```
    archive = models.FileField(upload_to='archives/')
    ```

    В формируемом пути можно использовать составные части текущих даты и времени: год, число, номер месяца и т. п., — вставив в строку с путем специальные символы, поддерживаемые функциями `strftime()` и `strptime()` Python. Перечень этих специальных символов можно найти в документации по Python или на странице **https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior**. Пример:

    ```
    archive = models.FileField(upload_to='archives/%Y/%m/%d/')
    ```

    В результате выгруженные файлы будут сохраняться во вложенных папках с именами формата `archives\<год>\<номер месяца>\<число>`, где *год*, *номер месяца* и *число* взяты из текущей даты.

    Чтобы выгруженные файлы сохранялись непосредственно в папку из параметра `MEDIA_ROOT`, достаточно указать в параметре `upload_to` "пустую" строку. Это, кстати, его значение по умолчанию;

  - функцию, возвращающую путь для сохранения файла, который включает и имя файла, — файл будет сохранен во вложенной папке, расположенной по полученному от функции пути, под полученным именем. Функция должна принимать два параметра: текущую запись модели и изначальное имя выгруженного файла.

Этот способ задания пути сохранения можно использовать для сохранения файлов под какими-либо отвлеченными именами, сформированными на основе, например, текущей временно́й отметки. Вот пример такой функции:

```
from datetime import datetime
from os.path import splitext
def get_timestamp_path(instance, filename):
    return '%s%s' % (datetime.now().timestamp(),
                     splitext(filename)[1])
. . .
file = models.FileField(upload_to=get_timestamp_path)
```

- `ImageField` — графический файл. Фактически хранит путь к выгруженному файлу, указанный относительно папки из параметра `MEDIA_ROOT` настроек проекта.

> ***Внимание!***
>
> **Для успешной обработки полей типа `ImageField` необходимо установить дополнительную библиотеку Pillow (если она не была установлена ранее). Сделать это можно подачей команды:**
>
> ```
> pip install pillow
> ```

Поддерживаются дополнительные параметры `max_length` и `upload_to`, описанные ранее, а также следующие параметры:

- `width_field` — имя поля модели, в которое будет записана ширина изображения из выгруженного файла. Если не указан, то ширина изображения нигде храниться не будет;
- `height_field` — имя поля модели, в которое будет записана высота изображения из выгруженного файла. Если не указан, то высота изображения нигде храниться не будет.

Эти поля будут созданы самим Django. Можно указать создание как обоих полей, так и лишь одного из них (если зачем-то понадобится хранить только один размер изображения).

В листинге 20.1 приведен код модели, в которой присутствует поле типа `ImageField`.

Листинг 20.1. Модель с полем для хранения выгруженного файла

```
from django.db import models

class Img(models.Model):
    img = models.ImageField(verbose_name='Изображение',
                            upload_to=get_timestamp_path)
    desc = models.TextField(verbose_name='Описание')

    class Meta:
        verbose_name='Изображение'
        verbose_name_plural='Изображения'
```

20.2.2. Поля форм, валидаторы и элементы управления, служащие для указания файлов

По умолчанию поле модели `FileField` представляется в форме полем типа `FileField`, а поле модели `ImageField` — полем формы `ImageField`. Оба этих типа полей формы объявлены в модуле `django.forms`:

- `FileField` — поле для ввода файла произвольного типа. Дополнительные параметры:
  - `max_length` — максимальная длина пути к файлу, заносимого в поле, в символах;
  - `allow_empty_file` — если `True`, то к выгрузке будут допускаться даже "пустые" файлы (с нулевым размером), если `False` — только файлы с содержимым (ненулевого размера). По умолчанию — `False`;
- `ImageField` — поле для ввода графического файла. Поддерживаются дополнительные параметры `max_length` и `allow_empty_file`, описанные ранее.

Помимо валидаторов, описанных в *разд. 4.7.1*, в полях этих типов можно использовать следующие, объявленные в модуле `django.core.validators`:

- `FileExtensionValidator` — класс, проверяет, входит ли расширение сохраняемого в поле файла в список допустимых. Формат конструктора:

  ```
  FileExtensionValidator(allowed_extensions=<допустимые расширения>[,
                         message=None][, code=None])
  ```

  Он принимает следующие параметры:
  - `allowed_extensions` — последовательность, содержащая допустимые расширения файлов. Каждое расширение представляется в виде строки без начальной точки;
  - `message` — строка с сообщением об ошибке. Если не указан, то выводится стандартное сообщение;
  - `code` — код ошибки. Если не указан, то используется код по умолчанию `"invalid_extension"`;
- `validate_image_file_extension` — переменная, хранит экземпляр класса `FileExtensionValidator`, настроенный считать допустимыми только расширения графических файлов. За основу берется список форматов файлов, поддерживаемых библиотекой Pillow.

Также поддерживаются дополнительные коды ошибок в дополнение к приведенным в *разд. 4.7.2*:

- `"invalid"` — применительно к полю типа `FileField` или `ImageField` сообщает, что задан неверный метод кодирования данных для формы. Следует указать метод `multipart/form-data`, воспользовавшись атрибутом `enctype` тега `<form>`;

- "missing" — файл по какой-то причине не был выгружен;
- "empty" — выгруженный файл "пуст" (имеет нулевой размер);
- "contradiction" — следует либо выбрать файл для выгрузки, либо установить флажок удаления файла из поля, но не одновременно;
- "invalid_extension" — расширение выбранного файла не входит в список допустимых;
- "invalid_image" — графический файл сохранен в неподдерживаемом формате или поврежден.

Для указания выгружаемых файлов применяются такие классы элементов управления из модуля django.forms.widgets:

- FileInput — обычное поле ввода файла;
- ClearableFileInput — поле ввода файла с возможностью очистки. Представляется как комбинация обычного поля ввода файла и флажка очистки, при установке которого сохраненный в поле файл удаляется.

В листинге 20.2 приведен код формы, связанной с моделью Img (см. листинг 20.1) и включающей поле для выгрузки графического изображения ImageField.

Листинг 20.2. Форма с полем для выгрузки файла

```
from django import forms
from django.core import validators

from .models import Img

class ImgForm(forms.ModelForm):
    img = forms.ImageField(label='Изображение',
          validators=[validators.FileExtensionValidator(
          allowed_extensions=('gif', 'jpg', 'png'))],
          error_messages={
                'invalid_extension': 'Этот формат не поддерживается'})
    desc = forms.CharField(label='Описание',
                           widget=forms.widgets.Textarea())

    class Meta:
        model = Img
        fields = '__all__'
```

20.2.3. Обработка выгруженных файлов

Обработка выгруженных файлов в контроллерах осуществляется так же, как обработка любых других данных, полученных от посетителя (см. *разд. 13.2*). Есть лишь два момента, которые нужно иметь в виду:

- при выводе формы на экран необходимо указать для нее метод кодирования данных `multipart/form-data`, воспользовавшись атрибутом `enctype` тега `<form>`:

  ```
  <form . . . enctype="multipart/form-data">
  ```

  Если этого не сделать, то файл не будет выгружен;

- при повторном создании формы конструктору ее класса вторым позиционным параметром следует передать значение атрибута `FILES` объекта запроса. Этот атрибут содержит словарь со всеми выгруженными из формы файлами.

  В листинге 20.3 приведен код контроллера, сохраняющего выгруженный графический файл в модели. Для выгрузки файла используется форма `ImgForm` (см. листинг 20.2).

Листинг 20.3. Контроллер, сохраняющий выгруженный файл

```
from django.shortcuts import render, redirect

from .models import Img
from .forms import ImgForm

def add(request):
    if request.method == 'POST':
        form = ImgForm(request.POST, request.FILES)
        if form.is_valid():
            form.save()
            return redirect('testapp:index')
    else:
        form = ImgForm()
    context = {'form': form}
    return render(request, 'testapp/add.html', context)
```

Сохранение выгруженного файла выполняет сама модель при вызове метода `save()` связанной с ней формы. Нам самим заниматься этим не придется.

Если для выгрузки файла используется форма, не связанная с моделью, то нам понадобится самостоятельно занести выгруженный файл в нужное поле записи модели. Этот файл можно найти в элементе словаря, хранящегося в атрибуте `cleaned_data` объекта формы. Вот пример:

```
form = ImgNonModelForm(request.POST, request.FILES)
if form.is_valid():
    img = Img()
    img.img = form.cleaned_data['img']
    img.desc = form.cleaned_data['desc']
    img.save()
```

И в этом случае сохранение файла выполняется самой моделью.

Аналогичным способом можно сохранять сразу нескольких выгруженных файлов. Сначала следует указать для поля ввода файла возможность выбора произвольного

количества файлов, добавив в создающий его тег <input> атрибут без значения multiple. Пример:

```
class ImgNonModelForm(forms.Form):
    img = forms.ImageField( . . .
       widget=forms.widgets.ClearableFileInput(attrs={'multiple': True}))
    . . .
```

Сложность в том, что элемент словаря из атрибута cleaned_data хранит лишь один из выгруженных файлов. Чтобы получить все файлы, нужно обратиться непосредственно к словарю из атрибута FILES объекта запроса и вызвать у него метод getlist(*<элемент словаря>*). В качестве результата он вернет последовательность файлов, хранящихся в указанном элементе. Вот пример:

```
form = ImgNonModelForm(request.POST, request.FILES)
if form.is_valid():
    for file in request.FILES.getlist('img'):
        img = Img()
        img.desc = form.cleaned_data['desc']
        img.img = file
        img.save()
```

20.2.4. Вывод выгруженных файлов

При обращении непосредственно к полю типа FileField, хранящему выгруженный файл, мы получим экземпляр класса FieldFile, содержащий различные сведения о выгруженном файле. Он поддерживает следующие атрибуты:

- url — интернет-адрес файла:

```
{% for img in imgs %}
    <div><img src="{{ img.img.url }}"></div>
    <div><a href="{{ img.img.url }}">Загрузить картинку</a></div>
{% endfor %}
```

- name — путь к файлу относительно папки, в которой он сохранен (путь к этой папке указывается в параметре MEDIA_ROOT настроек проекта);
- size — размер файла в байтах.

При обращении к полю ImageField мы получим экземпляр класса ImageFieldFile. Он является производным от класса FieldFile, поддерживает все его атрибуты и добавляет два своих собственных:

- width — ширина хранящегося в файле изображения в пикселах;
- height — высота хранящегося в файле изображения в пикселах.

20.2.5. Удаление выгруженного файла

К сожалению, при удалении записи модели, в которой хранится выгруженный файл, сам этот файл удален не будет. Нам придется удалить его самостоятельно.

Для удаления файла применяется метод `delete([save=True])` класса `FieldFile`. Помимо этого, он очищает поле записи, в котором хранится файл. Необязательный параметр `save` указывает, сохранять запись модели после удаления файла (значение `True`, используемое по умолчанию) или нет (значение `False`).

В листинге 20.4 приведен код контроллера, удаляющего файл вместе с записью модели, в которой он хранится. Этот контроллер принимает ключ удаляемой записи с URL-параметром `pk`.

Листинг 20.4. Контроллер, удаляющий выгруженный файл вместе с записью модели, в которой он хранится

```
from django.shortcuts import redirect

def delete(request, pk):
    img = Img.objects.get(pk=pk)
    img.img.delete()
    img.delete()
    return redirect('testapp:index')
```

Можно реализовать удаление сохраненных файлов непосредственно в классе модели, переопределив метод `delete(self, *args, **kwargs)`:

```
class Img(models.Model):
    . . .
    def delete(self, *args, **kwargs):
        self.img.delete(save=False)
        super().delete(*args, **kwargs)
```

20.3. Хранение путей к файлам в моделях

Поле модели, представленное классом `FilePathField` из модуля `django.db.models`, служит для хранения пути к файлу (папке), существующему на диске серверного компьютера и хранящемуся в указанной папке. Отметим, что сохранить путь к несуществующему файлу (папке) в этом поле нельзя.

Конструктор класса `FilePathField` поддерживает следующие дополнительные параметры:

- `path` — полный путь к папке. В поле могут сохраняться только пути к файлам или папкам, вложенным в нее.

  Начиная с Django 3.0, в качестве значения этого параметра может быть указана функция, не принимающая параметров и возвращающая путь к папке;

- `match` — регулярное выражение, записанное в виде строки. Если указано, то в поле могут быть сохранены только пути к файлам, имена которых (не пути целиком!) совпадают с этим регулярным выражением. По умолчанию — `None` (в поле можно сохранить путь к любому файлу из заданной папки);

- `recursive` — если `True`, то в поле можно сохранить путь не только к файлу, хранящемуся непосредственно в заданной папке, но и к любому файлу из вложенных в нее папок. Если `False`, то в поле можно сохранить только путь к файлу из указанной папки. По умолчанию — `False`;
- `allow_files` — если `True`, то в поле можно сохранять пути к файлам, хранящимся в указанной папке, если `False` — нельзя (по умолчанию — `True`);
- `allow_folders` — если `True`, то в поле можно сохранять пути к папкам, вложенным в указанную папку, если `False` — нельзя (по умолчанию — `False`).

> ***Внимание!***
> Допускается указание значения `True` только для одного из параметров: `allow_files` или `allow_folders`.

Для выбора пути к файлу (папке) служит поле формы класса `FilePathField` из модуля `django.forms`. Конструктор поддерживает те же самые параметры `path`, `match`, `recursive`, `allow_files` и `allow_folders`.

Для представления поля типа `FilePathField` на странице применяется список (элемент управления `Select`).

20.4. Низкоуровневые средства для сохранения выгруженных файлов

Низкоуровневые средства предоставляют опытному разработчику полный контроль над сохранением и выводом выгруженных файлов.

20.4.1. Класс *UploadedFile*: выгруженный файл. Сохранение выгруженных файлов

Ранее говорилось, что выгруженные файлы хранятся в словаре, доступном через атрибут `FILES` объекта запроса. Каждый такой файл представляется экземпляром класса `UploadedFile`.

Атрибуты этого класса:

- `name` — изначальное имя выгруженного файла;
- `size` — размер выгруженного файла в байтах;
- `content_type` — MIME-тип файла в виде строки;
- `content_type_extra` — дополнительные параметры MIME-типа файла, представленные в виде словаря;
- `charset` — кодировка, если файл текстовый.

Методы класса `UploadedFile`:

- `multiple_chunks([chunk_size=None])` — возвращает `True`, если файл настолько велик, что для обработки его придется разбивать на отдельные части, и `False`, если он может быть обработан как единое целое.

Необязательный параметр `chunk_size` указывает размер отдельной части (собственно, файл считается слишком большим, если его размер превышает размер части). Если этот параметр не указан, размер принимается равным 64 Кбайт;

- `read()` — считывает и возвращает в качестве результата все содержимое файла.

  Этот метод можно использовать, если файл не слишком велик (метод `multiple_chunks()` возвращает `False`);

- `chunks([chunk_size=None])` — возвращает итератор, который на каждой итерации выдает очередную часть файла.

  Необязательный параметр `chunk_size` указывает размер отдельной части. Если он не указан, размер принимается равным 64 Кбайт.

  Разработчики Django рекомендуют использовать этот метод, если файл слишком велик, чтобы быть обработанным за один раз (метод `multiple_chunks()` возвращает `True`). На практике же его можно применять в любом случае — это позволит упростить код.

В листинге 20.5 приведен код контроллера, сохраняющего выгруженный файл низкоуровневыми средствами.

Листинг 20.5. Контроллер, сохраняющий выгруженный файл низкоуровневыми средствами Django

```
from django.shortcuts import render, redirect
from samplesite.settings import BASE_DIR
from datetime import datetime
import os

from .forms import ImgForm

FILES_ROOT = os.path.join(BASE_DIR, 'files')

def add(request):
    if request.method == 'POST':
        form = ImgForm(request.POST, request.FILES)
        if form.is_valid():
            uploaded_file = request.FILES['img']
            fn = '%s%s' % (datetime.now().timestamp(),
                           os.path.splitext(uploaded_file.name)[1])
            fn = os.path.join(FILES_ROOT, fn)
            with open(fn, 'wb+') as destination:
                for chunk in uploaded_file.chunks():
                    destination.write(chunk)
            return redirect('testapp:index')
    else:
        form = ImgForm()
    context = {'form': form}
    return render(request, 'testapp/add.html', context)
```

Выгруженный файл сохраняется под именем, сформированным на основе текущей временнóй отметки, под изначальным расширением, в папке files, находящейся в папке проекта. Как видим, применяя низкоуровневые средства, файл можно сохранить в произвольной папке.

20.4.2. Вывод выгруженных файлов низкоуровневыми средствами

Вывести список выгруженных файлов можно, выполнив поиск всех файлов в нужной папке и сформировав их список. Для этого удобно применять функцию `scandir()` из модуля `os`.

В листинге 20.6 приведен код контроллера, который выводит список выгруженных файлов, хранящихся в папке files папки проекта.

Листинг 20.6. Контроллер, выводящий список выгруженных файлов

```
from django.shortcuts import render
from samplesite.settings import BASE_DIR
import os

FILES_ROOT = os.path.join(BASE_DIR, 'files')

def index(request):
    imgs = []
    for entry in os.scandir(FILES_ROOT):
        imgs.append(os.path.basename(entry))
    context = {'imgs': imgs}
    return render(request, 'testapp/index.html', context)
```

В шаблоне testapp\index.html нам нужно вывести изображения, хранящиеся в выгруженных файлах. Обратиться к атрибуту `url` мы не можем по вполне понятной причине. Однако мы можем написать еще один контроллер, который получит через URL-параметр имя выгруженного файла и сформирует на его основе ответ — экземпляр класса `FileResponse` (описан в *разд. 9.8.2*). Код этого контроллера приведен в листинге 20.7.

Листинг 20.7. Контроллер, отправляющий выгруженный файл клиенту

```
from django.http import FileResponse

def get(request, filename):
    fn = os.path.join(FILES_ROOT, filename)
    return FileResponse(open(fn, 'rb'),
                        content_type='application/octet-stream')
```

Маршрут, ведущий к этому контроллеру, может быть таким:

```
path('get/<path:filename>', get, name='get'),
```

Обратим внимание, что здесь используется обозначение формата `path`, т. е. любая непустая строка, включающая в себя любые символы.

И наконец, для вывода списка файлов мы напишем в шаблоне testapp\index.html следующий код:

```
{% for img in imgs %}
<div class="image">
    <p><img src="{% url 'testapp:get' filename=img %}"></p>
</div>
{% endfor %}
```

Низкоуровневые средства выгрузки файлов, поддерживаемые Django, основаны на инструментах Python, предназначенных для работы с файлами и папками (как мы только что убедились). Для хранения файлов они не требуют создания модели и имеют более высокое быстродействие. Однако им присущ ряд недостатков: невозможность сохранения дополнительной информации о выгруженном файле (например, описания или сведений о пользователе, выгрузившем файл) и трудности в реализации фильтрации и сортировки файлов по произвольным критериям.

20.5. Библиотека django-cleanup: автоматическое удаление ненужных файлов

В *разд. 20.2.5* говорилось, что при удалении записи модели, которая содержит поле типа `FileField` или `ImageField`, файл, сохраненный в этом поле, не удаляется. Аналогично, при записи в такое поле другого файла старый файл также не удаляется, а остается на диске.

Дополнительная библиотека django-cleanup отслеживает появление ненужных файлов и сама их удаляет. Установить ее можно подачей команды:

```
pip install django-cleanup
```

Ядро этой библиотеки — приложение `django_cleanup`, которое следует добавить в список зарегистрированных в проекте:

```
INSTALLED_APPS = [
    . . .
    'django_cleanup',
]
```

На этом какие-либо действия с нашей стороны закончены. Далее библиотека django-cleanup начнет работать самостоятельно.

> ***На заметку***
>
> Документацию по этой библиотеке можно найти на странице: **https://github.com/un1t/django-cleanup**.

20.6. Библиотека easy-thumbnails: вывод миниатюр

Очень часто при выводе списка графических изображений, хранящихся на сайте, показывают их *миниатюры* — уменьшенные копии. А при переходе на страницу выбранного изображения уже демонстрируют его полную редакцию.

Для автоматического формирования миниатюр согласно заданным параметрам удобно применять дополнительную библиотеку easy-thumbnails.

> ***На заметку***
>
> Полную документацию по библиотеке easy-thumbnails можно найти здесь: **http://easy-thumbnails.readthedocs.io/en/latest/**.

20.6.1. Установка easy-thumbnails

Для установки библиотеки следует набрать в командной строке команду:

```
pip install easy-thumbnails
```

Помимо easy-thumbnails, будет установлена библиотека Pillow, необходимая для работы.

Программное ядро библиотеки реализовано в виде приложения `easy-thumbnails`. Это приложение необходимо добавить в список зарегистрированных в проекте:

```
INSTALLED_APPS = [
    . . .
    'easy_thumbnails',
]
```

Наконец, чтобы все заработало, нужно выполнить миграции.

> ***На заметку***
>
> Для своих нужд easy-thumbnails создает в базе данных таблицы `easy_thumbnails_source`, `easy_thumbnails_thumbnail` и `easy_thumbnails_thumbnaildimensions`.

20.6.2. Настройка easy-thumbnails

Настройки библиотеки, как обычно, записываются в модуле settings.py пакета конфигурации.

20.6.2.1. Пресеты миниатюр

Прежде всего, необходимо указать набор *пресетов* (предопределенных комбинаций настроек), на основе которых будут создаваться миниатюры. Все параметры, которые можно указать в таких пресетах и которые затрагивают создаваемые библиотекой миниатюры, приведены далее:

- `size` — размеры миниатюры. Значением должен быть кортеж из двух элементов: ширины и высоты, заданных в пикселах.

Допускается вместо одного из размеров указывать число 0. Тогда библиотека сама подберет значение этого размера таким образом, чтобы пропорции изображения не искажались.

Примеры:

```
"size": (400, 300) # Миниатюра размерами 400x300 пикселов
"size": (400, 0)   # Миниатюра получит ширину в 400 пикселов,
                   # а высота будет подобрана так, чтобы не допустить
                   # искажения пропорций
"size": (0, 300)   # Миниатюра получит высоту в 300 пикселов,
                   # а ширина будет подобрана так, чтобы не допустить
                   # искажения пропорций
```

- `crop` — управляет обрезкой или масштабированием изображения до размеров, указанных в параметре `size`. Значением может быть одна из строк:
  - `"scale"` — изображение будет масштабироваться до указанных размеров. Обрезка проводиться не будет;
  - `"smart"` — будут обрезаны малозначащие, с точки зрения библиотеки, края изображения ("умная" обрезка);
  - "*<смещение слева>*,*<смещение сверху>*" — явно указывает местоположение фрагмента изображения, который будет вырезан и превращен в миниатюру. Величины *смещения слева* и *сверху* задаются в процентах от ширины и высоты изображения соответственно. Положительные значения указывают собственно смещение слева и сверху левой или верхней границы миниатюры, а отрицательные — смещения справа и снизу ее правой или нижней границы. Если задать значение 0, соответствующая граница миниатюры будет находиться на границе исходного изображения.

  Значение параметра по умолчанию: `"50,50"`;
- `autocrop` — если `True`, то белые поля на границах изображения будут обрезаны;
- `bw` — если `True`, то миниатюра станет черно-белой;
- `replace_alpha` — цвет, которым будет замещен прозрачный цвет в исходном изображении. Цвет указывается в формате *#RRGGBB*, где *RR* — доля красной составляющей, *GG* — зеленой, *BB* — синей. По умолчанию преобразование прозрачного цвета не выполняется;
- `quality` — качество миниатюры в виде числа от 1 (наихудшее качество) до 100 (наилучшее качество). Значение по умолчанию: 85;
- `subsampling` — обозначение уровня подвыборки цвета в виде числа 2 (значение по умолчанию), 1 (более четкие границы, небольшое увеличение размера файла) или 0 (очень четкие границы, значительное увеличение размера файла).

Пресеты записываются в параметре `THUMBNAIL_ALIASES` в виде словаря. Ключи элементов этого словаря указывают области действия пресетов, записанные в одном из следующих форматов:

- "пустая" строка — пресет действует во всех приложениях проекта;
- "*<псевдоним приложения>*" — пресет действует только в приложении с указанным *псевдонимом*;
- "*<псевдоним приложения>.<имя модели>*" — пресет действует только в модели с заданным *именем* в приложении с указанным *псевдонимом*;
- "*<псевдоним приложения>.<имя модели>.<имя поля>*" — пресет действует только для поля с указанным *именем*, в модели с заданным *именем*, в приложении с указанным *псевдонимом*.

Значениями элементов этого словаря должны быть словари, указывающие сами пресеты. Ключи элементов зададут имена пресетов, а элементы, также словари, укажут настройки, относящиеся к соответствующему пресету.

В листинге 20.8 приведен пример указания пресетов для библиотеки easy-thumbnails.

Листинг 20.8. Пример указания пресетов для библиотеки easy-thumbnails

```
THUMBNAIL_ALIASES = {
    'bboard.Bb.picture': {
        'default': {
            'size': (500, 300),
            'crop': 'scale',
        },
    },
    'testapp': {
        'default': {
            'size': (400, 300),
            'crop': 'smart',
            'bw': True,
        },
    },
    '': {
        'default': {
            'size': (180, 240),
            'crop': 'scale',
        },
        'big': {
            'size': (480, 640),
            'crop': '10,10',
        },
    },
}
```

Для поля `picture` модели `Bb` приложения `bboard` мы создали пресет `default`, в котором указали размеры миниатюры 500×300 пикселов и масштабирование без обрезки. Для приложения `testapp` мы также создали пресет `default`, где задали размеры

400×300 пикселов, "умную" обрезку и преобразование в черно-белый вид. А для всего проекта мы расстарались на два пресета: `default` (размеры 180×240 пикселов и масштабирование) и `big` (размеры 480×640 пикселов и обрезка, причем миниатюра будет находиться на расстоянии 10% от левой и верхней границ исходного изображения).

Параметр `THUMBNAIL_DEFAULT_OPTIONS` указывает параметры по умолчанию, применяемые ко всем пресетам, в которых они не были переопределены. Значением этого параметра должен быть словарь, аналогичный тому, который задает параметры отдельного пресета. Пример:

```
THUMBNAIL_DEFAULT_OPTIONS = {'quality': 90, 'subsampling': 1,}
```

20.6.2.2. Остальные параметры библиотеки

Далее перечислены остальные параметры библиотеки easy-thumbnails, которые могут пригодиться:

- `THUMBNAIL_MEDIA_URL` — префикс, добавляемый к интернет-адресу файла со сгенерированной миниатюрой. Если указать "пустую" строку, то будет использоваться префикс из параметра `MEDIA_URL`. По умолчанию — "пустая" строка;
- `THUMBNAIL_MEDIA_ROOT` — полный путь к папке, хранящей файлы с миниатюрами. Если указать "пустую" строку, то будет использована папка из параметра `MEDIA_ROOT`. По умолчанию — "пустая" строка.

  В случае указания параметров `THUMBNAIL_MEDIA_URL` и `THUMBNAIL_MEDIA_ROOT` необходимо записать соответствующий маршрут, чтобы Django смог загрузить созданные библиотекой миниатюры. Код, создающий этот маршрут, аналогичен представленному в *разд. 20.1.2* и может выглядеть так:

  ```
  urlpatterns += static(settings.THUMBNAIL_MEDIA_URL,
                        document_root=settings.THUMBNAIL_MEDIA_ROOT)
  ```

- `THUMBNAIL_BASEDIR` — имя папки, хранящей файлы миниатюр и находящейся в папке из параметра `THUMBNAIL_MEDIA_ROOT`. Так, если задать значение `"thumbs"`, то миниатюра изображения images\others\img1.jpg будет сохранена в файле thumbs\images\others\img1.jpg. По умолчанию — "пустая" строка (т. е. файлы миниатюр будут сохраняться непосредственно в папке из параметра `THUMBNAIL_MEDIA_ROOT`);
- `THUMBNAIL_SUBDIR` — имя вложенной папки, хранящей файлы миниатюр и создаваемой в каждой из вложенных папок, которые есть в папке из параметра `THUMBNAIL_MEDIA_ROOT`. Так, если задать значение `"thumbs"`, то миниатюра изображения images\others\img1.jpg будет сохранена в файле images\others\thumbs\img1.jpg. По умолчанию — "пустая" строка (т. е. вложенные папки для миниатюр создаваться не будут);
- `THUMBNAIL_PREFIX` — префикс, добавляемый в начало имен файлов с миниатюрами (по умолчанию — "пустая" строка);

- THUMBNAIL_EXTENSION — формат файлов для сохранения миниатюр без поддержки прозрачности (по умолчанию: "jpg");
- THUMBNAIL_TRANSPARENCY_EXTENSION — формат файлов для сохранения миниатюр с поддержкой прозрачности (по умолчанию: "png");
- THUMBNAIL_PRESERVE_EXTENSIONS — последовательность расширений файлов, для которых следует создать миниатюры в тех же форматах, в которых были сохранены оригинальные файлы. Расширения должны быть указаны без начальных точек в нижнем регистре. Пример:

  ```
  THUMBNAIL_PRESERVE_EXTENSIONS = ('png',)
  ```

  Теперь для файлов с расширением png будут созданы миниатюры также в формате PNG, а не формате из параметра THUMBNAIL_EXTENSION.

  Если указать значение True, то для файлов всех форматов будут создаваться миниатюры в тех же форматах, что и исходные файлы.

  Значение по умолчанию — None;
- THUMBNAIL_PROGRESSIVE — величина размера изображения в пикселах, при превышении которой изображение будет сохранено в прогрессивном формате JPEG. Учитывается любой размер — как ширина, так и высота. Если указать значение False, то прогрессивный JPEG вообще не будет использоваться. По умолчанию: 100;
- THUMBNAIL_QUALITY — качество изображения JPEG в диапазоне от 1 до 100 (по умолчанию: 85);
- THUMBNAIL_WIDGET_OPTIONS — параметры миниатюры, генерируемой для элемента управления ImageClearableFileInput (будет описан позже). Записываются в виде словаря в том же формате, что и параметры отдельного пресета (см. *разд. 20.6.2.1*). По умолчанию: {'size': (80, 80)} (размеры 80×80 пикселов).

20.6.3. Вывод миниатюр в шаблонах

Прежде чем выводить в шаблонах сгенерированные библиотекой easy-thumbnails миниатюры, нужно загрузить библиотеку тегов с псевдонимом thumbnail:

```
{% load thumbnail %}
```

Для вывода миниатюры можно использовать:

- thumbnail_url:*<название пресета>* — фильтр, выводит интернет-адрес файла с миниатюрой, созданной на основе пресета с указанным *названием* и исходного изображения, взятого из поля типа FileField или ImageField. Если пресета с указанным *названием* нет, будет выведена "пустая" строка. Пример:

  ```
  <img src="{{ img.img|thumbnail_url:'default' }}">
  ```
- thumbnail — тег, выводит интернет-адрес файла с миниатюрой, созданной на основе *исходного изображения*, взятого из поля типа FileField или ImageField. Формат тега:

```
thumbnail <исходное изображение> <название пресета>|<размеры> ↵
[<параметры>] [as <переменная>]
```

Размеры могут быть указаны либо в виде строки формата "*<ширина>*x*<высота>*", либо в виде переменной, которая может содержать строку описанного ранее формата или кортеж из двух элементов, из которых первый укажет ширину, а второй — высоту.

Параметры записываются в том же формате, что и в объявлении пресетов (см. *разд. 20.6.2.1*). Если вместо *размеров* указано *название пресета*, то заданные *параметры* переопределят значения, записанные в пресете.

Примеры:

```
{# Используем пресет default #}
<img src="{% thumbnail img.img 'default' %}">
{# Используем пресет default и дополнительно указываем преобразование
миниатюр в черно-белый вид #}
<img src="{% thumbnail img.img 'default' bw=True %}">
{# Явно указываем размеры миниатюр и "умный" режим обрезки #}
<img src="{% thumbnail img.img '300x200' crop='smart' %}">
```

Мы можем сохранить созданную тегом миниатюру в *переменной*, чтобы использовать ее впоследствии. Эта миниатюра представляется экземпляром класса `ThumbnailFile`, являющегося производным от класса `ImageFieldFile` (см. *разд. 20.2.4*) и поддерживающего те же атрибуты.

Пример:

```
{% thumbnail img.img 'default' as thumb %}
<img src="{{ thumb.url }}">
```

20.6.4. Хранение миниатюр в моделях

Библиотека easy-thumbnails поддерживает два класса полей модели, объявленных в модуле `easy_thumbnails.fields`:

- `ThumbnailerField` — подкласс класса `FileField`. Выполняет часть работ по генерированию миниатюры непосредственно при сохранении записи, а при ее удалении также удаляет все миниатюры, сгенерированные на основе сохраненного в поле изображения. В остальном ведет себя так же, как знакомое нам поле `FileField`;
- `ThumbnailerImageField` — подкласс классов `ImageField` и `ThumbnailerField`.

  Конструктор класса поддерживает дополнительный параметр `resize_source`, задающий параметры генерируемой миниатюры. Эти параметры записываются в виде словаря в том же формате, что и при объявлении пресета (см. *разд. 20.6.2.1*).

  > ***Внимание!***
  >
  > **Если в конструкторе класса `ThumbnailerImageField` указать параметр `resize_source`, в поле будет сохранено не исходное изображение, а сгенерированная на его основе миниатюра.**

Пример:

```
from easy_thumbnails.fields import ThumbnailerImageField
. . .
class Img(models.Model):
    img = ThumbnailerImageField(
          resize_source={'size': (400, 300), 'crop': 'scale'})
    . . .
```

Теперь для вывода миниатюры, сохраненной в поле, можно использовать средства, описанные в *разд. 20.2.4*:

```
<img src="{{ img.img.url }}">
```

Если же параметр `resize_source` в конструкторе поля не указан, то в поле будет сохранено оригинальное изображение, и для вывода миниатюры придется прибегнуть к средствам, описанным в *разд. 20.6.3*.

Эти поля могут представляться в форме элементом управления `ImageClearableFileInput`, класс которого объявлен в модуле `easy_thumbnails.widgets`. Он является подклассом класса `ClearableFileInput`, но дополнительно выводит миниатюру выбранного в нем изображения. Параметры этой миниатюры можно указать в необязательном параметре `thumbnail_options` в виде словаря. Пример:

```
from easy_thumbnails.widgets import ImageClearableFileInput
. . .
class ImgForm(forms.Form):
    img = forms.ImageField(widget=ImageClearableFileInput(
                          thumbnail_options={'size': (300, 200)}))
    . . .
```

Если параметры миниатюры для этого элемента не указаны, то они будут взяты из параметра `THUMBNAIL_WIDGET_OPTIONS` настроек проекта (см. *разд. 20.6.2.2*).

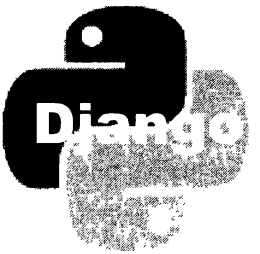
20.6.5. Дополнительная команда *thumbnail_cleanup*

Библиотека easy-thumbnails добавляет утилите manage.py поддержку команды `thumbnail_cleanup`, удаляющей все сгенерированные миниатюры. Формат ее вызова следующий:

```
manage.py thumbnail_cleanup [--last-n-days <количество дней>]
[--path <путь для очистки>] [--dry-run]
```

Поддерживаются следующие дополнительные ключи:

- `--last-n-days` — оставляет миниатюры, сгенерированные в течение указанного *количества дней*. Если не задан, будут удалены все миниатюры;
- `--path` — удаляет миниатюры, хранящиеся по указанному *пути*. Если не указан, будут удалены все миниатюры;
- `--dry-run` — выводит на экран сведения об удаляемых миниатюрах, но не удаляет их.



ГЛАВА 21

Разграничение доступа: расширенные инструменты и дополнительная библиотека

Подсистема разграничения доступа, реализованная в Django, предоставляет ряд инструментов, полезных при программировании на низком уровне.

21.1. Настройки проекта, касающиеся разграничения доступа

Немногочисленные настройки, затрагивающие работу подсистемы разграничения доступа, записываются в модуле settings.py пакета приложения:

- `AUTH_PASSWORD_VALIDATORS` — список валидаторов, применяемых при валидации пароля, который пользователь заносит при регистрации. Каждый элемент списка задает один валидатор и должен представлять собой словарь с элементами `NAME` (задает имя класса валидатора в виде строки) и `OPTIONS` (словарь с дополнительными параметрами валидатора).

  Валидаторы, приведенные в списке, задействуются в формах для смены и сброса пароля, в командах создания суперпользователя и смены пароля. Во всех прочих случаях они никак не используются.

  Значение по умолчанию — "пустой" список, однако сразу при создании проекта этому параметру присваивается список из всех валидаторов, поставляемых в составе Django с параметрами по умолчанию;

- `AUTHENTICATION_BACKENDS` — список имен классов, реализующих аутентификацию и авторизацию, представленных в виде строк. По умолчанию — список с единственным элементом `"django.contrib.auth.backends.ModelBackend"` (этот класс реализует аутентификацию и авторизацию пользователей из списка, хранящегося в модели);

- `AUTH_USER_MODEL` — имя класса модели, хранящей список зарегистрированных пользователей, в виде строки. По умолчанию: `"auth.User"` (стандартная модель `User`).

21.2. Работа с пользователями

Django предлагает ряд инструментов для работы с пользователями: их создания, смены пароля и пр.

21.2.1. Создание пользователей

Для создания пользователя применяются два описанных далее метода, поддерживаемые классом диспетчера записей `UserManager`, который используется в модели `User`:

- `create_user(<имя>, password=<пароль>[, email=<адрес электронной почты>][, <дополнительные поля>])` — создает и сразу сохраняет нового пользователя с указанными *именем*, *паролем* и *адресом электронной почты* (если он задан). Также могут быть указаны значения для *дополнительных полей*, которые будут сохранены в модели пользователя. Созданный пользователь делается активным (полю `is_active` присваивается значение `True`) и возвращается в качестве результата. Примеры:

```
user1 = User.objects.create_user('ivanov', password='1234567890',
                                 email='ivanov@site.ru')
user2 = User.objects.create_user('petrov', password='0987654321',
                                 email='petrov@site.ru', is_stuff=True)
```

- `create_superuser(<имя>, <адрес электронной почты>, <пароль>[, <дополнительные поля>])` — создает и сразу сохраняет нового суперпользователя с указанными *именем*, *паролем* и *адресом электронной почты* (если он задан). Также могут быть указаны значения для *дополнительных полей* модели пользователя. Созданный суперпользователь делается активным (полю `is_active` присваивается значение `True`) и возвращается в качестве результата.

21.2.2. Работа с паролями

Еще четыре метода, поддерживаемые моделью `User`, предназначены для работы с паролями:

- `check_password(<пароль>)` — возвращает `True`, если заданный *пароль* совпадает с хранящимся в списке, и `False` — в противном случае:

```
from django.contrib.auth.models import User
admin = User.objects.get(name='admin')
if admin.check_password('password'):
    # Пароли совпадают
else:
    # Пароли не совпадают
```

- `set_password(<новый пароль>)` — задает для текущего пользователя *новый пароль*. Сохранение пользователя не выполняет. Пример:

```
admin.set_password('newpassword')
admin.save()
```

- `set_unusable_password()` — задает для текущего пользователя недействительный пароль. При проверке такого пароля функцией `check_password()` последняя всегда будет возвращать `False`. Сохранение пользователя не выполняет.

  Недействительный пароль указывается у тех пользователей, для которых процедура входа на сайт выполняется не средствами Django, а какой-либо сторонней библиотекой — например, Python Social Auth, описываемой далее;
- `has_usable_password()` — возвращает `True`, если текущий пользователь имеет действительный пароль, и `False`, если его пароль недействителен (была вызвана функция `set_unusable_password()`).

21.3. Аутентификация и выход с сайта

Аутентификация, т. е. вход на сайт, с применением низкоуровневых инструментов выполняется в два этапа: поиск пользователя в списке и собственно вход. Здесь нам понадобятся три функции, объявленные в модуле `django.contrib.auth`.

Для поиска пользователя по указанным им на странице входа имени и паролю применяется функция `authenticate()`:

```
authenticate(<запрос>, username=<имя>, password=<пароль>)
```

Запрос должен быть представлен экземпляром класса `HttpRequest`. Если пользователь с указанными *именем* и *паролем* существует в списке, функция возвращает представляющую его запись модели `User`. В противном случае возвращается `None`.

Собственно вход выполняется вызовом функции `login(<запрос>, <пользователь>)`. *Запрос* должен быть представлен экземпляром класса `HttpRequest`, а *пользователь*, от имени которого выполняется вход, — записью модели `User`.

Вот пример кода, получающего в POST-запросе данные из формы входа и выполняющего вход на сайт:

```
from django.contrib.auth import authenticate, login

def my_login(request):
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(request, username=username, password=password)
    if user is not None:
        login(request, user)
        # Вход выполнен
    else:
        # Вход не был выполнен
```

Выход с сайта выполняется вызовом функции `logout(<запрос>)`. *Запрос* должен быть представлен экземпляром класса `HttpRequest`. Пример:

```
from django.contrib.auth import logout

def my_logout (request):
    logout(request)
    # Выход был выполнен. Выполняем перенаправление на какую-либо страницу
```

21.4. Валидация паролей

В *разд. 21.1* описывался параметр `AUTH_PASSWORD_VALIDATORS` настроек проекта, задающий набор валидаторов паролей. Эти валидаторы будут работать в формах для смены и сброса пароля, в командах создания суперпользователя и смены пароля.

Значение этого параметра по умолчанию — "пустой" список. Однако сразу при создании проекта для него задается такое значение:

```
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth.password_validation.' + \
                'UserAttributeSimilarityValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.' + \
                'MinimumLengthValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.' + \
                'CommonPasswordValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.' + \
                'NumericPasswordValidator',
    },
]
```

Это список, включающий все (четыре) стандартные валидаторы, поставляемые в составе Django.

21.4.1. Стандартные валидаторы паролей

Все стандартные валидаторы реализованы в виде классов и объявлены в модуле `django.contrib.auth.password_validation`:

- `UserAttributeSimilarityValidator()` — позволяет удостовериться, что пароль в достаточной степени отличается от остальных сведений о пользователе. Формат вызова конструктора:

```
UserAttributeSimilarityValidator(
    [user_attributes=self.DEFAULT_USER_ATTRIBUTES][,]
    [max_similarity=0.7])
```

Необязательный параметр `user_attributes` задает последовательность имен полей модели `User`, из которых будут браться сведения о пользователе для сравнения с паролем, имена полей должны быть представлены в виде строк. По умолчанию берется кортеж, хранящийся в атрибуте `DEFAULT_USER_ATTRIBUTES` этого же класса и перечисляющий поля `username`, `first_name`, `last_name` и `email`.

Необязательный параметр `max_similarity` задает степень схожести пароля со значением какого-либо из полей, указанных в последовательности `user_attributes`. Значение параметра должно представлять собой вещественное число от `0` (будут отклоняться все пароли без исключения) до `1` (будут отклоняться только пароли, полностью совпадающие со значением поля). По умолчанию установлено значение `0.7`;

- `MinimumLengthValidator([min_length=8])` — проверяет, не оказались ли длина пароля меньше заданной в параметре `min_length` (по умолчанию `8` символов);
- `CommonPasswordValidator()` — проверяет, не входит ли пароль в указанный перечень наиболее часто встречающихся паролей. Формат вызова конструктора:

```
CommonPasswordValidator(
            [password_list_path=self.DEFAULT_PASSWORD_LIST_PATH])
```

Необязательный параметр `password_list_path` задает полный путь к файлу со списком недопустимых паролей. Этот файл должен быть сохранен в текстовом формате, а каждый из паролей должен находиться на отдельной строке. По умолчанию используется файл с порядка 1000 паролей, путь к которому хранится в атрибуте `DEFAULT_PASSWORD_LIST_PATH` класса;

- `NumericPasswordValidator` — проверяет, не содержит ли пароль одни цифры.

Вот пример кода, задающего новый список валидаторов:

```
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth.password_validation.' + \
                'MinimumLengthValidator',
        'OPTIONS": {'min_length': 10}
    },
    {
        'NAME': 'django.contrib.auth.password_validation.' + \
                'NumericPasswordValidator',
    },
]
```

Новый список содержит только валидаторы `MinimumLengthValidator` и `NumericPasswordValidator`, причем для первого указана минимальная длина пароля `10` символов.

21.4.2. Написание своих валидаторов паролей

Валидаторы паролей обязательно должны быть реализованы в виде классов, поддерживающих два метода:

- `validate(self, password, user=None)` — выполняет валидацию пароля, получаемого с параметром `password`. С необязательным параметром `user` может быть получен текущий пользователь.

  Метод не должен возвращать значения. Если пароль не проходит валидацию, то следует возбудить исключение `ValidationError` из модуля `django.core.exceptions`;

- `get_help_text(self)` — должен возвращать строку с требованиями к вводимому паролю.

В листинге 21.1 приведен код валидатора `NoForbiddenCharsValidator`, проверяющего, не содержатся ли в пароле недопустимые символы, заданные в параметре `forbidden_chars`.

Листинг 21.1. Пример валидатора паролей

```
from django.core.exceptions import ValidationError

class NoForbiddenCharsValidator:
    def __init__(self, forbidden_chars=(' ',)):
        self.forbidden_chars = forbidden_chars

    def validate(self, password, user=None):
        for fc in self.forbidden_chars:
            if fc in password:
                raise ValidationError(
                      'Пароль не должен содержать недопустимые ' + \
                      'символы %s' % ', '.join(self.forbidden_chars),
                      code='forbidden_chars_present')

    def get_help_text(self):
        return 'Пароль не должен содержать недопустимые символы %s' % \
               ', '.join(self.forbidden_chars)
```

Такой валидатор может быть использован наряду со стандартными:

```
AUTH_PASSWORD_VALIDATORS = [
    . . .
    {
        'NAME': 'NoForbiddenCharsValidator',
        'OPTIONS': {'forbidden_chars': (' ', ',', '.', ':', ';')},
    },
]
```

21.4.3. Выполнение валидации паролей

Валидаторы из параметра `AUTH_PASSWORD_VALIDATORS` используются в ограниченном количестве случаев. Чтобы осуществить валидацию пароля там, где нам нужно (например, в написанной нами самими форме), мы прибегнем к набору функций из модуля `django.contrib.auth.password_validation`:

- `validate_password(<пароль>[, user=None][, password_validators=None])` — выполняет валидацию *пароля*. Если пароль не проходит валидацию, возбуждает исключение `ValidationError`;
- `password_validators_help_texts([password_validators=None])` — возвращает список строк, содержащий требования к вводимым паролям от всех валидаторов. Строка с такими требованиями возвращается методом `get_help_text()` валидатора (см. *разд. 21.4.2*);
- `password_validators_help_texts_html([password_validators=None])` — то же самое, что и `password_validators_help_text()`, но возвращает HTML-код, создающий маркированный список со всеми требованиями;
- `password_changed(<пароль>[, user=None][, password_validators=None])` — сообщает всем валидаторам, что пароль пользователя изменился.

  Вызов этой функции следует выполнять сразу после каждой смены пароля, если для этого не использовалась функция `set_password()`, описанная в *разд. 21.2.2*. После выполнения функции `set_password()` функция `password_changed()` вызывается автоматически.

Необязательный параметр `user`, принимаемый большинством функций, задает пользователя, чей пароль проходит валидацию. Это значение может понадобиться некоторым валидаторам.

Необязательный параметр `password_validators`, поддерживаемый всеми этими функциями, указывает список валидаторов, которые будут заниматься валидацией пароля. Если он не указан, используется список из параметра `AUTH_PASSWORD_VALIDATORS` настроек проекта.

Чтобы сформировать свой список валидаторов, следует применить функцию `get_password_validators(<настройки валидаторов>)`. *Настройки валидаторов* указываются в том же формате, что и значение параметра `AUTH_PASSWORD_VALIDATORS` настроек проекта. Пример:

```
from django.contrib.auth import password_validation
my_validators = [
    {
        'NAME': 'django.contrib.auth.password_validation.' + \
                'NumericPasswordValidator',
    },
    {
        'NAME': 'NoForbiddenCharsValidator',
        'OPTIONS': {'forbidden_chars': (' ', ',', '.', ':', ';')},
    },
]
validator_config = password_validation.get_password_validators(my_validators)
password_validation.validate_password(password, validator_config)
```

21.5. Библиотека Python Social Auth: регистрация и вход через социальные сети

В настоящее время очень и очень многие пользователи Интернета являются подписчиками какой-либо социальной сети, а то и не одной. Неудивительно, что появились решения, позволяющие выполнять регистрацию в списках пользователей различных сайтов и вход на них посредством социальных сетей. Одно из таких решений — дополнительная библиотека Python Social Auth.

Python Social Auth позволяет выполнять вход посредством более чем 100 социальных сетей и интернет-сервисов, включая "ВКонтакте", Facebook, Twitter, GitHub, Instagram и др. Отметим, что она поддерживает не только Django, но и ряд других веб-фреймворков, написанных на Python.

В этой главе рассмотрены установка и использование библиотеки для выполнения регистрации и входа на сайт посредством социальной сети "ВКонтакте".

> ***На заметку***
>
> Полное руководство по Python Social Auth располагается здесь: **https://python-social-auth.readthedocs.io/**.

21.5.1. Создание приложения "ВКонтакте"

Чтобы успешно выполнять регистрацию и вход на сайт посредством "ВКонтакте", необходимо предварительно зарегистрировать в этой социальной сети новое приложение. Вот шаги, которые надо произвести:

1. Выполните вход в социальную сеть "ВКонтакте". Если вы не являетесь подписчиком этой сети, предварительно зарегистрируйтесь в ней.
2. Перейдите на страницу списка приложений, созданных текущим пользователем, которая находится по интернет-адресу **https://vk.com/apps?act=manage**.
3. Перейдите на страницу создания нового приложения, нажав кнопку **Создать приложение**.
4. Заполните форму со сведениями о создаваемом приложении (рис. 21.1). Название приложения, заносимое в одноименное поле ввода, может быть каким угодно. В группе **Платформа** следует выбрать переключатель **Веб-сайт**. В поле ввода **Адрес сайта** заносится интернет-адрес сайта, для которого необходимо реализовать вход через сеть "ВКонтакте", а в поле ввода **Базовый домен** — его домен. Если сайт в данный момент еще разрабатывается и развернут на локальном хосте, следует ввести интернет-адрес **http://localhost** и домен **localhost** соответственно. Введя все нужные данные, нажмите кнопку **Подключить сайт**.
5. Возможно, придется запросить SMS-подтверждение на создание нового приложения. Сделайте это, следуя появляющимся на экране инструкциям.
6. Щелкните на гиперссылке **Настройки**, находящейся на следующей странице, где выводятся полные сведения о созданном приложении. В появившейся на

экране форме настроек приложения, на самом ее верху, найдите поля **ID приложения** и **Защищенный ключ** (рис. 21.2). Эти величины лучше переписать куда-нибудь — они нам скоро пригодятся.

Внимание!

ID приложения и в особенности его защищенный ключ необходимо хранить в тайне.

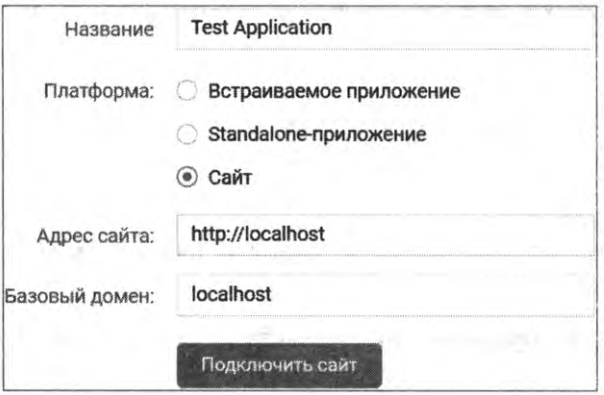


Рис. 21.1. Форма для создания нового приложения "ВКонтакте"

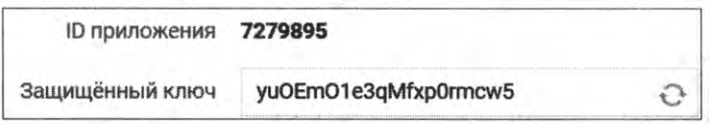


Рис. 21.2. Поля ID приложения и Защищенный ключ, находящиеся в форме настроек приложения "ВКонтакте"

21.5.2. Установка и настройка Python Social Auth

Для установки редакции библиотеки, предназначенной для Django, необходимо в командной строке подать команду:

```
pip install social-auth-app-django
```

Одновременно с самой Python Social Auth будет установлено довольно много других библиотек, используемых ею в работе.

После установки следует выполнить следующие шаги:

- зарегистрировать в проекте приложение `social_django` — программное ядро библиотеки:

```
INSTALLED_APPS = [
    . . .
    'social_django',
]
```

- выполнить миграции, чтобы приложение создало в базе данных все необходимые для своих моделей структуры;

- если используется СУБД PostgreSQL — добавить в модуль settings.py пакета конфигурации такой параметр:

```
SOCIAL_AUTH_POSTGRES_JSONFIELD = True
```

Он разрешает хранение данных поля типа `JSONField`, поддерживаемого этой СУБД (см. *разд. 18.1.1.1*);

- добавить в список классов, реализующих аутентификацию и авторизацию, класс `social_core.backends.vk.VKOAuth2`:

```
AUTHENTICATION_BACKENDS = (
    'social_core.backends.vk.VKOAuth2',
    'django.contrib.auth.backends.ModelBackend',
)
```

Параметр `AUTHENTICATION_BACKENDS` придется добавить в модуль settings.py, т. к. изначально его там нет;

- добавить в список обработчиков контекста для используемого нами шаблонизатора классы `social_django.context_processors.backends` и `social_django.context_processors.login_redirect`:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        . . .
        'OPTIONS': {
            'context_processors': [
                . . .
                'social_django.context_processors.backends',
                'social_django.context_processors.login_redirect',
            ],
            . . .
        },
    },
]
```

- добавить в модуль settings.py параметры, указывающие полученные ранее ID приложения и защищенный ключ:

```
SOCIAL_AUTH_VK_OAUTH2_KEY = 'XXXXXXX'
SOCIAL_AUTH_VK_OAUTH2_SECRET = 'XXXXXXXXXXXXXXXXXXXX'
```

- если необходимо, помимо всех прочих сведений о пользователе, получать от сети "ВКонтакте" еще и его адрес электронной почты — добавить в модуль settings.py такой параметр:

```
SOCIAL_AUTH_VK_OAUTH2_SCOPE = ['email']
```

21.5.3. Использование Python Social Auth

Сначала нужно создать маршруты, которые ведут на контроллеры, выполняющие регистрацию и вход. Эти маршруты добавляются в список маршрутов уровня проекта — в модуль urls.py пакета конфигурации. Вот пример:

```
urlpatterns = [
    . . .
    path('social/', include('social_django.urls', namespace='social')),
]
```

Префикс, указываемый в первом параметре функции `path()`, может быть любым.

Далее нужно добавить на страницу входа гиперссылку на контроллер, выполняющий вход на сайт и, если это необходимо, регистрацию нового пользователя на основе сведений, полученных от сети "ВКонтакте". Вот код, создающий эту гиперссылку:

```
<a href="{% url 'social:begin' 'vk-oauth2' %}">Войти через ВКонтакте</a>
```

При щелчке на такой гиперссылке появится окно с формой для входа в сеть "ВКонтакте". После успешного выполнения входа пользователь будет перенаправлен на сайт по пути, записанному в параметре `LOGIN_REDIRECT_URL` настроек проекта (см. *разд. 15.2.1*). Если же пользователь ранее выполнил вход на сеть "ВКонтакте", он будет просто перенаправлен по пути, записанному в упомянутом ранее параметре.

21.6. Создание своей модели пользователя

Для хранения списка пользователей в подсистеме разграничения доступа Django предусмотрена стандартная модель `User`, объявленная в модуле `django.contrib.auth.models`. Эта модель хранит объем сведений о пользователе, вполне достаточный для многих случаев. Однако часто приходится сохранять в составе сведений о пользователе дополнительные данные: номер телефона, интернет-адрес сайта, признак, хочет ли пользователь получать по электронной почте уведомления о новых сообщениях, и т. п.

Можно объявить дополнительную модель, поместить в нее поля для хранения всех нужных данных и добавить поле, устанавливающее связь "один-с-одним" со стандартной моделью пользователя. Вот пример создания подобной дополнительной модели:

```
from django.db import models
from django.contrib.auth.models import User
class Profile(models.Model):
    phone = models.CharField(max_length=20)
    user = models.OneToOneField(User, on_delete=models.CASCADE)
```

Разумеется, при создании нового пользователя придется явно создавать связанную с ним запись модели, хранящую дополнительные сведения. Зато не будет никаких проблем с подсистемой разграничения доступа и старыми дополнительными биб-

лиотеками, поскольку для хранения основных сведений о пользователях будет использоваться стандартная модель `User`.

Другой подход заключается в написании своей собственной модели пользователя. Такую модель следует сделать производной от класса `AbstractUser`, который объявлен в модуле `django.contrib.auth.models`, реализует всю функциональность по хранению пользователей и представляет собой абстрактную модель (см. *разд. 16.4.2*) — собственно, класс стандартной модели пользователей `User` также является производным от класса `AbstractUser`. Пример:

```
from django.db import models
from django.contrib.auth.models import AbstractUser
class AdvUser(AbstractUser):
    phone = models.CharField(max_length=20)
```

Новую модель пользователя следует указать в параметре `AUTH_USER_MODEL` настроек проекта (см. *разд. 21.1*):

```
AUTH_USER_MODEL = 'testapp.models.AdvUser'
```

В таком случае не придется самостоятельно создавать связанные записи, хранящие дополнительные сведения о пользователе, — это сделает Django. Однако нужно быть готовым к тому, что некоторые дополнительные библиотеки, в особенности старые, не считывают имя модели пользователя из параметра `AUTH_USER_MODEL`, а обращаются напрямую к модели `User`. Если такая библиотека добавит в список нового пользователя, то он будет сохранен без дополнительных сведений, и код, использующий эти сведения, не будет работать.

Если нужно лишь расширить или изменить функциональность модели пользователя, то можно создать на его основе прокси-модель, также не забыв занести ее в параметр `AUTH_USER_MODEL`:

```
from django.db import models
from django.contrib.auth.models import User
class AdvUser(User):
    . . .
    class Meta:
        proxy = True
```

И наконец, можно написать полностью свой класс модели. Однако такой подход применяется весьма редко из-за его трудоемкости. Интересующиеся могут обратиться к странице **https://docs.djangoproject.com/en/3.0/topics/auth/customizing/**, где приводятся все нужные инструкции.

21.7. Создание своих прав пользователя

В *главе 15* описывались права, определяющие операции, которые пользователь может выполнять над записями какой-либо модели. Изначально для каждой модели создаются четыре стандартных права: на просмотр, добавление, правку и удаление записей.

Для любой модели можно создать дополнительный набор произвольных прав. Для этого мы воспользуемся параметром `permissions`, задаваемым для самой модели — во вложенном классе `Meta`. В качестве его значения указывается список или кортеж, каждый элемент которого описывает одно право и также представляет собой кортеж из двух элементов: обозначения, используемого самим Django, и наименования, предназначенного для вывода на экран. Пример:

```
class Comment(models.Model):
    . . .
    class Meta:
        permissions = (
            ('hide_comments', 'Можно скрывать комментарии'),
        )
```

Обрабатываются эти права точно так же, как и стандартные. В частности, можно программно проверить, имеет ли текущий пользователь право скрывать комментарии:

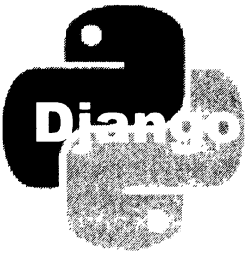
```
def hide_comment(request):
    if request.user.has_perm('bboard.hide_comments'):
        # Пользователь может скрывать комментарии
```

Можно также изменить набор стандартных прав, создаваемых для каждой модели самим Django. Правда, автору книги не понятно, зачем это может понадобиться...

Стандартные права указываются в параметре модели `default_permissions` в виде списка или кортежа, содержащего строки с их наименованиями: `"view"` (просмотр), `"add"` (добавление), `"change"` (правка) и `"delete"` (удаление). Вот пример указания у модели только прав на правку и удаление записей:

```
class Comment(models.Model):
    . . .
    class Meta:
        default_permissions = ('change', 'delete')
```

Значение параметра `default_permissions` по умолчанию: `('view', 'add', 'change', 'delete')` — т. е. полный набор стандартных прав.



ГЛАВА 22

Посредники и обработчики контекста

Посредник (middleware) Django — это программный модуль, выполняющий предварительную обработку клиентского запроса перед передачей его контроллеру и окончательную обработку ответа, выданного контроллером, перед отправкой его клиенту. Список посредников, зарегистрированных в проекте, указывается в параметре `MIDDLEWARE` настроек проекта (см. *разд. 3.3.4*).

Посредники в Django можно использовать не только для обработки запросов и ответов, но и для добавления в контекст шаблона каких-либо значений. Ту же самую задачу выполняют и обработчики контекста, список которых указывается в дополнительном параметре `context_processors` настроек шаблонизатора (см. *разд. 11.1*).

22.1. Посредники

Посредники — весьма мощный инструмент по обработке данных, пересылаемых по сети. Немалая часть функциональности Django реализована именно в посредниках.

22.1.1. Стандартные посредники

Посредники, изначально включенные в список параметра `MIDDLEWARE` настроек проекта, были описаны в *разд. 3.3.4*. Помимо них, в составе Django имеется еще ряд посредников:

- `django.middleware.gzip.GZipMiddleware` — сжимает запрашиваемую страницу с применением алгоритма GZip, если размер страницы превышает 200 байтов, страница не была сжата на уровне контроллера (для чего достаточно указать у него декоратор `gzip_page()`, описанный в *разд. 9.11*), а веб-обозреватель способен обрабатывать сжатые страницы.

  В списке зарегистрированных посредников должен находиться перед теми, которые получают доступ к содержимому ответа с целью прочитать или изменить его, и после посредника `django.middleware.cache.UpdateCacheMiddleware`;

- `django.middleware.http.ConditionalGetMiddleware` — выполняет обработку заголовков, связанных с кэшированием страниц на уровне клиента. Если ответ не имеет заголовка `E-Tag`, такой заголовок будет добавлен. Если ответ имеет заголовки `E-Tag` или `Last-Modified`, а запрос — заголовки `If-None-Match` или `If-Modified-Since`, то вместо страницы будет отправлен "пустой" ответ с кодом 304 (запрашиваемая страница не была изменена).

  В списке зарегистрированных посредников должен находиться перед `django.middleware.common.CommonMiddleware`;

- `django.middleware.cache.UpdateCacheMiddleware` — обновляет кэш при включенном режиме кэширования всего сайта.

  В списке зарегистрированных посредников должен находиться перед теми, которые модифицируют заголовок `Vary` (`django.contrib.sessions.middleware.SessionMiddleware` и `django.middleware.gzip.GZipMiddleware`);

- `django.middleware.cache.FetchFromCacheMiddleware` — извлекает запрошенную страницу из кэша при включенном режиме кэширования всего сайта.

  В списке зарегистрированных посредников должен находиться после тех, которые модифицируют заголовок `Vary` (`django.contrib.sessions.middleware.SessionMiddleware` и `django.middleware.gzip.GZipMiddleware`).

  О кэшировании будет рассказано в *главе 26*.

Любой из этих посредников в случае необходимости можно вставить в список параметра `MIDDLEWARE` настроек проекта согласно указаниям касательно очередности их следования.

22.1.2. Порядок выполнения посредников

Посредники, зарегистрированные в проекте, при получении запроса и формировании ответа выполняются дважды.

1. Первый раз — при получении запроса, перед передачей его контроллеру, в том порядке, в котором записаны в списке параметра `MIDDLEWARE` настроек проекта.
2. Второй раз — после того, как контроллер сгенерирует ответ, до отправки его клиенту. Если ответ представлен экземпляром класса `TemplateResponse`, то посредники выполняются до непосредственного рендеринга шаблона (что позволяет изменить некоторые параметры запроса — например, добавить какие-либо данные в контекст шаблона). Порядок выполнения посредников на этот раз противоположен тому, в каком они записаны в списке параметра `MIDDLEWARE`.

Рассмотрим для примера посредники, зарегистрированные во вновь созданном проекте:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    . . .
```

```
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

При получении запроса сначала будет выполнен самый первый в списке посредник `django.middleware.security.SecurityMiddleware`, далее — `django.contrib.sessions.middleware.SessionMiddleware` и т. д. После выполнения посредника `django.middleware.clickjacking.XFrameOptionsMiddleware`, последнего в списке, управление будет передано контроллеру.

После того как контроллер сгенерирует ответ, выполнится последний в списке посредник `django.middleware.clickjacking.XFrameOptionsMiddleware`, за ним — предпоследний `django.contrib.messages.middleware.MessageMiddleware` и т. д. После выполнения самого первого в списке посредника `django.middleware.security.SecurityMiddleware` ответ отправится клиенту.

22.1.3. Написание своих посредников

Если разработчику не хватает стандартных посредников, он может написать свой собственный, реализовав его в виде функции или класса.

22.1.3.1. Посредники-функции

Посредники-функции проще в написании, но предоставляют не очень много функциональных возможностей.

Посредник-функция должен принимать один параметр. С ним будет передан либо следующий в списке посредник (если это не последний посредник в списке), либо контроллер (если текущий посредник — последний в списке).

Посредник-функция в качестве результата должна возвращать функцию, в качестве единственного параметра принимающую запрос в виде экземпляра класса `HttpRequest`. В этой "внутренней" функции и будет выполняться предварительная обработка запроса и окончательная — ответа в следующей последовательности:

- если нужно — выполняется предварительная обработка запроса, получаемого возвращаемой функцией в единственном параметре (здесь можно, например, изменить содержимое запроса и добавить в него новые атрибуты);

  > ***Внимание!***
  >
  > Содержимое можно изменить только у обычного, непотокового ответа. Объект потокового ответа не поддерживает атрибут `content`, поэтому добраться до его содержимого невозможно (подробнее о потоковом ответе — в *разд. 9.8.1*).

- обязательно — вызывается функция, полученная с параметром посредником-функцией. В качестве единственного параметра полученной функции передается объект запроса, а в качестве результата она вернет ответ в виде экземпляра класса `HttpResponse`;
- если нужно — выполняется окончательная обработка ответа, ранее возвращенного полученной функцией;

- обязательно — объект ответа возвращается из "внутренней" функции в качестве результата.

Вот своеобразный шаблон, согласно которому пишутся посредники-функции:

```
def my_middleware(next):
    # Здесь можно выполнить какую-либо инициализацию

    def core_middleware(request):
        # Здесь выполняется обработка клиентского запроса

        response = next(request)

        # Здесь выполняется обработка ответа

        return response

    return core_middleware
```

Регистрируется такой посредник следующим образом (подразумевается, что он объявлен в модуле middleware.py пакета приложения `bboard`):

```
MIDDLEWARE = [
    . . .
    'bboard.middleware.my_middleware',
    . . .
]
```

22.1.3.2. Посредники-классы

Посредники-классы предлагают больше функциональных возможностей, но писать их несколько сложнее.

Посредник-класс должен объявлять, по меньшей мере, два метода:

- конструктор `__init__(self, next)` — должен принять в параметре `next` либо следующий в списке посредник, либо контроллер (если посредников больше нет) и сохранить его. Также может выполнить какую-либо инициализацию.

  Если в теле конструктора возбудить исключение `MiddlewareNotUsed` из модуля `django.core.exceptions`, то посредник деактивируется и более не будет использоваться в дальнейшем;

- `__call__(self, request)` — должен принимать в параметре `request` объект запроса и возвращать объект ответа. Тело этого метода пишется по тем же правилам, что и тело "внутренней" функции у посредника-функции.

Далее приведен аналогичный шаблон исходного кода, согласно которому пишутся посредники-классы:

```
class MyMiddleware:
    def __init__(self, next):
        self.next = next
        # Здесь можно выполнить какую-либо инициализацию
```

```
def __call__(self, request):
    # Здесь выполняется обработка клиентского запроса

    response = self.next(request)

    # Здесь выполняется обработка ответа

    return response
```

Дополнительно в посреднике-классе можно объявить следующие методы:

- `process_view(self, request, view_func, view_args, view_kwargs)` — выполняется непосредственно перед вызовом следующего в списке посредника или контроллера (если это последний посредник в списке).

  Параметром `request` методу передается запрос в виде экземпляра класса `HttpRequest`, параметром `view_func` — ссылка на функцию, реализующую контроллер. Это может быть контроллер-функция или функция, возвращенная методом `as_view()` контроллера-класса. Параметром `view_args` методу передается список позиционных URL-параметров (в текущих версиях Django не используется), а параметром `view_kwargs` — словарь с именованными URL-параметрами, передаваемыми контроллеру.

  Метод должен возвращать один из перечисленных далее результатов:

  - `None` — тогда обработка запроса продолжится: будет вызван следующий в списке посредник или контроллер;
  - экземпляр класса `HttpResponse` (т. е. ответ) — тогда обработка запроса прервется, и возвращенный методом ответ будет отправлен клиенту;

- `process_exception(self, request, exception)` — вызывается при возбуждении исключения в теле контроллера. Параметром `request` методу передается запрос в виде экземпляра класса `HttpRequest`, параметром `exception` — само исключение в виде экземпляра класса `Exception`.

  Метод должен возвращать:

  - `None` — тогда будет выполнена обработка исключения по умолчанию;
  - экземпляр класса `HttpResponse` (т. е. ответ) — тогда возвращенный методом ответ будет отправлен клиенту;

- `process_template_response(self, request, response)` — вызывается уже после того, как контроллер сгенерировал ответ, но перед рендерингом шаблона. Параметром `request` методу передается запрос в виде экземпляра класса `HttpRequest`, параметром `response` — ответ в виде экземпляра класса `TemplateResponse`.

  Метод должен возвращать ответ в виде экземпляра класса `TemplateResponse` — либо полученный с параметром `response` и измененный, либо новый, сгенерированный на основе полученного. Этот ответ и будет отправлен клиенту.

  Метод может заменить имя шаблона, занеся его в атрибут `template_name` ответа, или содержимое контекста шаблона, доступного из атрибута `context_data`.

Внимание!

Эффект от замены имени шаблона или изменения содержимого контекста в методе `process_template_response()` будет достигнут только в том случае, если ответ представлен экземпляром класса `TemplateResponse`.

В листинге 22.1 приведен код посредника `RubricsMiddleware`, который добавляет в контекст шаблона список рубрик, взятый из модели `Rubric`.

Листинг 22.1. Посредник, добавляющий в контекст шаблона дополнительные данные

```
from .models import Rubric

class RubricsMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        return self.get_response(request)

    def process_template_response(self, request, response):
        response.context_data['rubrics'] = Rubric.objects.all()
        return response
```

Не забудем зарегистрировать этот посредник в проекте (предполагается, что он сохранен в модуле `bboard.middlewares`):

```
MIDDLEWARE = [
    . . .
    'bboard.middlewares.RubricsMiddleware',
]
```

После этого мы можем удалить из контроллеров код, добавляющий в контекст шаблона список рубрик, разумеется, при условии, что ответ во всех этих контроллерах формируется в виде экземпляра класса `TemplateResponse`.

22.2. Обработчики контекста

Обработчик контекста — это программный модуль, добавляющий в контекст шаблона какие-либо дополнительные данные уже после формирования ответа контроллером.

Обработчики контекста удобно использовать, если нужно просто добавить в контекст шаблона какие-либо данные. Обработчики контекста реализуются проще, чем посредники, и работают в любом случае, независимо от того, представлен ответ экземпляром класса `TemplateResponse` или `HttpResponse`.

Обработчик контекста реализуется в виде обычной функции. Единственным параметром она должна принимать запрос в виде экземпляра класса `HttpRequest` и возвращать словарь с данными, которые нужно добавить в контекст шаблона.

В листинге 22.2 приведен код обработчика контекста `rubrics`, который добавляет в контекст шаблона список рубрик.

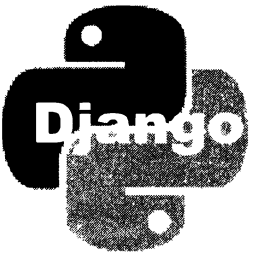
Листинг 22.2. Пример обработчика контекста

```
from .models import Rubric

def rubrics(request):
    return {'rubrics': Rubric.objects.all()}
```

Этот обработчик шаблона мы занесем в список параметра `context_processors`, который входит в состав дополнительных параметров используемого нами шаблонизатора:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        . . .
        'OPTIONS': {
            'context_processors': [
                . . .
                'bboard.middlewares.rubrics',
            ],
        . . .
        },
    },
]
```



ГЛАВА 23

Cookie, сессии, всплывающие сообщения и подписывание данных

Django поддерживает развитые средства для обработки cookie, хранения данных в сессиях, вывода всплывающих сообщений и защиты данных цифровой подписью.

23.1. Cookie

Cookie — небольшой, не более 4 Кбайт, фрагмент произвольных данных, сохраняемый на компьютере клиента. Обычно применяется для хранения служебных данных, настроек сайта и пр.

Все cookie, сохраненные на стороне клиента, относящиеся к текущему домену и еще не просроченные, доступны через атрибут `COOKIES` объекта запроса (экземпляра класса `HttpRequest`). Ключами элементов этого словаря выступают ключи всех доступных cookie, а значениями элементов — значения, сохраненные в этих cookie и представленные в виде строк. Значения cookie доступны только для чтения.

Вот пример извлечения из cookie текущего значения счетчика посещений страницы и увеличения его на единицу:

```
if 'counter' in request.COOKIES:
    cnt = int(request.COOKIES['counter']) + 1
else:
    cnt = 1
```

Для записи значения в cookie применяется метод `set_cookie()` класса `HttpResponse`, представляющего ответ. Вот формат вызова этого метода:

```
set_cookie(<ключ>[, value=''][, max_age=None][, expires=None][,
           path='/'][, domain=None][, secure=False][, httponly=False][,
           samesite=None])
```

Ключ записываемого значения указывается в виде строки. Само значение задается в параметре `value`; если он не указан, то будет записана пустая строка.

Параметр `max_age` указывает время хранения cookie на стороне клиента в секундах. Параметр `expires` задает дату и время, после которых cookie станет недействительным и будет удален, в виде объекта типа `datetime` из модуля `datetime` Python. В вызове метода следует указать один из этих параметров, но не оба сразу. Если ни один из этих параметров не указан, то cookie будет храниться лишь до тех пор, пока посетитель не уйдет с сайта.

Параметр `path` указывает путь, к которому будет относиться cookie, — в таком случае при запросе с другого пути сохраненное в cookie значение получить не удастся. Например, если задать значение `"/testapp/"`, cookie будет доступен только в контроллерах приложения `testapp`. Значение по умолчанию: `"/"` (путь к корневой папке), — в результате чего cookie станет доступным с любого пути.

Параметр `domain` указывает корневой домен, откуда должен быть доступен сохраняемый cookie, и применяется, если нужно создать cookie, доступный с другого домена. Так, если указать значение `"site.ru"`, то cookie будет доступен с доменов **www.site.ru**, **support.site.ru**, **shop.site.ru** и др. Если параметр не указан, cookie будет доступен только в текущем домене.

Если параметру `secure` дать значение `True`, то cookie будет доступен только при обращении к сайту по защищенному протоколу. Если параметр не указан (или если ему дано значение `False`), cookie будет доступен при обращении по любому протоколу.

Если параметру `httponly` дать значение `True`, cookie будет доступен только серверу. Если параметр не указан (или если ему дано значение `False`), cookie также будет доступен в клиентских веб-сценариях, написанных на JavaScript.

Параметр `samesite` разрешает или запрещает отправку сохраненного cookie при выполнении запросов на другие сайты. Доступны три значения:

- `None` — разрешает отправку cookie (поведение по умолчанию);
- `"Lax"` — разрешает отправку cookie только при переходе на другие сайты по гиперссылкам;
- `"Strict"` — полностью запрещает отправку cookie другим сайтам.

Вот пример записи в cookie значения счетчика посещений страницы, полученного ранее:

```
response = HttpResponse(. . .)
response.set_cookie('counter', cnt)
```

Удалить cookie можно вызовом метода `delete_cookie()` класса `HttpResponse`:

```
delete_cookie(<ключ>[, path='/'][, domain=None])
```

Значения параметров `path` и `domain` должны быть теми же, что использовались в вызове метода `set_cookie()`, создавшего удаляемый cookie. Если cookie с заданным ключом не найден, метод ничего не делает.

Django также поддерживает создание и чтение *подписанных* cookie, в которых сохраненное значение дополнительно защищено цифровой подписью.

Сохранение значения в подписанном cookie выполняется вызовом метода `set_signed_cookie()` класса `HttpResponse`:

```
set_signed_cookie(<ключ>[, value=''][, salt=''][, max_age=None][,
                  expires=None][, path='/'][, domain=None][,
                  secure=False][, httponly=False][, samesite=None])
```

Здесь указываются те же самые параметры, что и у метода `set_cookie()`. Дополнительный параметр `salt` задает *соль* — особое значение, участвующее в генерировании цифровой подписи и служащее для повышения ее стойкости.

Если в параметре `max_age` или `expires` задано время существования подписанного cookie, то сгенерированная цифровая подпись будет действительна в течение указанного времени.

Прочитать значение из подписанного cookie и удостовериться, что оно не скомпрометировано, позволяет метод `get_signed_cookie()` класса `HttpRequest`:

```
get_signed_cookie(<ключ>[, default=RAISE_ERROR][, salt=''][,
                                                 max_age=None])
```

Значение соли, заданное в параметре `salt`, должно быть тем же, что использовалось в вызове метода `set_signed_cookie()`, создавшего этот cookie.

Если цифровая подпись у сохраненного значения не была скомпрометирована, то метод вернет сохраненное значение. В противном случае будет возвращено значение, заданное в необязательном параметре `default`. То же самое случится, если cookie с заданным ключом не был найден.

Если в качестве значения параметра `default` указать значение переменной `RAISE_ERROR` из модуля `django.http.request`, то будет возбуждено одно из двух исключений: `BadSignature` из модуля `django.core.signing`, если цифровая подпись скомпрометирована, или `KeyError`, если cookie с заданным ключом не найден.

Если в необязательном параметре `max_age` указано время существования подписанного cookie, то дополнительно будет выполнена проверка, не устарела ли цифровая подпись. Если цифровая подпись устарела, метод возбудит исключение `SignatureExpired` из модуля `django.core.signing`.

Удалить подписанный cookie можно так же, как и cookie обычный, — вызовом метода `delete_cookie()` объекта ответа.

23.2. Сессии

Сессия — это промежуток времени, в течение которого посетитель пребывает на текущем сайте. Сессия начинается, как только посетитель заходит на сайт, и завершается после его ухода.

К сессии можно привязать произвольные данные и сохранить их в каком-либо хранилище (базе данных, файле и др.) на стороне. Эти данные будут храниться во время существования сессии и останутся в течение определенного времени после ее

завершения, пока не истечет указанный промежуток времени и сессия не перестанет быть актуальной. Такие данные тоже называют *сессией*.

Для каждой сессии Django генерирует уникальный идентификатор, который затем сохраняется в подписанном cookie на стороне клиента (*cookie сессии*). Поскольку содержимое всех cookie, сохраненных для того или иного домена, автоматически отсылается серверу в составе заголовка каждого запроса, Django впоследствии без проблем получит сохраненный на стороне клиента идентификатор и по нему найдет данные, записанные в соответствующей сессии.

Мы можем сохранить в сессии любые данные, какие нам нужны. В частности, подсистема разграничения доступа хранит в таких сессиях ключ пользователя, который выполнил вход на сайт.

Поскольку данные сессии сохраняются на стороне сервера, в них можно хранить конфиденциальные сведения, которые не должны быть доступны никому.

23.2.1. Настройка сессий

Чтобы успешно работать с сессиями, предварительно следует:

- проверить, присутствует ли приложение `django.contrib.sessions` в списке зарегистрированных в проекте (параметр `INSTALLED_APPS`);
- проверить, присутствует ли посредник `django.contrib.sessions.middleware.SessionMiddleware` в списке зарегистрированных в проекте (параметр `MIDDLEWARE`).

Впрочем, и приложение, и посредник присутствуют в списках изначально, поскольку активно используются другими стандартными приложениями Django.

Параметры, влияющие на работу подсистемы сессий, как обычно, указываются в настройках проекта — в модуле settings.py пакета конфигурации:

- `SESSION_ENGINE` — имя класса, реализующего хранилище для сессий, в виде строки. Можно указать следующие классы:
  - `django.contrib.sessions.backends.db` — хранит сессии в базе данных. Имеет среднюю производительность, но гарантирует максимальную надежность хранения данных;
  - `django.contrib.sessions.backends.file` — хранит сессии в обычных файлах. По сравнению с предыдущим хранилищем имеет пониженную производительность, но создает меньшую нагрузку на базу данных;
  - `django.contrib.sessions.backends.cache` — хранит сессии в кэше стороны сервера. Обеспечивает высокую производительность, но требует наличия активной подсистемы кэширования;
  - `django.contrib.sessions.backends.cached_db` — хранит сессии в кэше стороны сервера, одновременно дублируя их в базе данных для надежности. По сравнению с предыдущим хранилищем обеспечивает повышенную надежность, но увеличивает нагрузку на базу данных;

- `django.contrib.sessions.backends.signed_cookies` — хранит сессии непосредственно в cookie сессии. Обеспечивает максимальную производительность, но для каждой сессии позволяет сохранить не более 4 Кбайт данных.

Значение по умолчанию: `"django.contrib.sessions.backends.db"`;

- `SESSION_SERIALIZER` — имя класса сериализатора, который будет использоваться для сериализации сохраняемых в сессиях данных, указанное в виде строки. В составе Django поставляются два сериализатора:
  - `django.contrib.sessions.serializers.JSONSerializer` — сериализует данные в формат JSON. Может обрабатывать только элементарные типы Python;
  - `django.contrib.sessions.serializers.PickleSerializer` — сериализует средствами модуля `pickle`. Способен обработать значение любого типа.

  Значение по умолчанию: `"django.contrib.sessions.serializers.JSONSerializer"`;
- `SESSION_EXPIRE_AT_BROWSER_CLOSE` — если `True`, то сессии со всеми сохраненными в них данными будут автоматически удаляться, как только посетитель закроет веб-обозреватель, если `False`, то сессии будут сохраняться. По умолчанию — `False`;
- `SESSION_SAVE_EVERY_REQUEST` — если `True`, то сессии будут сохраняться в хранилище при обработке каждого запроса, если `False` — только при изменении записанных в них данных. По умолчанию — `False`;
- `SESSION_COOKIE_DOMAIN` — домен, к которому будут относиться cookie сессий. По умолчанию — `None` (т. е. текущий домен);
- `SESSION_COOKIE_PATH` — путь, к которому будут относиться cookie сессий (по умолчанию: `"/"`);
- `SESSION_COOKIE_AGE` — время существования cookie сессий, в виде целого числа в секундах. По умолчанию: `1209600` (2 недели);
- `SESSION_COOKIE_NAME` — ключ, под которым в cookie будет сохранен идентификатор сессии (по умолчанию: `"sessionid"`);
- `SESSION_COOKIE_HTTPONLY` — если `True`, то cookie сессий будут доступны только серверу. Если `False`, то cookie сессий также будут доступны клиентским вебсценариям. По умолчанию — `True`;
- `SESSION_COOKIE_SECURE` — если `True`, то cookie сессий будут доступны только при обращении к сайту по защищенному протоколу HTTPS, если `False` — при обращении по любому протоколу. По умолчанию — `False`;
- `SESSION_COOKIE_SAMESITE` — признак, разрешающий или запрещающий отправку cookie сессий при переходе на другие сайты. Доступны три значения:
  - `None` — разрешает отправку cookie сессий (поведение по умолчанию);
  - `"Lax"` — разрешает отправку cookie сессий только при переходе на другие сайты по гиперссылкам;
  - `"Strict"` — полностью запрещает отправку cookie сессий другим сайтам;

- `SESSION_FILE_PATH` — полный путь к папке, в которой будут храниться файлы с сессиями. Если указано значение `None`, то Django использует системную папку для хранения временных файлов. По умолчанию — `None`.

  Этот параметр принимается во внимание только в том случае, если для хранения сессий были выбраны обычные файлы;

- `SESSION_CACHE_ALIAS` — название кэша, в котором будут храниться сессии. По умолчанию: `"default"` (кэш по умолчанию).

  Этот параметр принимается во внимание только в том случае, если для хранения сессий был выбран кэш стороны сервера без дублирования в базе данных или же с таковым.

Если в качестве хранилища сессий были выбраны база данных или кэш стороны сервера с дублированием в базе данных, то перед использованием сессий следует выполнить миграции.

> ***На заметку***
>
> Если для хранения сессий были выбраны база данных или кэш стороны сервера с дублированием в базе данных, то в базе данных будет создана таблица `django_session`.

23.2.2. Использование сессий

Посредник `django.contrib.sessions.middleware.SessionMiddleware` добавляет объекту запроса атрибут `sessions`. Он хранит объект, поддерживающий функциональность словаря и содержащий все значения, которые были сохранены в текущей сессии.

Вот пример реализации счетчика посещений страницы, аналогичного представленному в *разд. 23.1*, но хранящего текущее значение в сессии:

```
if 'counter' in request.session:
    cnt = request.session['counter'] + 1
else:
    cnt = 1
. . .
request.session['counter'] = cnt
```

Помимо этого, объект, хранящийся в атрибуте `sessions` объекта запроса, поддерживает следующие методы:

- `flush()` — удаляет все данные, сохраненные в текущей сессии, наряду с cookie сессии (метод `clear()`, поддерживаемый тем же объектом, равно как и словарями Python, не удаляет cookie);
- `set_test_cookie()` — создает тестовый cookie, позволяющий удостовериться, что веб-обозреватель клиента поддерживает cookie;
- `test_cookie_worked()` — возвращает `True`, если веб-обозреватель клиента поддерживает cookie, и `False` — в противном случае. Проверка, поддерживает ли веб-обозреватель cookie, запускается вызовом метода `set_test_cookie()`;
- `delete_test_cookie()` — удаляет созданный ранее тестовый cookie.

Вот пример использования трех из описанных выше методов:

```
def test_cookie(request):
    if request.method == 'POST':
        if request.session.test_cookie_worked():
            request.session.delete_test_cookie()
            # Веб-обозреватель поддерживает cookie
        else:
            # Веб-обозреватель не поддерживает cookie
    request.session.set_test_cookie()
    return render(request, 'testapp/test_cookie.html')
```

- `set_expiry(<время>)` — задает время устаревания текущей сессии, по достижении которого сессия будет удалена. В качестве значения *времени* можно указать:
  - целое число — задаст количество секунд, в течение которых сессия будет актуальна;
  - объект типа `datetime` или `timedelta` из модуля `datetime` — укажет временну́ю отметку устаревания сессии. Поддерживается только при использовании сериализатора `django.contrib.sessions.serializers.JSONSerializer`;
  - `0` — сессия перестанет быть актуальной и будет удалена, как только посетитель закроет веб-обозреватель;
  - `None` — будет использовано значение из параметра `SESSION_COOKIE_AGE` настроек проекта;
- `get_expiry_age([modification=datetime.datetime.today()][,][expiry=None])` — возвращает время, в течение которого текущая сессия еще будет актуальной, в секундах. Необязательный параметр `modification` указывает временну́ю отметку последнего изменения сессии (по умолчанию — текущие дата и время), а параметр `expiry` — время ее устаревания в виде временно́й отметки, количества секунд или `None` (в этом случае будет использовано время устаревания, заданное вызовом метода `set_expiry()` или, если этот метод не вызывался, взятое из параметра `SESSION_COOKIE_AGE`);
- `get_expiry_date([modification=datetime.datetime.today()][,][expiry=None])` — возвращает временну́ю отметку устаревания текущей сессии. Необязательный параметр `modification` указывает временну́ю отметку последнего изменения сессии (по умолчанию — текущие дата и время), а параметр `expiry` — время ее устаревания в виде временно́й отметки, количества секунд или `None` (в этом случае будет использовано время устаревания, заданное вызовом метода `set_expiry()` или, если этот метод не вызывался, взятое из параметра `SESSION_COOKIE_AGE`);
- `get_expire_at_browser_close()` — возвращает `True`, если текущая сессия устареет и будет удалена, как только посетитель закроет веб-обозреватель, и `False` — в противном случае;
- `clear_expired()` — удаляет устаревшие сессии;

- `cycle_key()` — создает новый идентификатор для текущей сессии без потери сохраненных в ней данных.

23.2.3. Дополнительная команда *clearsessions*

Для удаления всех устаревших сессий, которые по какой-то причине не были удалены автоматически, достаточно применить команду `clearsessions` утилиты manage.py. Формат ее вызова очень прост:

```
manage.py clearsessions
```

23.3. Всплывающие сообщения

Всплывающие сообщения существуют только во время выполнения текущего запроса. Они применяются для вывода на страницу какого-либо сообщения (например, об успешном добавлении новой записи), актуального только в данный момент.

23.3.1. Настройка всплывающих сообщений

Перед использованием подсистемы всплывающих сообщений необходимо:

- проверить, присутствует ли приложение `django.contrib.messages` в списке зарегистрированных в проекте (параметр `INSTALLED_APPS`);
- проверить, присутствуют ли посредники `django.contrib.sessions.middleware.SessionMiddleware` и `django.contrib.messages.middleware.MessageMiddleware` в списке зарегистрированных в проекте (параметр `MIDDLEWARE`);
- проверить, присутствует ли обработчик контекста `django.contrib.messages.context_processors.messages` в списке зарегистрированных для используемого шаблонизатора.

Впрочем, и приложение, и посредники, и обработчик контекста присутствуют в списках изначально.

Немногочисленные настройки, управляющие работой подсистемы всплывающих сообщений, записываются в модуле settings.py пакета конфигурации:

- `MESSAGE_STORAGE` — имя класса, реализующего хранилище всплывающих сообщений, представленное в виде строки. В составе Django поставляются три хранилища всплывающих сообщений:
  - `django.contrib.messages.storage.cookie.CookieStorage` — использует cookie;
  - `django.contrib.messages.storage.session.SessionStorage` — использует сессии;
  - `django.contrib.messages.storage.fallback.FallbackStorage` — использует cookie для хранения сообщений, чей объем не превышает 4 Кбайт, а более объемные сохраняет в сессии.

  Значение по умолчанию:
  `"django.contrib.messages.storage.fallback.FallbackStorage"`;

- MESSAGE_LEVEL — минимальный уровень всплывающих сообщений, которые будут выводиться подсистемой. Указывается в виде целого числа. По умолчанию: 20;
- MESSAGE_TAGS — соответствия между уровнями сообщений и стилевыми классами. Более подробно это будет рассмотрено позже.

23.3.2. Уровни всплывающих сообщений

Каждое всплывающее сообщение Django, помимо текстового содержимого, имеет так называемый *уровень*, указывающий его ранг и выражаемый целым числом. Каждому такому уровню соответствует свой стилевой класс, привязываемый к HTML-тегу, в котором выводится текст сообщения.

Изначально в Django объявлено пять уровней всплывающих сообщений, каждый из которых имеет строго определенную область применения. Значение каждого из этих уровней занесено в отдельную переменную, и эти переменные, объявленные в модуле `django.contrib.messages`, можно использовать для указания уровней сообщений при их выводе. Все уровни, вместе с хранящими их переменными и соответствующими им стилевыми классами, приведены в табл. 23.1.

***Таблица 23.1.** Уровни всплывающих сообщений, объявленные в Django*

Переменная	Значение	Описание	Стилевой класс
DEBUG	10	Отладочные сообщения, предназначенные только для разработчиков	debug
INFO	20	Информационные сообщения для посетителей	info
SUCCESS	25	Сообщения об успешном выполнении каких-либо действий	success
WARNING	30	Сообщения о возникновении каких-либо нештатных ситуаций, которые могут привести к сбою	warning
ERROR	40	Сообщения о неуспешном выполнении каких-либо действий	error

Параметр MESSAGE_LEVEL настроек проекта указывает минимальный уровень сообщений, которые будут выводиться на страницы. Если уровень создаваемого сообщения меньше указанной в нем величины, то сообщение не будет выведено. По умолчанию этот параметр имеет значение 20 (переменная INFO), следовательно, сообщения с меньшим уровнем, в частности отладочные (DEBUG), обрабатываться не будут. Если нужно сделать так, чтобы отладочные сообщения также выводились на экран, необходимо задать для этого параметра подходящее значение:

```
from django.contrib import messages
MESSAGE_LEVEL = messages.DEBUG
```

23.3.3. Создание всплывающих сообщений

Создать всплывающее сообщение для его последующего вывода можно вызовом описанных далее функций из модуля `django.contrib.messages`.

В первую очередь это "универсальная" функция `add_message()`, создающая сообщение произвольного уровня. Вот формат ее вызова:

```
add_message(<запрос>, <уровень сообщения>, <текст сообщений>[,
                    extra_tags=''][, fail_silently=False])
```

Запрос представляется экземпляром класса `HttpRequest`, *уровень сообщения* — целым числом, а его *текст* — строкой.

Необязательный параметр `extra_tags` указывает перечень дополнительных стилевых классов, привязываемых к HTML-тегу, в котором будет выведен текст сообщения. Перечень стилевых классов должен представлять собой строку, а стилевые классы в нем должны отделяться друг от друга пробелами.

Если задать необязательному параметру `fail_silently` значение `True`, то в случае невозможности создания нового всплывающего сообщения (например, если соответствующая подсистема отключена) ничего не произойдет. Используемое по умолчанию значение `False` указывает в таком случае возбудить исключение `MessageFailure` из того же модуля `django.contrib.messages`.

Пример создания нового всплывающего сообщения:

```
from django.contrib import messages
. . .
def edit(request, pk):
    . . .
    messages.add_message(request, messages.SUCCESS,
                                  'Объявление исправлено')
    . . .
```

Пример создания всплывающего сообщения с добавлением дополнительных стилевых классов `first` и `second`:

```
messages.add_message(request, messages.SUCCESS, 'Объявление исправлено',
                     extra_tags='first second')
```

Функции `debug()`, `info()`, `success()`, `warning()` и `error()` создают сообщение соответствующего уровня. Все они имеют одинаковый формат вызова:

```
debug|info|success|warning|error(<запрос>, <текст сообщений>[,
                                extra_tags=''][, fail_silently=False])
```

Значения параметров здесь задаются в том же формате, что и у функции `add_message()`.

Пример:

```
messages.success(request, 'Объявление исправлено')
```

"Научить" создавать всплывающие сообщения высокоуровневые контроллеры-классы можно, унаследовав их от класса-примеси `SuccessMessageMixin` из модуля

`django.contrib.messages.views`. Этот класс поддерживает следующие атрибут и метод:

- `success_message` — текст сообщения об успешном выполнении операции в виде строки. В строке допускается применять специальные символы вида `%(<имя поля формы>)s`, вместо которых будут подставлены значения соответствующих *полей*;
- `get_success_message(self, cleaded_data)` — должен возвращать полностью сформированный текст всплывающего сообщения об успешном выполнении операции. Словарь с данными формы из параметра `cleaned_data` содержит полностью готовые к использованию значения полей формы.

  В изначальной реализации возвращает результат форматирования строки из атрибута `success_message` с применением полученного словаря с данными формы.

В листинге 23.1 приведен код контроллера, создающего новое объявление, который в случае успешного его создания отправляет посетителю всплывающее сообщение.

Листинг 23.1. Использование примеси `SuccessMessageMixin`

```
from django.views.generic.edit import CreateView
from django.contrib.messages.views import SuccessMessageMixin
from .models import Bb
from .forms import BbForm

class BbCreateView(SuccessMessageMixin, CreateView):
    template_name = 'bboard/create.html'
    form_class = BbForm
    success_url = '/{rubric_id}'
    success_message = 'Объявление о продаже товара "%(title)s" создано.'
```

23.3.4. Вывод всплывающих сообщений

Вывести всплывающие сообщения в шаблоне удобнее всего посредством обработчика контекста `django.contrib.messages.context_processors.messages`. Он добавляет в контекст шаблона переменную `messages`, которая хранит последовательность всех всплывающих сообщений, созданных к настоящему времени в текущем запросе.

Каждый элемент этой последовательности представляет собой экземпляр класса `Message`. Все необходимые сведения о сообщении хранятся в его атрибутах:

- `message` — текст всплывающего сообщения;
- `level` — уровень всплывающего сообщения в виде целого числа;
- `level_tag` — имя основного стилевого класса, соответствующего уровню сообщения;
- `extra_tags` — строка с дополнительными стилевыми классами, указанными в параметре `extra_tags` при создании сообщения (см. *разд. 23.3.3*);

- `tags` — строка со всеми стилевыми классами (и основным, и дополнительными), записанными через пробелы.

Вот пример кода шаблона, выполняющего вывод всплывающих сообщений:

```
{% if messages %}
<ul class="messages">
    {% for message in messages %}
    <li{% if message.tags %} class="{{ message.tags }}"{% endif %}>
        {{ message }}
    </li>
    {% endfor %}
</ul>
{% endif %}
```

Еще обработчик контекста `django.contrib.messages.context_processors.messages` добавляет в контекст шаблона переменную `DEFAULT_MESSAGE_LEVELS`. Она хранит словарь, в качестве ключей элементов которого выступают строковые названия уровней сообщений, а значений элементов — соответствующие им числа. Этот словарь можно использовать в операциях сравнения, подобных этой:

```
<li{% if message.tags %} class="{{ message.tags }}"{% endif %}>
    {% if message.level == DEFAULT_MESSAGE_LEVELS.ERROR %}
    Внимание!
    {% endif %}
    {{ message }}
</li>
```

Если же нужно получить доступ к сообщениям в контроллере, то можно воспользоваться функцией `get_messages(<запрос>)` из модуля `django.contrib.messages`. *Запрос* представляется экземпляром класса `HttpRequest`, а результатом будет список сообщений в виде экземпляров класса `Message`. Пример:

```
from django.contrib import messages
. . .
def edit(request, pk):
    . . .
    messages = messages.get_messages(request)
    first_message_text = messages[0].message
    . . .
```

23.3.5. Объявление своих уровней всплывающих сообщений

Никто и ничто не мешает нам при создании всплывающего сообщения указать произвольный уровень:

```
CRITICAL = 50
messages.add_message(request, CRITICAL, 'Случилось что-то очень нехорошее...')
```

Нужно только проследить за тем, чтобы выбранное значение уровня не совпало с каким-либо из объявленных в самом Django (см. *разд. 23.3.2*).

Если мы хотим, чтобы при выводе всплывающих сообщений на экран для объявленного нами уровня устанавливался какой-либо стилевой класс, то должны выполнить дополнительные действия. Мы объявим словарь, добавим в него элемент, соответствующий объявленному нами уровню сообщений, установим в качестве ключа элемента значение уровня, а в качестве значения элемента — строку с именем стилевого класса, после чего присвоим этот словарь параметру `MESSAGE_TAGS` настроек проекта. Вот пример:

```
MESSAGE_TAGS = {
    CRITICAL: 'critical',
}
```

23.4. Подписывание данных

Для подписывания строковых значений обычной цифровой подписью применяется класс `Signer` из модуля `django.core.signing`. Конструктор этого класса вызывается в формате:

```
Signer([key=None][,][sep=':'][,][salt=None])
```

Параметр `key` указывает секретный ключ, на основе которого будет генерироваться цифровая подпись (по умолчанию используется секретный ключ из параметра `SECRET_KEY` настроек проекта). Параметр `sep` задает символ, которым будут отделяться друг от друга подписанное значение и сама подпись (по умолчанию — двоеточие). Наконец, параметр `salt` указывает соль (если он опущен, соль задействована не будет).

Класс `Signer` поддерживает два метода:

- `sign(<подписываемое значение>)` — подписывает полученное *значение* и возвращает результат:

```
>>> from django.core.signing import Signer
>>> signer = Signer()
>>> val = signer.sign('Django')
>>> val
'Django:PklE2-T6egEtTnNCA_jh3LXOtD8'
```

- `unsign(<подписанное значение>)` — из полученного *подписанного значения* извлекает оригинальную величину, которую и возвращает в качестве результата:

```
>>> signer.unsign(val)
'Django'
```

  Если подписанное значение скомпрометировано (не соответствует цифровой подписи), то возбуждается исключение `BadSignature` из модуля `django.core.signing`:

```
>>> signer.unsign(val + '3')
Traceback (most recent call last):
. . .
django.core.signing.BadSignature:
Signature "PklE2-T6egEtTnNCA_jh3LXOtD83" does not match
```

Класс `TimestampSigner` из того же модуля `django.core.signing` подписывает значение цифровой подписью с ограниченным сроком действия. Формат вызова его конструктора такой же, как у конструктора класса `Signer`.

Класс `TimestampSigner` поддерживает два метода:

- `sign(<подписываемое значение>)` — подписывает полученное *значение* и возвращает результат:

```
>>> from django.core.signing import TimestampSigner
>>> signer = TimestampSigner()
>>> val = signer.sign('Python')
>>> val
'Python:1irdN8:G9z3m4d4CQZkLKANMuUSi_M1so4'
```

- `unsign(<подписанное значение>[, max_age=None])` — из полученного *подписанного значения* извлекает оригинальную величину, которую и возвращает в качестве результата. Параметр `max_age` задает промежуток времени, в течение которого актуальна цифровая подпись, в виде целого числа, в секундах, или в виде объекта типа `timedelta` из модуля `datetime`. Если подписанное значение скомпрометировано, то возбуждается исключение `BadSignature` из модуля `django.core.signing`. Примеры:

```
>>> signer.unsign(val, max_age=3600)
'Python'
>>> from datetime import timedelta
>>> signer.unsign(val, max_age=timedelta(minutes=30))
'Python'
```

Если цифровая подпись уже не актуальна, возбуждается исключение `SignatureExpired` из модуля `django.core.signing`:

```
>>> signer.unsign(val, max_age=timedelta(seconds=30))
Traceback (most recent call last):
. . .
django.core.signing.SignatureExpired: Signature age
89.87273287773132 > 30.0 seconds
```

Если параметр `max_age` не указан, то проверка на актуальность цифровой подписи не проводится, и метод `unsign()` работает так же, как его "тезка" у класса `Signer`:

```
>>> signer.unsign(val)
'Python'
```

Если нужно подписать значение, отличающееся от строки, то следует воспользоваться двумя функциями из модуля `django.core.signing`:

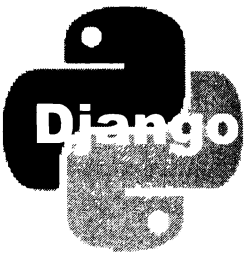
- `dumps(<значение>[, key=None][, salt='django.core.signing'][, compress=False])` — подписывает указанное *значение* с применением класса `TimestampSigner` и возвращает результат в виде строки. Параметр `key` задает секретный ключ (по умолчанию — значение из параметра `SECRET_KEY` настроек проекта), а параметр `salt` — соль (по умолчанию — строка `"django.core.signing"`). Если параметру `compress` передать значение `True`, то результат будет сформирован в сжатом виде (значение по умолчанию — `False`). Сжатие будет давать более заметный результат при подписывании данных большого объема. Примеры:

```
>>> from django.core.signing import dumps
>>> s1 = dumps(123456789)
>>> s1
'MTIzNDU2Nzg5:1irdQB:NyV0AIq2QJ4GHjNlb_yv3i8CNQc'
>>> s2 = dumps([1, 2, 3, 4])
>>> s2
'WzEsMiwzLDRd:1irdQW:JjXHCbjZ7AOGUOIb6rU0rMhoDA4'
>>> s3 = dumps([1, 2, 3, 4], compress=True)
>>> s3
'WzEsMiwzLDRd:1irdQo:TqAtb1rJRPWvHmSUoFjogP-2EdU'
```

- `loads(<подписанное значение>[, key=None][, salt='django.core.signing'][, max_age=None])` — из полученного *подписанного значения* извлекает оригинальную величину и возвращает в качестве результата. Параметры `key` и `salt` указывают соответственно секретный ключ и соль — эти значения должны быть теми же, что использовались при подписывании значения вызовом функции `dumps()`. Параметр `max_age` задает промежуток времени, в течение которого цифровая подпись актуальна. Если он опущен, то проверка на актуальность подписи не проводится. Примеры:

```
>>> from django.core.signing import loads
>>> loads(s1)
123456789
>>> loads(s2)
[1, 2, 3, 4]
>>> loads(s3)
[1, 2, 3, 4]
>>> loads(s3, max_age=10)
Traceback (most recent call last):
. . .
django.core.signing.SignatureExpired: Signature age
81.08918499946594 > 10 seconds
```

Если цифровая подпись скомпрометирована или потеряла актуальность, то будут возбуждены исключения `BadSignature` или `SignatureExpired` соответственно.



ГЛАВА 24

Сигналы

Сигнал сообщает о выполнении Django какого-либо действия: создании новой записи в модели, удалении записи, входе пользователя на сайт, выходе с него и пр. К сигналу можно привязать *обработчик* — функцию или метод, который будет вызываться при возникновении сигнала.

Сигналы предоставляют возможность вклиниться в процесс работы самого фреймворка или отдельных приложений — неважно, стандартных или написанных самим разработчиком сайта — и произвести какие-либо дополнительные действия. Скажем, приложение `django-cleanup`, рассмотренное нами в *разд. 20.5* и удаляющее ненужные файлы, чтобы отследить момент правки или удаления записи, обрабатывает сигналы `post_init`, `pre_save`, `post_save` и `post_delete`.

24.1. Обработка сигналов

Все сигналы в Django представляются экземплярами класса `Signal` или его подклассов. Этот класс поддерживает два метода, предназначенные для привязки к сигналу обработчика или отмены его привязки.

Для привязки обработчика к сигналу применяется метод `connect()` класса `Signal`:

```
connect(<обработчик>[, sender=None][, weak=True][, dispatch_uid=None])
```

Обработчик сигнала, как было сказано ранее, должен представлять собой функцию или метод. Формат написания этой функции (метода) мы рассмотрим позднее.

В необязательном параметре `sender` можно указать класс объекта, из которого отправляется текущий сигнал (*отправителя*). После чего *обработчик* будет обрабатывать сигналы исключительно от отправителей, относящихся к этому классу.

Если необязательному параметру `weak` присвоено значение `True` (а это его значение по умолчанию), то для связи отправителя и обработчика будет использована слабая ссылка Python, если присвоено значение `False` — обычная. Давать этому параметру значение `False` следует в случае, если после удаления всех отправителей нужда в обработчике пропадает, — тогда он будет автоматически выгружен из памяти.

Необязательный параметр `dispatch_uid` указывается, если к одному и тому же сигналу несколько раз привязывается один и тот же обработчик, и возникает необходимость как-то отличить одну такую привязку от другой. В этом случае в разных вызовах метода `connect()` нужно указать разные значения этого параметра, которые должны представлять собой строки.

Если к одному и тому же сигналу привязано несколько обработчиков, то они будут выполняться один за другим в той последовательности, в которой были привязаны к сигналу.

Рассмотрим несколько примеров привязки обработчика к сигналу `post_save`, возникающему после сохранения записи модели:

```
from django.db.models.signals import post_save
# Простая привязка обработчика post_save_dispatcher() к сигналу
post_save.connect(post_save_dispatcher)
# Простая привязка обработчика к сигналу, возникающему в модели Bb
post_save.connect(post_save_dispatcher, sender=Bb)
# Двукратная привязка обработчиков к сигналу с указанием разных значений
# параметра dispatch_uid
post_save.connect(post_save_dispatcher,
                  dispatch_uid='post_save_dispatcher_1')
post_save.connect(post_save_dispatcher,
                  dispatch_uid='post_save_dispatcher_2')
```

Обработчик — функция или метод — должен принимать один позиционный параметр, с которым передается класс объекта-отправителя сигнала. Помимо этого, обработчик может принимать произвольное число именованных параметров, набор которых у каждого сигнала различается (стандартные сигналы Django и передаваемые ими параметры мы рассмотрим позже). Вот своего рода шаблоны для написания обработчиков разных типов:

```
def post_save_dispatcher(sender, **kwargs):
    # Тело функции-обработчика
    # Получаем класс объекта-отправителя сигнала
    snd = sender
    # Получаем значение переданного обработчику именованного параметра
    # instance
    instance = kwargs['instance']
    . . .

class SomeClass:
    def post_save_dispatcher(self, sender, **kwargs):
        # Тело метода-обработчика
        . . .
```

Вместо метода `connect()` объекта сигнала можно использовать декоратор `receiver(<сигнал>)`, объявленный в модуле `django.dispatch`:

```
from django.dispatch import receiver
@receiver(post_save)
def post_save_dispatcher(sender, **kwargs):
    . . .
```

Код, выполняющий привязку к сигналам обработчиков, которые должны действовать все время, пока работает сайт, обычно записывается в модуле apps.py или models.py.

Отменить привязку обработчика к сигналу позволяет метод `disconnect()` класса `Signal`:

```
disconnect([receiver=None][,][sender=None][,][dispatch_uid=None])
```

В параметре `receiver` указывается обработчик, ранее привязанный к сигналу. Если этот обработчик был привязан к сигналам, отправляемым конкретным классом, то последний следует указать в параметре `sender`. Если в вызове метода `connect()`, выполнившем привязку обработчика, был задан параметр `dispatch_uid` с каким-либо значением, то удалить привязку можно, записав в вызове метода `disconnect()` только параметр `dispatch_uid` и указав в нем то же значение. Примеры:

```
post_save.disconnect(receiver=post_save_dispatcher)
post_save.disconnect(receiver=post_save_dispatcher, sender=Bb)
post_save.disconnect(dispatch_uid='post_save_dispatcher_2')
```

24.2. Встроенные сигналы Django

Сигналы, отправляемые подсистемой доступа к базам данных и объявленные в модуле `django.db.models.signals`:

- `pre_init` — отправляется в самом начале создания новой записи модели, перед выполнением конструктора ее класса. Обработчику передаются следующие параметры:
  - `sender` — класс модели, запись которой создается;
  - `args` — список позиционных аргументов, переданных конструктору модели;
  - `kwargs` — словарь именованных аргументов, переданных конструктору модели.

  Например, при создании нового объявления выполнением выражения:

  ```
  Bb.objects.create(title='Дом', content='Трехэтажный, кирпич', price=50000000)
  ```

  обработчик с параметром `sender` получит ссылку на класс модели `Bb`, с параметром `args` — "пустой" список, а с параметром `kwargs` — словарь `{'title': 'Дом', 'content': 'Трехэтажный, кирпич', 'price': 50000000}`;
- `post_init` — отправляется в конце создания новой записи модели, после выполнения конструктора ее класса. Обработчику передаются следующие параметры:
  - `sender` — класс модели, запись которой была создана;
  - `instance` — объект созданной записи;

- pre_save — отправляется перед сохранением записи модели, до вызова ее метода save(). Обработчику передаются параметры:
  - sender — класс модели, запись которой сохраняется;
  - instance — объект сохраняемой записи;
  - raw — True, если запись будет сохранена как есть, без обращения к другим записям за дополнительными данными и без исправления других записей, и False — в противном случае;
  - update_fields — множество имен полей, заданных в параметре update_fields метода save(), или None, если этот параметр не был указан;
- post_save — отправляется после сохранения записи модели, после вызова ее метода save(). Обработчику передаются такие параметры:
  - sender — класс модели, запись которой была сохранена;
  - instance — объект сохраненной записи;
  - created — True, если это вновь созданная запись, и False — в противном случае;
  - raw — True, если запись была сохранена как есть, без обращения к другим записям за дополнительными данными и без исправления других записей, и False — в противном случае;
  - update_fields — множество имен полей, заданных в параметре update_fields метода save(), или None, если этот параметр не был указан.

  Вероятно, это один из наиболее часто обрабатываемых сигналов. Вот пример его обработки с целью вывести в консоли Django сообщение о добавлении объявления:

```
from django.db.models.signals import post_save

def post_save_dispatcher(sender, **kwargs):
    if kwargs['created']:
        print('Объявление в рубрике "%s" создано' % \
              kwargs['instance'].rubric.name)

post_save.connect(post_save_dispatcher, sender=Bb)
```

- pre_delete — отправляется перед удалением записи, до вызова ее метода delete(). Параметры, передаваемые обработчику:
  - sender — класс модели, запись которой удаляется;
  - instance — объект удаляемой записи;
- post_delete — отправляется после удаления записи, после вызова ее метода delete(). Обработчик получит следующие параметры:

- sender — класс модели, запись которой была удалена;
- instance — объект удаленной записи. Отметим, что эта запись более не существует в базе данных;

❒ m2m_changed — отправляется связующей моделью при изменении состава записей моделей, связанных посредством связи "многие-со-многими" (см. *разд. 4.3.3*).

Связующая модель может быть явно задана в параметре through конструктора класса ManyToManyField или же создана фреймворком неявно. В любом случае связующую модель можно получить из атрибута through объекта поля типа ManyToManyField. Пример привязки обработчика к сигналу, отправляемому связующей моделью, которая была неявно создана при установлении связи "многие-со-многими" между моделями Machine и Spare (см. листинг 4.2):

```
m2m_changed.connect(m2m_dispatcher, sender=Machine.spares.through)
```

Обработчик этого сигнала принимает параметры:

- sender — класс связующей модели;
- instance — объект записи, в котором выполняются манипуляции по изменению состава связанных записей (т. е. у которого вызываются методы add(), create(), set() и др., описанные в *разд. 6.6.3*);
- action — строковое обозначение выполняемого действия:
  - "pre_add" — начало добавления новой записи в состав связанных;
  - "post_add" — окончание добавления новой связанной записи в состав связываемых;
  - "pre_remove" — начало удаления записи из состава связанных;
  - "post_remove" — окончание удаления записи из состава связанных;
  - "pre_clear" — начало удаления всех записей из состава связанных;
  - "post_clear" — окончание удаления всех записей из состава связанных;
- reverse — False, если изменение состава связанных записей выполняется в записи ведущей модели, и True, если в записи ведомой модели;
- model — класс модели, к которой принадлежит запись, добавляемая в состав связанных или удаляемая оттуда;
- pk_set — множество ключей записей, добавляемых в состав связанных или удаляемых оттуда. Для действий "pre_clear" и "post_clear" всегда None.

Например, при выполнении операций:

```
m = Machine.objects.create(name='Самосвал')
s = Spare.objects.create(name='Болт')
m.spares.add(s)
```

обработчик сигнала m2m_changed с параметром sender получит ссылку на класс промежуточной модели (у нас — созданной самим фреймворком), с параметром

instance — запись m (т. к. действия по изменению состава связанных записей выполняются в ней), с параметром action — строку "pre_add", с параметром reverse — False (действия по изменению состава связанных записей выполняются в записи ведущей модели), с параметром model — модель Spare, а с параметром pk_set — множество из единственного элемента — ключа записи s. Впоследствии тот же самый сигнал будет отправлен еще раз, и его обработчик получит с параметрами те же данные, за исключением параметра action, который будет иметь значение "post_add".

А после выполнения действия:

```
s.machine_set.remove(m)
```

обработчик сигнала m2m_changed с параметром sender получит ссылку на класс промежуточной модели, с параметром instance — запись s, с параметром action — строку "pre_remove", с параметром reverse — True (поскольку теперь действия по изменению состава связанных записей выполняются в записи ведомой модели), с параметром model — модель Machine, а с параметром pk_set — множество из единственного элемента — ключа записи m. Далее тот же самый сигнал будет отправлен еще раз, и его обработчик получит с параметрами те же данные, за исключением параметра action, который будет иметь значение "post_remove".

Сигналы, отправляемые подсистемой обработки запросов и объявленные в модуле django.core.signals:

- request_started — отправляется в самом начале обработки запроса. Обработчик получит параметры:
  - sender — класс django.core.handlers.wsgi.WsgiHandler, обрабатывающий все полученные запросы;
  - environ — словарь, содержащий переменные окружения;
- request_finished — отправляется после пересылки ответа клиенту. Обработчик с параметром sender получит класс django.core.handlers.wsgi.WsgiHandler, обрабатывающий все полученные запросы;
- got_request_exception — отправляется при возбуждении исключения в процессе обработки запроса. Вот параметры, передаваемые обработчику:
  - sender — None;
  - request — сам запрос в виде экземпляра класса HttpRequest.

Сигналы, отправляемые подсистемой разграничения доступа и объявленные в модуле django.contrib.auth.signals:

- user_logged_in — отправляется после удачного входа на сайт. Параметры, передаваемые обработчику:
  - sender — класс модели пользователя (User, если не была задана другая модель);

- `request` — текущий запрос, представленный экземпляром класса `HttpRequest`;
- `user` — запись пользователя, который вошел на сайт;

❐ `user_logged_out` — отправляется после удачного выхода с сайта. Вот параметры, которые получит обработчик:

- `sender` — класс модели пользователя или `None`, если пользователь ранее не выполнил вход на сайт;
- `request` — текущий запрос в виде экземпляра класса `HttpRequest`;
- `user` — запись пользователя, который вышел с сайта, или `None`, если пользователь ранее не выполнил вход на сайт;

❐ `user_login_failed` — отправляется, если посетитель не смог войти на сайт. Параметры, передаваемые обработчику:

- `sender` — строка с именем модуля, выполнявшего аутентификацию;
- `credentials` — словарь со сведениями, занесенными посетителем в форму входа и переданными впоследствии функции `authenticate()`. Вместо пароля будет подставлена последовательность звездочек;
- `request` — текущий запрос в виде экземпляра класса `HttpRequest`, если таковой был передан функции `authenticate()`, в противном случае — `None`.

> ***На заметку***
>
> Некоторые специфические сигналы, используемые внутренними механизмами Django или подсистемами, не рассматриваемыми в этой книге, здесь не рассмотрены. Их описание можно найти на странице **https://docs.djangoproject.com/en/3.0/ref/signals/**.

24.3. Объявление своих сигналов

Сначала нужно объявить сигнал, создав экземпляр класса `Signal` из модуля `django.dispatch`. Конструктор этого класса вызывается согласно формату:

```
Signal(providing_args=<список имен параметров, передаваемых обработчику>)
```

Имена параметров в передаваемом *списке* должны быть представлены в виде строк.

Пример объявления сигнала `add_bb`, который будет передавать обработчику параметры `instance` и `rubric`:

```
from django.dispatch import Signal
add_bb = Signal(providing_args=['instance', 'rubric'])
```

Для отправки объявленного сигнала применяются два следующих метода класса `Signal`:

❐ `send(<отправитель>[, <именованные параметры, указанные при объявлении сигнала>])` — выполняет отправку текущего сигнала от имени указанного *отправителя*, возможно, с *именованными параметрами*, которые были указаны при объявлении сигнала и будут отправлены его обработчику.

В качестве результата метод возвращает список, каждый из элементов которого представляет один из привязанных к текущему сигналу обработчиков. Каждый элемент этого списка представляет собой кортеж из двух элементов: ссылки на обработчик и возвращенный им результат. Если обработчик не возвращает результата, то вторым элементом станет значение `None`.

Пример:

```
add_bb.send(Bb, instance=bb, rubric=bb.rubric)
```

Если к сигналу привязано несколько обработчиков и в одном из них было возбуждено исключение, последующие обработчики выполнены не будут;

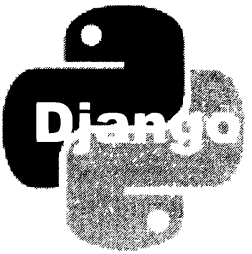
- `send_robust(<отправитель>[, <именованные параметры, указанные при объявлении сигнала>])` — то же самое, что `send()`, но обрабатывает все исключения, что могут быть возбуждены в обработчиках. Объекты исключений будут присутствовать в результате, возвращенном методом, во вторых элементах соответствующих вложенных кортежей.

  Поскольку исключения обрабатываются внутри метода, то, если к сигналу привязано несколько обработчиков и в одном из них было возбуждено исключение, последующие обработчики все же будут выполнены.

Объявленный нами сигнал может быть обработан точно так же, как и любой из встроенных в Django:

```
def add_bb_dispatcher(sender, **kwargs):
    print('Объявление в рубрике "%s" с ценой %.2f создано' % \
          (kwargs['rubric'].name, kwargs['instance'].price))

add_bb.connect(add_bb_dispatcher)
```



ГЛАВА 25

Отправка электронных писем

Django предоставляет развитые средства для отправки электронных писем, в том числе составных и с вложениями.

25.1. Настройка подсистемы отправки электронных писем

Настройки подсистемы рассылки писем записываются в модуле settings.py пакета конфигурации:

- `EMAIL_BACKEND` — строка с именем класса, который реализует отправку писем. В составе Django поставляются следующие классы-отправители писем:
  - `django.core.mail.backends.smtp.EmailBackend` — отправляет письма на почтовый сервер по протоколу SMTP. Может использоваться как при разработке сайта, так и при его эксплуатации;

  Следующие классы применяются исключительно при разработке и отладке сайта:
  - `django.core.mail.backends.filebased.EmailBackend` — сохраняет письма в файлах;
  - `django.core.mail.backends.console.EmailBackend` — выводит письма в командной строке;
  - `django.core.mail.backends.locmem.EmailBackend` — сохраняет письма в оперативной памяти. В модуле `django.core.mail` создается переменная `outbox`, и все отправленные письма записываются в нее в виде списка;
  - `django.core.mail.backends.dummy.EmailBackend` — никуда не отправляет, нигде не сохраняет и не выводит письма.

  Значение параметра по умолчанию: `"django.core.mail.backends.smtp.EmailBackend"`;
- `DEFAULT_FROM_EMAIL` — адрес электронной почты отправителя, по умолчанию указываемый в отправляемых письмах (по умолчанию: `"webmaster@localhost"`).

Следующие параметры принимаются во внимание, только если в качестве класса-отправителя писем был выбран `django.core.mail.backends.smtp.EmailBackend`:

- `EMAIL_HOST` — интернет-адрес SMTP-сервера, которому будут отправляться письма (по умолчанию: `"localhost"`);
- `EMAIL_PORT` — номер TCP-порта, через который работает SMTP-сервер, в виде числа (по умолчанию: `25`);
- `EMAIL_HOST_USER` — имя пользователя для аутентификации на SMTP-сервере (по умолчанию — "пустая" строка);
- `EMAIL_HOST_PASSWORD` — пароль для аутентификации на SMTP-сервере (по умолчанию — "пустая" строка).

  Если значение хотя бы одного из параметров `EMAIL_HOST_USER` и `EMAIL_HOST_PASSWORD` равно "пустой" строке, то аутентификация на SMTP-сервере выполняться не будет;
- `EMAIL_USE_SSL` — если `True`, то взаимодействие с SMTP-сервером будет происходить через протокол SSL (Secure Sockets Layer, уровень защищенных сокетов), если `False`, то таковой применяться не будет. Протокол SSL работает через TCP-порт 465. По умолчанию — `False`;
- `EMAIL_USE_TLS` — если `True`, то взаимодействие с SMTP-сервером будет происходить через протокол TLS (Transport Layer Security, протокол защиты транспортного уровня), если `False`, то таковой применяться не будет. Протокол TLS работает через TCP-порт 587. По умолчанию — `False`;

  Внимание!

  Можно указать значение `True` только для одного из параметров: `EMAIL_USE_SSL` или `EMAIL_USE_TLS`.

- `EMAIL_SSL_CERTFILE` — строка с путем к файлу сертификата. Принимается во внимание, только если для одного из параметров: `EMAIL_USE_SSL` или `EMAIL_USE_TLS` — было указано значение `True`. По умолчанию — `None`;
- `EMAIL_SSL_KEYFILE` — строка с путем к файлу с закрытым ключом. Принимается во внимание, только если для одного из параметров: `EMAIL_USE_SSL` или `EMAIL_USE_TLS` — было указано значение `True`. По умолчанию — `None`;
- `EMAIL_TIMEOUT` — промежуток времени, в течение которого класс-отправитель будет пытаться установить соединение с SMTP-сервером, в виде целого числа в секундах. Если соединение установить не удастся, будет возбуждено исключение `timeout` из модуля `socket`. Если для параметра указать значение `None`, то будет использовано значение промежутка времени по умолчанию. Значение параметра по умолчанию — `None`;
- `EMAIL_USE_LOCALTIME` — если `True`, то в заголовках отправляемых писем будет указано локальное время, если `False` — всемирное координированное время (UTC). По умолчанию — `False`;

- EMAIL_FILE_PATH — полный путь к папке, в которой будут сохраняться файлы с письмами. Принимается во внимание, только если в качестве класса-отправителя писем был выбран django.core.mail.backends.filebased.EmailBackend. По умолчанию — "пустая" строка.

25.2. Низкоуровневые инструменты для отправки писем

Инструменты низкого уровня для отправки электронных писем имеет смысл применять лишь в том случае, если нужно отправить письмо с вложениями.

25.2.1. Класс *EmailMessage*: обычное электронное письмо

Класс EmailMessage из модуля django.core.mail позволяет отправить обычное текстовое электронное письмо, возможно, с вложениями.

Конструктор этого класса принимает довольно много параметров. Все они являются именованными и необязательными:

- subject — тема письма;
- body — тело письма;
- from_email — адрес отправителя в виде строки. Также может быть записан в формате *<имя отправителя> <<адрес электронной почты>>*, например: "Admin <admin@supersite.ru>". Если не указан, то адрес будет взят из параметра DEFAULT_FROM_EMAIL настроек сайта;
- to — список или кортеж адресов получателей письма;
- cc — список или кортеж адресов получателей копии письма;
- bcc — список или кортеж адресов получателей скрытой копии письма;
- reply_to — список или кортеж адресов для отправки ответа на письмо;
- attachments — список вложений, которые нужно добавить в письмо. Каждое вложение может быть задано в виде:
  - экземпляра класса MIMEBase из модуля email.mime.base Python или одного из его подклассов;
  - кортежа из трех элементов: строки с именем файла, строки или объекта bytes с содержимым файла и строки с MIME-типом файла;
- headers — словарь с дополнительными заголовками, которые нужно добавить в письмо. Ключи элементов этого словаря задают имена заголовков, а значения элементов — значения заголовков;
- connection — объект соединения для отправки письма (его использование будет описано позже). Если не указан, то для отправки каждого письма будет установлено отдельное соединение.

Для работы с письмами класс `EmailMessage` предоставляет ряд методов:

- `attach()` — добавляет вложение к письму. Форматы вызова метода:

  `attach(<объект вложения>)`
  `attach(<имя файла>, <содержимое файла>[, <MIME-тип содержимого>])`

  Первый формат принимает в качестве параметра *объект вложения*, представленный экземпляром класса `MIMEBase` или одного из его подклассов.

  Второй формат принимает *имя файла*, который будет сформирован в письме в качестве вложения, *содержимое* этого файла в виде строки или объекта `bytes` и строку с *MIME-типом содержимого* файла, которую можно не указывать. Если MIME-тип не указан, то Django определит его по расширению из *имени файла*;

- `attach_file(<путь к файлу>[, <MIME-тип файла>])` — добавляет файл к письму в качестве вложения. Если *MIME-тип* не указан, то Django определит его по расширению файла;

- `send([fail_silently=False])` — отправляет письмо. Если параметру `fail_silently` дать значение `True`, то в случае возникновения нештатной ситуации при отправке письма никаких исключений возбуждено не будет (по умолчанию в таких случаях возбуждается исключение `SMTPException` из модуля `smtplib`);

- `message()` — возвращает экземпляр класса `SafeMIMEText` из модуля `django.core.mail`, представляющий текущее письмо. Этот класс является производным от класса `MIMEBase`, следовательно, может быть указан в списке вложений, задаваемых параметром `attachments` конструктора класса, или в вызове метода `attach()`;

- `recipients()` — возвращает список адресов всех получателей письма и его копий, которые указаны в параметрах `to`, `cc` и `bcc` конструктора класса.

Вот примеры, демонстрирующие наиболее часто встречающиеся на практике случаи отправки электронных писем:

```
from django.core.mail import EmailMessage
# Отправка обычного письма
em = EmailMessage(subject='Test', body='Test', to=['user@supersite.ru'])
em.send()
# Отправка письма с вложением, заданным через параметр attachments
# конструктора. Отметим, что файл password.txt не загружается с диска,
# а формируется программно
em = EmailMessage(subject='Ваш новый пароль',
          body='Ваш новый пароль находится во вложении',
          attachments=[('password.txt', '123456789', 'text/plain')],
          to=['user@supersite.ru'])
em.send()
# Отправка письма с вложением файла, взятого с локального диска
em = EmailMessage(subject='Запрошенный вами файл',
                  body='Получите запрошенный вами файл',
                  to=['user@supersite.ru'])
em.attach_file(r'C:\work\file.txt')
em.send()
```

25.2.2. Формирование писем на основе шаблонов

Электронные письма могут быть сформированы на основе обычных шаблонов Django (см. *главу 11*). Для этого применяется функция `render_to_string()` из модуля `django.template.loader`, описанная в *разд. 9.3.2*.

Вот пример отправки электронного письма, которое формируется на основе шаблона email\letter.txt:

```
from django.core.mail import EmailMessage
from django.template.loader import render_to_string
context = {'user': 'Вася Пупкин'}
s = render_to_string('email/letter.txt', context)
em = EmailMessage(subject='Оповещение', body=s,
                                        to=['vpupkin@othersite.ru'])
em.send()
```

Код шаблона email\letter.txt может выглядеть так:

```
Уважаемый {{ user }}, вам пришло сообщение!
```

25.2.3. Использование соединений. Массовая рассылка писем

При отправке каждого письма, представленного экземпляром класса `EmailMessage`, устанавливается отдельное соединение с SMTP-сервером. Установление соединения отнимает заметное время, которое может достигать нескольких секунд. Если отправляется одно письмо, с такой задержкой еще можно смириться, но при массовой рассылке писем это совершенно неприемлемо.

Параметр `connection` конструктора класса `EmailMessage` позволяет указать объект соединения, используемый для отправки текущего письма. Мы можем задействовать одно и то же соединение для отправки произвольного количества писем, тем самым устранив необходимость каждый раз устанавливать соединение.

Получить соединение мы можем вызовом функции `get_connection()` из модуля `django.core.mail`. Функция в качестве результата вернет то, что нам нужно, — объект соединения.

Для открытия соединения мы вызовем у полученного объекта метод `open()`, а для закрытия — метод `close()`.

Вот пример отправки трех писем с применением одного и того же соединения:

```
from django.core.mail import EmailMessage, get_connection
con = get_connection()
con.open()
email1 = EmailMessage( . . . connection=con)
email1.send()
email2 = EmailMessage( . . . connection=con)
email2.send()
```

```
email3 = EmailMessage( . . . connection=con)
email3.send()
con.close()
```

Недостатком здесь может стать то, что соединение придется указывать при создании каждого письма. Но существует удобная альтернатива — использование метода `send_messages(<отправляемые письма>)` объекта соединения. Список *отправляемых писем*, представленных экземплярами класса `EmailMessage`, передается методу единственным параметром. Пример:

```
from django.core.mail import EmailMessage, get_connection
con = get_connection()
con.open()
email1 = EmailMessage( . . . )
email2 = EmailMessage( . . . )
email3 = EmailMessage( . . . )
con.send_messages([email1, email2, email3])
con.close()
```

25.2.4. Класс *EmailMultiAlternatives*: составное письмо

Класс `EmailMultiAlternatives` из того же модуля `django.core.mail` представляет составное письмо, которое состоит из нескольких частей, записанных в разных форматах. Обычно такое письмо содержит основную часть, представляющую собой обычный текст, и дополнительную часть, которая написана на языке HTML.

Класс `EmailMultiAlternatives` является производным от класса `EmailMessage` и имеет тот же формат вызова конструктора. Объявленный в нем метод `attach_alternative(<содержимое части>, <MIME-тип части>)` добавляет к письму новую часть с указанным в виде строки *содержимым*.

Пример отправки письма, которое, помимо текстовой части, содержит еще и фрагмент, написанный на HTML:

```
from django.core.mail import EmailMultiAlternatives
em = EmailMultiAlternatives(subject='Test', body='Test',
                                          to=['user@supersite.ru'])
em.attach_alternative('<h1>Test</h1>', 'text/html')
em.send()
```

Разумеется, для формирования дополнительных частей в таких письмах можно использовать шаблоны.

25.3. Высокоуровневые инструменты для отправки писем

Если нет необходимости создавать письма с вложениями, то можно воспользоваться высокоуровневыми средствами отправки писем.

25.3.1. Отправка писем по произвольным адресам

Отправку писем по произвольным адресам выполняют две функции из модуля `django.core.mail`:

- `send_mail()` — отправляет одно письмо, возможно, включающее HTML-часть, по указанным адресам. Формат вызова функции:

```
send_mail(<тема>, <тело>, <адрес отправителя>, <адреса получателей>[,
          fail_silently=False][, auth_user=None][,
          auth_password=None][, connection=None][, html_message=None])
```

Адрес отправителя указывается в виде строки. Его также можно записать в формате *<имя отправителя> <<адрес электронной почты>>*, например: `"Admin <admin@supersite.ru>"`. *Адреса получателей* указываются в виде списка или кортежа.

Если параметру `fail_silently` задать значение `True`, то в случае возникновения нештатной ситуации при отправке письма никаких исключений возбуждено не будет. Значение `False` указывает возбудить в таких случаях исключение `SMTPException` из модуля `smtplib`.

Параметры `auth_user` и `auth_password` задают соответственно имя пользователя и пароль для подключения к SMTP-серверу. Если эти параметры не заданы, то их значения будут взяты из параметров `EMAIL_HOST_USER` и `EMAIL_HOST_PASSWORD` настроек проекта (см. *разд. 25.1*).

Параметр `connection` указывает объект соединения, применяемого для отправки письма. Если он не задан, то для отправки письма будет установлено отдельное соединение.

Параметр `html_message` задает строку с HTML-кодом второй части. Если он отсутствует, то письмо будет содержать всего одну часть — текстовую.

Функция возвращает `1` в случае успешной отправки письма и `0`, если письмо по какой-то причине отправить не удалось.

Пример:

```
from django.core.mail import send_mail
send_mail('Test', 'Test!!!', 'webmaster@supersite.ru',
          ['user@othersite.ru'], html_message='<h1>Test!!!</h1>')
```

- `send_mass_mail()` — выполняет отправку писем из указанного *перечня*. Формат вызова:

```
send_mass_mail(<перечень писем>[, fail_silently=False][,
               auth_user=None][, auth_password=None][,
               connection=None])
```

Перечень писем должен представлять собой кортеж, каждый элемент которого описывает одно из отправляемых писем и также представляет собой кортеж из четырех элементов: темы письма, тела письма, адреса отправителя и списка или кортежа адресов получателей.

Назначение остальных параметров этого метода было описано ранее, при рассмотрении метода `send_mail()`.

Для отправки всех писем из *перечня* используется всего одно соединение с SMTP-сервером.

В качестве результата метод `send_mass_mail()` возвращает количество успешно отправленных писем.

Пример:

```
from django.core.mail import send_mass_mail
msg1 = ('Подписка', 'Подтвердите, пожалуйста, подписку',
        'subscribe@supersite.ru',
        ['user@othersite.ru', 'user2@thirdsite.ru'])
msg2 = ('Подписка', 'Ваша подписка подтверждена',
        'subscribe@supersite.ru', ['megauser@megasite.ru'])
send_mass_mail((msg1, msg2))
```

25.3.2. Отправка писем зарегистрированным пользователям

Класс `User`, реализующий модель пользователя, предлагает метод `email_user()`, отправляющий письмо текущему зарегистрированному пользователю:

```
email_user(<тема>, <тело>[, from_email=None][,
                          <дополнительные параметры>])
```

Параметр `from_email` указывает адрес отправителя. Если он опущен, то адрес отправителя будет взят из параметра `DEFAULT_FROM_EMAIL` настроек проекта.

Все *дополнительные параметры*, указанные в вызове метода, будут без изменений переданы функции `send_mail()` (см. *разд. 25.3.1*), которая служит для выполнения отправки.

Пример:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.get(username='admin')
>>> user.email_user('Подъем!', 'Admin, не спи!', fail_silently=True)
```

В этом примере параметр `fail_silently` будет передан функции `send_mail()`, используемой методом `email_user()`.

25.3.3. Отправка писем администраторам и редакторам сайта

Django может рассылать письма по адресам, указанным в двух параметрах настроек сайта. Один из этих параметров задает список адресов администраторов, получающих сообщения об ошибках в программном коде сайта, а второй — список адресов редакторов, которым будут отсылаться уведомления об отсутствующих страницах.

Обычно инструменты, рассылающие письма по этим адресам, применяются в подсистеме журналирования (речь о которой пойдет в *главе 28*).

Параметры, управляющие этими инструментами, записываются в модуле settings.py пакета конфигурации:

- ADMINS — список администраторов. Каждый его элемент обозначает одного из администраторов и должен представлять собой кортеж из двух элементов: имени администратора и его адреса электронной почты. Пример:

```
ADMIN = [
    ('Admin1', 'admin1@supersite.ru'),
    ('Admin2', 'admin2@othersite.ru'),
    ('MegaAdmin', 'megaadmin@megasite.ru')
]
```

  Значение по умолчанию — "пустой" список;

- MANAGERS — список редакторов. Задается в том же формате, что и список администраторов из параметра ADMIN. По умолчанию — "пустой" список;

- SERVER_EMAIL — адрес электронной почты отправителя, указываемый в письмах, что отправляются администраторам и редакторам (по умолчанию: "root@localhost");

- EMAIL_SUBJECT_PREFIX — префикс, который добавляется к теме каждого письма, отправляемого администраторам и редакторам (по умолчанию: "[Django]").

Для отправки писем администраторам применяется функция mail_admins(), для отправки писем редакторам — функция mail_managers(). Обе функции объявлены в модуле django.core.mail и имеют одинаковый формат вызова:

```
mail_admins|mail_managers(<тема>, <тело>[, fail_silently=False][,
                                  connection=None][, html_message=None])
```

Если параметру fail_silently задать значение True, то в случае возникновения нештатной ситуации при отправке письма никаких исключений возбуждено не будет. Значение False указывает возбудить в таких случаях исключение SMTPException из модуля smtplib.

Параметр connection указывает объект соединения, применяемого для отправки письма. Если он не задан, то для отправки письма будет установлено отдельное соединение.

Параметр html_message задает строку с HTML-кодом второй части. Если он отсутствует, то письмо будет содержать всего одну часть — текстовую.

Пример:

```
from django.core.mail import mail_managers
mail_managers('Подъем!', 'Редакторы, не спите!',
              html_message='<strong>Редакторы, не спите!</strong>')
```

25.4. Отладочный SMTP-сервер

Чтобы проверить в работе рассылку электронных писем, можно воспользоваться встроенным в Python отладочным SMTP-сервером, выводящим содержимое отправленных ему писем непосредственно в командной строке. Он запускается подачей команды формата:

```
python -m smtpd -n -c DebuggingServer ↳
localhost:<номер используемого TCP-порта>
```

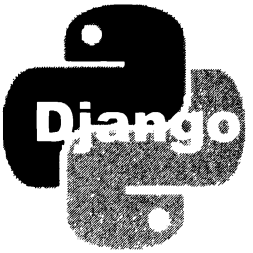
Например, чтобы запустить отладочный сервер и указать ему использовать для приема писем порт № 1025, следует задать команду:

```
python -m smtpd -n -c DebuggingServer localhost:1025
```

Для использования отладочного SMTP-сервера в модуле settings.py пакета конфигурации необходимо лишь указать номер требуемого TCP-порта — в параметре `EMAIL_PORT` (подробности — в *разд. 25.1*):

```
EMAIL_PORT = 1025
```

Завершить работу отладочного сервера можно, закрыв окно командной строки, в которой он был запущен.



ГЛАВА 26

Кэширование

Когда веб-обозреватель получает от веб-сервера какой-либо файл, он сохраняет его на локальном диске — выполняет его *кэширование*. Впоследствии, если этот файл не изменился, веб-обозреватель использует его кэшированную копию вместо того, чтобы вновь загружать с сервера, — это заметно увеличивает производительность. Так работает *кэширование на стороне клиента*.

Django предоставляет ряд инструментов для управления кэшированием на стороне клиента. Мы можем отправлять в заголовке запроса временну́ю отметку последнего изменения страницы или какой-либо признак, указывающий, изменилась ли она с момента последнего обращения к ней. Также мы можем отключать кэширование каких-либо страниц на стороне клиента, если хотим, чтобы они всегда отображали актуальную информацию.

Помимо этого, Django может выполнять *кэширование на стороне сервера*, сохраняя какие-либо данные, фрагменты страниц или даже целые страницы в особом хранилище — *кэше сервера*. Благодаря этому можно увеличить быстродействие сайтов с высокой нагрузкой.

26.1. Кэширование на стороне сервера

Кэш стороны сервера в Django организован по принципу словаря Python: каждая кэшируемая величина сохраняется в нем под уникальным ключом. Ключ этот может как генерироваться самим Django на основе каких-либо сведений (например, интернет-адреса страницы), так и задаваться произвольно.

26.1.1. Подготовка подсистемы кэширования на стороне сервера

26.1.1.1. Настройка подсистемы кэширования на стороне сервера

Все настройки этой подсистемы указываются в параметре `CACHES` модуля settings.py пакета конфигурации.

Значением этого параметра должен быть словарь. Ключи его элементов указывают псевдонимы созданных кэшей, которых может быть произвольное количество. По умолчанию будет использоваться кэш с псевдонимом `default`.

В качестве значений элементов этого словаря также указываются словари, хранящие собственно параметры соответствующего кэша. Каждый элемент вложенного словаря указывает отдельный параметр.

Вот значение параметра `CACHES` по умолчанию:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
    }
}
```

Оно задает единственный кэш, используемый по умолчанию и сохраняющий данные в оперативной памяти.

Доступные для указания параметры кэша:

- `BACKEND` — строка с именем класса, выполняющего сохранение кэшируемых данных. В составе Django поставляются следующие классы:
  - `django.core.cache.backends.db.DatabaseCache` — сохраняет данные в указанной таблице базы данных. Обеспечивает среднюю производительность и высокую надежность, но дополнительно нагружает базу данных;
  - `django.core.cache.backends.filebased.FileBasedCache` — сохраняет данные в файлах, находящихся в заданной папке. Немного медленнее предыдущего класса, но не нагружает базу данных;
  - `django.core.cache.backends.locmem.LocMemCache` — сохраняет данные в оперативной памяти. Дает наивысшую производительность, но при отключении компьютера содержимое кэша будет потеряно;
  - `django.core.cache.backends.memcached.MemcachedCache` — использует для хранения данных популярную программу Memcached (ее применение будет описано далее);
  - `django.core.cache.backends.dummy.DummyCache` — вообще не сохраняет кэшируемые данные. Служит исключительно для отладки;
- `LOCATION` — назначение параметра зависит от класса, указанного в параметре `BACKEND`:
  - имя таблицы — если выбран класс, хранящий кэш в таблице базы данных;
  - полный путь к папке — если выбран класс, хранящий данные в файлах по указанному пути;
  - псевдоним хранилища — если выбран класс, хранящий данные в оперативной памяти. Указывается только в том случае, если используются несколько кэшей такого типа. Пример:

```
CACHES = {
    'default': {
        'BACKEND':
                'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'cache1',
    }
    'special': {
        'BACKEND':
                'django.core.cache.backends.locmem.LocMemCache',
        'LOCATION': 'cache2',
    }
}
```

- интернет-адрес и используемый TCP-порт — если выбран класс, сохраняющий данные в Memcached.

Значение по умолчанию — "пустая" строка;

- TIMEOUT — время, в течение которого кэшированное значение будет считаться актуальным, в виде целого числа в секундах. Устаревшие значения впоследствии будут удалены. По умолчанию: 300;
- OPTIONS — дополнительные параметры кэша. Значение указывается в виде словаря, каждый элемент которого задает отдельный параметр. Поддерживаются два универсальных параметра, обрабатываемые всеми классами-хранилищами данных:
  - MAX_ENTRIES — количество значений, которые могут храниться в кэше, в виде целого числа (по умолчанию: 300);
  - CULL_FREQUENCY — часть кэша, которая будет очищена, если количество значений в кэше превысит величину из параметра MAX_ENTRIES, указанная в виде целого числа. Так, если дать параметру CULL_FREQUENCY значение 2, то при заполнении кэша будет удалена половина хранящихся в нем значений. Если задать значение 0, то при заполнении кэша из него будут удалены все значения. Значение по умолчанию: 3.

  Здесь же можно задать параметры, специфичные для классов-хранилищ, которые поставляются в составе сторонних библиотек;
- KEY_PREFIX — заданный по умолчанию префикс, который участвует в формировании конечного ключа, записываемого в кэше (по умолчанию — "пустая" строка);
- VERSION — назначенная по умолчанию версия кэша, которая участвует в формировании конечного ключа, записываемого в кэше (по умолчанию: 1);
- KEY_FUNCTION — строка с именем функции, которая формирует конечный ключ, записываемый в кэше, из префикса, номера версии и ключа кэшируемого значения. По умолчанию используется функция, которая составляет конечный ключ из префикса, номера версии и ключа, разделяя их символами двоеточия, и имеет следующий вид:

```
    def make_key(key, key_prefix, version):
        return ':'.join([key_prefix, str(version), key])
```

Пример указания параметров кэша:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'cache_table',
        'TIMEOUT': 120,
        'OPTIONS': {
            'MAX_ENTRIES': 200,
        }
    }
}
```

26.1.1.2. Создание таблицы для хранения кэша

Если был выбран класс, сохраняющий кэшированные данные в таблице базы данных, то эту таблицу необходимо создать. В этом нам поможет команда `createcachetable` утилиты manage.py:

```
manage.py createcachetable [--dry-run]
```

Дополнительный ключ `--dry-run` выводит на экран сведения о создаваемой таблице, но не создает ее.

26.1.1.3. Применение Memcached

Memcached — популярная программа кэширования, сохраняющая данные исключительно в оперативной памяти.

Чтобы использовать ее для кэширования данных в Django, необходимо:

- установить дополнительную библиотеку python-memcached подачей команды:
  ```
  pip install python-memcached
  ```
- записать в параметре `BACKEND` настроек кэша имя класса `django.core.cache.backends.memcached.MemcachedCache`;
- указать в параметре `LOCATION` настроек кэша строку формата:

  <интернет-адрес Memcached>:<номер TCP-порта, используемого Memcached>

  Например, если Memcached установлена на локальном хосте и использует порт по умолчанию № 11211, следует записать:

  ```
  CACHES = {
      'default': {
          'BACKEND':
                  'django.core.cache.backends.memcached.MemcachedCache',
          'LOCATION': 'localhost:11211',
      }
  }
  ```

Если кэшируемые данные одновременно сохраняются в нескольких экземплярах Memcached, то в параметре LOCATION следует указать список или кортеж из строк в приведенном ранее формате. Например, если для кэширования используются два экземпляра Memcached, доступные по интернет-адресам 172.18.27.240 и 172.18.27.241 и TCP-портам 112211 и 22122, то следует указать такие настройки:

```
CACHES = {
    'default': {
        'BACKEND':
                'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': ('172.18.27.240:11211', '172.18.27.241:22122'),
    }
}
```

26.1.2. Высокоуровневые средства кэширования

Высокоуровневые средства кэшируют либо все страницы сайта, либо только страницы, сгенерированные отдельными контроллерами.

Ключ, под которым сохраняется кэшированная страница, формируется самим Django на основе ее пути и набора GET-параметров.

26.1.2.1. Кэширование всего веб-сайта

При этом подсистема кэширования Django работает согласно следующим принципам:

- кэшируются все страницы, сгенерированные контроллерами в ответ на получение GET- и HEAD-запросов, с кодом статуса 200 (т. е. запрос был обработан успешно);
- страницы с одинаковыми путями, но разным набором GET-параметров и разными cookie считаются разными, и для каждой из них в кэше создается отдельная копия.

Чтобы запустить кэширование всего сайта, необходимо добавить посредники `django.middleware.cache.UpdateCacheMiddleware` и `django.middleware.cache.FetchFromCacheMiddleware` в список зарегистрированных в проекте (параметр MIDDLEWARE):

```
MIDDLEWARE = [
    . . .
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
    . . .
]
```

Еще можно указать дополнительные параметры:

- CACHE_MIDDLEWARE_ALIAS — псевдоним кэша, в котором будут сохраняться страницы (по умолчанию: "default");
- CACHE_MIDDLEWARE_SECONDS — время, в течение которого кэшированная страница будет считаться актуальной, в виде целого числа в секундах (по умолчанию: 600);
- CACHE_MIDDLEWARE_KEY_PREFIX — префикс конечного ключа, применяемый только при кэшировании всего сайта (по умолчанию — "пустая" строка).

В ответы, отсылаемые клиентам и содержащие страницы, добавляются два следующих заголовка:

- Expires — в качестве значения задается временна́я отметка устаревания кэшированной страницы, полученная сложением текущих даты и времени и значения из параметра CACHE_MIDDLEWARE_SECONDS;
- Cache-Control — добавляется параметр max-age со значением, взятым из параметра CACHE_MIDDLEWARE_SECONDS.

Во многих случаях кэширование всего сайта — наилучший вариант повышения его производительности. Оно быстро реализуется, не требует ни сложного программирования, ни переписывания кода контроллеров, ответы которых нужно кэшировать.

При кэшировании всего сайта Django учитывает ситуации, при которых одна и та же страница может генерироваться в разных редакциях в зависимости от каких-либо "внутренних" условий (например, выполнил пользователь вход на сайт или нет). Дело в том, что идентификатор сессии сохраняется в cookie, а, как говорилось ранее, страницы с одинаковыми путями, но разными cookie (и GET-параметрами) считаются разными, и в кэше сохраняются их отдельные редакции — для каждого из наборов cookie. Таким образом, ситуации, при которых пользователь открывает страницу в качестве гостя, выполняет вход на сайт, заходит после этого на ту же страницу и получает ее устаревшую, "гостевую", копию из кэша, полностью исключены.

26.1.2.2. Кэширование на уровне отдельных контроллеров

Если же требуется кэшировать не все страницы сайта, а лишь некоторые (например, наименее часто обновляемые и при этом наиболее часто посещаемые), то следует применить кэширование на уровне отдельных контроллеров.

Кэширование на уровне контроллера задействует декоратор cache_page() из модуля django.views.decorators.cache:

```
cache_page(<время хранения страницы>[, cache=None][, key_prefix=None])
```

Время хранения сгенерированной страницы в кэше задается в виде целого числа в секундах.

Необязательный параметр cache указывает псевдоним кэша, в котором будет сохранена страница. Если он не указан, будет использован кэш по умолчанию с псевдонимом default.

В необязательном параметре `key_prefix` можно указать другой префикс конечного ключа. Если он не задан, то применяется префикс из параметра `KEY_PREFIX` настроек текущего кэша.

Декоратор `cache_page()` указывается непосредственно у контроллера-функции, результаты работы которого необходимо кэшировать:

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 5)
def by_rubric(request, pk):
    . . .
```

Также этот декоратор можно указать в объявлении маршрута, ведущего на нужный контроллер. Это можно использовать для кэширования страниц, генерируемых контроллерами-классами. Пример:

```
from django.views.decorators.cache import cache_page

urlpatterns = [
    . . .
    path('<int:rubric_id>/', cache_page(60 * 5)(by_rubric)),
    path('', cache_page(60 * 5)(BbIndexView.as_view())),
]
```

Выбрав кэширование на уровне контроллеров, мы сможем указать, какие страницы следует кэшировать, а какие — нет. Понятно, что в первую очередь кэшировать стоит страницы, которые просматриваются большей частью посетителей и не изменяются в зависимости от каких-либо "внутренних" условий. Страницы же, меняющиеся в зависимости от таких условий, лучше не кэшировать (или, как вариант, применять средства управления кэшированием, которые будут рассмотрены прямо сейчас).

26.1.2.3. Управление кэшированием

Если в кэше требуется сохранять отдельную редакцию страницы для каждого из возможных значений заданного "внутреннего" признака, то следует как-то указать этот признак подсистеме кэширования, поместив его в заголовок `Vary` ответа. Сделать это можно посредством декоратора `vary_on_headers()` из модуля `django.views.decorators.vary`:

```
vary_on_headers(<заголовок 1>, <заголовок 2> . . . <заголовок n>)
```

В качестве признаков, на основе значений которых в кэше будут создаваться отдельные редакции страницы, здесь указываются *заголовки* запроса, записанные в виде строк. Можно указать произвольное количество *заголовков*.

Вот как можно задать создание в кэше отдельных редакций одной и той же страницы для разных значений заголовка `User-Agent` (он обозначает программу веб-обозревателя, используемую клиентом):

```
from django.views.decorators.vary import vary_on_headers

@vary_on_headers('User-Agent')
def index(request):
    . . .
```

Теперь в кэше будет создаваться отдельная копия страницы на каждый веб-обозреватель, который ее запрашивал.

Чтобы создавать отдельные редакции страницы для гостя и пользователя, выполнившего вход, можно воспользоваться тем, что при создании сессии (в которой сохраняется ключ вошедшего на сайт пользователя) веб-обозревателю посылается cookie с идентификатором этой сессии. А при отправке запроса на сервер все cookie, сохраненные для домена, по которому отправляется запрос, посылаются серверу в заголовке `Cookie`. В таком случае нужно написать код:

```
@vary_on_headers('Cookie')
def user_profile(request):
    . . .
```

Пример указания в декораторе `vary_on_headers()` нескольких заголовков:

```
@vary_on_headers('User-Agent', 'Cookie')
def user_account(request):
    . . .
```

Поскольку создание в кэше разных редакций страницы для разных значений заголовка `Cookie` является распространенной практикой, Django предоставляет декоратор `vary_on_cookie()` из того же модуля `django.views.decorators.vary`. Пример:

```
from django.views.decorators.vary import vary_on_cookie

@vary_on_cookie
def user_profile(request):
    . . .
```

26.1.3. Низкоуровневые средства кэширования

Низкоуровневые средства кэширования применяются для сохранения в кэше сложных частей страниц и значений, получение которых сопряжено с интенсивными вычислениями.

При кэшировании на низком уровне ключ, под которым сохраняется кэшируемая величина, указывается самим разработчиком. Этот ключ объединяется с префиксом и номером версии для получения *конечного ключа*, под которым значение и будет сохранено в кэше.

26.1.3.1. Кэширование фрагментов веб-страниц

Для кэширования фрагментов страниц понадобится библиотека тегов с псевдонимом `cache`:

```
{% load cache %}
```

Собственно кэширование выполняет парный тег `cache . . . endcache`:

```
cache <время хранения фрагмента> <ключ фрагмента> [<набор параметров>] ↳
[using="template_fragments"]
```

Время хранения фрагмента в кэше указывается в секундах. Если задать значение `None`, то фрагмент будет храниться вечно (пока не будет явно удален).

Ключ фрагмента необходим для формирования конечного ключа, под которым фрагмент будет сохранен в кэше, и должен быть уникальным в пределах всех шаблонов текущего проекта.

Набор параметров указывается в случаях, если требуется сохранять разные копии фрагмента для каждой комбинации значений заданных в *наборе* параметров.

Необязательный параметр `using` указывает псевдоним кэша. По умолчанию будет применяться кэш с псевдонимом `template_fragments` или, если таковой отсутствует, кэш по умолчанию.

Вот так мы можем сохранить в кэше панель навигации нашего сайта на 300 секунд:

```
{% cache 300 navbar %}
<nav>
    <a href="{% url 'bboard:index' %}">Главная</a>
    {% for rubric in rubrics %}
    <a href="{% url 'bboard:by_rubric' rubric_id=rubric.pk %}">
    {{ rubric }}</a>
    {% endfor %}
</nav>
{% endcache %}
```

Если нам нужно хранить в кэше две копии фрагмента, одна из которых должна выдаваться гостям, а другая — пользователям, выполнившим вход, мы используем код такого вида:

```
{% cache 300 navbar request.user.is_authenticated %}
<nav>
    <a href="{% url 'bboard:index' %}">Главная</a>
    {% if user.is_authenticated %}
    <a href="{% url 'logout' %}">Выйти</a>
    {% else %}
    <a href="{% url 'login' %}">Войти</a>
    {% endif %}
</nav>
{% endcache %}
```

А если нужно хранить по отдельной копии фрагмента еще и для каждого из зарегистрированных пользователей сайта, мы напишем код:

```
{% cache 300 greeting request.user.is_authenticated ↳
request.user.username %}
{% if user.is_authenticated %}
<p>Добро пожаловать, {{ user.username }}!</p>
```

```
{% endif %}
{% endcache %}
```

26.1.3.2. Кэширование произвольных значений

Словарь со всеми кэшами, объявленными в настройках проекта, хранится в переменной `caches` из модуля `django.core.cache`. Ключи элементов этого словаря совпадают с псевдонимами кэшей, а значениями элементов являются сами кэши, представленные особыми объектами. Пример:

```
from django.core.cache import caches
default_cache = caches['default']
special_cache = caches['special']
```

Если кэша с указанным псевдонимом не существует, то будет возбуждено исключение `InvalidCacheBackendError` из модуля `django.core.cache.backends.base`.

Переменная `cache`, также объявленная в модуле `django.core.cache`, хранит ссылку на объект кэша по умолчанию.

Любой объект кэша поддерживает следующие методы:

- `set(<ключ>, <значение>[, timeout=<время хранения>][, version=None])` — заносит в кэш *значение* под заданным *ключом*. Если заданный *ключ* уже существует в кэше, перезаписывает сохраненное под ним значение. Параметр `timeout` указывает время хранения значения в виде целого числа в секундах. Если он не задан, используется значение параметра `TIMEOUT` соответствующего кэша (см. *разд. 26.1.1.1*).

  В необязательном параметре `version` можно задать версию, на основе которой будет формироваться конечный ключ для сохраняемого в кэше значения. Если версия не указана, будет использовано значение параметра `VERSION` соответствующего кэша.

  Примеры:

  ```
  from django.core.cache import cache
  cache.set('rubrics', Rubric.objects.all())
  cache.set('rubrics_sorted', Rubric.objects.order_by('name'), timeout=240)
  ```

  Имеется возможность сохранить произвольное количество значений под одним ключом, просто указав для этих значений разные номера версий:

  ```
  # Если номер версии не указан, для сохраняемого значения будет
  # установлен номер версии из параметра VERSION (по умолчанию — 1)
  cache('val', 10)
  cache('val', 100, version=2)
  cache('val', 1000, version=3)
  ```

- `add(<ключ>, <значение>[, timeout=<время хранения>][, version=None])` — то же самое, что `set()`, но сохраняет *значение* только в том случае, если указанный *ключ* не существует в кэше. В качестве результата возвращает `True`, если значение было сохранено в кэше, и `False` — в противном случае;

- `get(<ключ>[, <значение по умолчанию>][, version=None])` — извлекает из кэша значение, ранее сохраненное под заданным *ключом*, и возвращает его в качестве результата. Если указанного *ключа* в кэше нет (например, значение еще не создано или уже удалено по причине устаревания), то возвращает *значение по умолчанию* или `None`, если оно не указано. Параметр `version` указывает номер версии. Пример:

```
rubrics_sorted = cache.get('rubrics_sorted')
rubrics = cache.get('rubrics', Rubric.objects.all())
```

Сохранив ранее набор значений под одним ключом, мы можем извлекать эти значения, задавая для них номера версий, под которыми они были сохранены:

```
# Извлекаем значение, сохраненное под версией 1
val1 = cache.get('val')
val2 = cache.get('val', version=2)
val3 = cache.get('val', version=3)
```

- `get_or_set(<ключ>, <значение>[, timeout=<время хранения>][, version=None])` — если указанный *ключ* существует в кэше, то извлекает хранящееся под ним значение и возвращает в качестве результата. В противном случае заносит в кэш *значение* под этим *ключом* и также возвращает это значение в качестве результата. Параметр `timeout` задает время хранения, а параметр `version` — номер версии. Пример:

```
rubrics = cache.get_or_set('rubrics', Rubric.objects.all(), 240)
```

- `incr(<ключ>[, delta=1][, version=None])` — увеличивает значение, хранящееся в кэше под заданным *ключом*, на величину, которая указана параметром `delta` (по умолчанию: `1`). В качестве результата возвращает новое значение. Параметр `version` задает номер версии. Пример:

```
cache.set('counter', 0)
cache.incr('counter')
cache.incr('counter', delta=2)
```

- `decr(<ключ>[, delta=1][, version=None])` — уменьшает значение, хранящееся в кэше под заданным *ключом*, на величину, которая указана параметром `delta` (по умолчанию: `1`). В качестве результата возвращает новое значение. Параметр `version` задает номер версии;

- `has_key(<ключ>[, version=None])` — возвращает `True`, если в кэше существует значение с указанным ключом, и `False` — в противном случае. Параметр `version` задает номер версии. Пример:

```
if cache.has_key('counter'):
    # Значение с ключом counter в кэше существует
```

- `delete(<ключ>[, version=None])` — удаляет из кэша значение под указанным *ключом*. Параметр `version` задает номер версии. Пример:

```
cache.delete('rubrics')
cache.delete('rubrics_sorted')
```

- `set_many(<данные>[, timeout=<время хранения>][, version=None])` — заносит в кэш заданные *данные*, представленные в виде словаря. Параметр `timeout` указывает время хранения данных, а параметр `version` — номер версии. Пример:

```
data = {
    'rubrics': Rubric.objects.get(),
    'rubrics_sorted': Rubric.objects.order_by('name')
}
cache.set_many(data, timeout=600)
```

- `get_many(<ключи>[, version=None])` — извлекает из кэша значения, ранее сохраненные под заданными *ключами*, и возвращает в виде словаря. *Ключи* должны быть представлены в виде списка или кортежа. Параметр `version` указывает номер версии. Пример:

```
data = cache.get_many(['rubrics', 'counter'])
rubrics = data['rubrics']
counter = data['counter']
```

- `delete_many(<ключи>[, version=None])` — удаляет из кэша значения под указанными *ключами*, которые должны быть представлены в виде списка или кортежа. Параметр `version` задает номер версии. Пример:

```
cache.delete_many(['rubrics_sorted', 'counter'])
```

- `touch(<ключ>[, timeout=<время хранения>])` — задает для значения с заданным *ключом* новое время хранения, указанное в параметре `timeout`. Если этот параметр опущен, то задается время кэширования из параметра `TIMEOUT` настроек кэша;
- `incr_version(<ключ>[, delta=1][, version=None])` — увеличивает версию значения, хранящегося в кэше под заданными *ключом* и версией, которая указана в параметре `version`, на величину, заданную в параметре `delta`. Новый номер версии возвращается в качестве результата;
- `decr_version(<ключ>[, delta=1][, version=None])` — уменьшает версию значения, хранящегося в кэше под заданными *ключом* и версией, которая указана в параметре `version`, на величину, заданную в параметре `delta`. Новый номер версии возвращается в качестве результата;
- `clear()` — полностью очищает кэш;

> ***Внимание!***
>
> Вызов метода `clear()` выполняет полную очистку кэша, при которой из него удаляются абсолютно все данные.

- `close()` — закрывает соединение с хранилищем, используемым в качестве кэша.

26.2. Использование Redis

Redis — другая популярная программа для кэширования, сохраняющая кэшируемые данные в файлах. Использовать ее для кэширования Django-сайтов позволяет дополнительная библиотека django-redis.

На заметку

Документация по самой программе Redis находится по адресу **https://redis.io/**, а полная документация по библиотеке django-redis — по адресу **https://niwinz.github.io/django-redis/latest/**.

26.2.1. Установка django-redis и основные настройки кэша

Библиотека устанавливается подачей команды:

```
pip install django-redis
```

Помимо django-redis, будет установлена библиотека redis, необходимая для работы.

Далее в настройках кэша следует указать следующие параметры:

- `BACKEND` — **имя класса** `django_redis.cache.RedisCache`;
- `location` — **интернет-адрес для подключения к базе данных Redis формата:**

  <протокол>`://[`*<пароль>*`@]`*<интернет-адрес Redis>*`:`↳
  <используемый TCP-порт>`/`*<номер базы данных Redis>*

  или

  <протокол>`://[`*<пароль>*`@]`*<интернет-адрес Redis>*`:`↳
  <используемый TCP-порт>`?db=`*<номер базы данных Redis>*

  В качестве *протокола* можно указать `redis` (простой протокол) или `rediss` (защищенный протокол). *Пароль* задается только в случае, если для доступа к Redis используется аутентификация.

Примеры задания настроек:

```
# Redis установлен на локальном хосте, работает по обычному протоколу
# через TCP-порт по умолчанию № 6379 без аутентификации. Используется
# база данных № 0
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'redis://localhost:6379/0',
    }
}

# Redis установлен на хосте 172.18.27.240, работает по защищенному
# протоколу через TCP-порт № 3697 и доступен по паролю "my-redis".
# Используется база данных № 432
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'rediss://my-redis@172.18.27.240:3697/432',
    }
}
```

Если пароль не может быть указан непосредственно в интернет-адресе, поскольку содержит недопустимые символы, его можно записать в дополнительном параметре PASSWORD:

```
CACHES = {
    'default': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'rediss://172.18.27.240:3697/432',
        'OPTIONS': {
            'PASSWORD': 'my Redis server',
        }
    }
}
```

Чтобы использовать кэш Redis для хранения сессий, необходимо указать непосредственно в настройках проекта (а не кэша!) следующие параметры:

```
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'
SESSION_CACHE_ALIAS = <псевдоним кэша в виде строки>
```

Пример:

```
CACHES = {
    'default': {
        . . .
    },
    'session_storage': {
        'BACKEND': 'django_redis.cache.RedisCache',
        'LOCATION': 'redis://localhost:6379/2',
    }
}
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'
SESSION_CACHE_ALIAS = 'session_storage'
```

26.2.2. Дополнительные инструменты кэширования, предоставляемые django-redis

При занесении значения в кэш Redis вызовом метода `set()` (см. *разд. 26.1.3.2*) можно сделать его "вечным", указав у параметра `timeout` значение `None`:

```
cache.set('rubrics', Rubric.objects.all(), timeout=None)
```

Такое значение никогда не устареет и может быть удалено только явно, вызовом метода `delete()`, `delete_many()` или `clear()`.

Объект кэша Redis поддерживает следующие дополнительные методы:

- `ttl(<ключ>[, version=None])` — возвращает оставшееся время хранения (в секундах) кэшированного значения с заданным *ключом*. Если значение "вечное", возвращается `None`. Параметр `version` задает номер версии. Примеры:

  ```
  >>> cache.set('platform', 'Django', timeout=20)
  . . .
  ```

```
>>> cache.ttl('platform')
9
>>> print(cache.ttl('rubrics'))
None
```

- `expire(<ключ>, timeout=<время хранения>[, version=None])` — указывает для значения с заданным *ключом* другое *время хранения* (в секундах). Параметр `version` задает номер версии. Пример:

```
>>> cache.expire('rubrics', timeout=3600)
>>> cache.ttl('rubrics')
3595
```

- `persist(<ключ>[, version=None])` — делает значение с заданным *ключом* "вечным". Параметр `version` задает номер версии. Пример:

```
>>> cache.persist('rubrics')
>>> print(cache.ttl('rubrics'))
None
```

- `keys(<шаблон>[, version=None])` — ищет в кэше ключи, совпадающие с заданным *шаблоном*, и возвращает их в виде списка. В шаблоне можно использовать специальные символы `*` (обозначает произвольное количество любых символов) и `?` (обозначает один произвольный символ). Параметр `version` задает номер версии. Примеры:

```
>>> cache.clear()
>>> cache.set('rubric1', 'Недвижимость')
>>> cache.set('rubric2', 'Транспорт')
>>> cache.set('rubric223', 'Бытовая техника')
>>> cache.set('bb1', 'Мотоцикл')
>>> cache.keys('rubric?')
['rubric2', 'rubric1']
>>> cache.keys('rubric*')
['rubric2', 'rubric223', 'rubric1']
>>> cache.keys('*b*')
['rubric2', 'bb1', 'rubric223', 'rubric1']
```

- `iter_keys(<шаблон>[, version=None])` — то же самое, что и `keys()`, но возвращает итератор, последовательно выдающий совпадающие с *шаблоном* ключи. Параметр `version` задает номер версии. Пример:

```
>>> for k in cache.iter_keys('rubric?'): print(k)
...
rubric1
rubric2
```

- `delete_pattern(<шаблон> [, prefix=None][, version=None])` — удаляет значения с ключами, совпадающими с заданным *шаблоном*. Параметр `prefix` указывает префикс, который должен присутствовать в начале каждого ключа (если не задан, префикс не учитывается). Параметр `version` указывает номер версии. Пример:

```
>>> cache.delete_pattern('bb?')
1
```

- `lock()` — блокирует значение с указанным *ключом*:

```
lock(<ключ>[, version=None][, timeout=None][, sleep=0.1][,
                                              blocking_timeout=None])
```

Необязательные параметры:

- `version` — номер версии;
- `timeout` — время существования блокировки (в секундах). Если `None`, то блокировка будет существовать до тех пор, пока не будет явно снята;
- `sleep` — время ожидания между последовательными попытками установить блокировку, если значение с указанным ключом уже заблокировано другим процессом (в секундах);
- `blocking_timeout` — время, в течение которого библиотека будет пытаться заблокировать значение.

В качестве результата возвращается экземпляр класса `Lock` из модуля `redis`, представляющий наложенную на значение блокировку. Класс `Lock` поддерживает протокол обработчиков контента, вследствие чего его экземпляр может использоваться в языковой конструкции `with`. Пример:

```
with.lock('rubrics'):
    # Делаем что-либо с заблокированным значением
```

26.2.3. Расширенные настройки django-redis

Следующие расширенные настройки записываются в дополнительных параметрах кэша (в словаре, присваиваемом параметру `OPTIONS` настроек кэша):

- `COMPRESSOR` — имя модуля, выполняющего сжатие кэшируемых данных, в виде строки. Библиотека включает следующие модули сжатия:
  - `django_redis.compressors.identity.IdentityCompressor` — вообще не сжимает данные (используется по умолчанию);
  - `django_redis.compressors.zlib.ZlibCompressor` — сжимает данные по алгоритму zlib;
  - `django_redis.compressors.lzma.LzmaCompressor` — сжимает по алгоритму lzma;
  - `django_redis.compressors.lz4.Lz4Compressor` — сжимает по алгоритму LZ4;
- `SOCKET_CONNECT_TIMEOUT` — время, в течение которого Django будет пытаться установить соединение с сервером Redis, в виде целого числа в секундах. Если соединиться за указанное время с Redis не получится, будет возбуждено исключение `ConnectionError` из модуля `redis.exceptions`;
- `SOCKET_TIMEOUT` — время, в течение которого Django будет ожидать завершения операции чтения или записи значения из кэша, в виде целого числа в секундах.

Если прочитать или записать значение за заданное время не удастся, будет возбуждено исключение `TimeoutError` из модуля `redis.exceptions`;

- `IGNORE_EXCEPTIONS` — если `True`, то никакие исключения в случае неполадок с Redis возбуждаться не будут, если `False` — будут (по умолчанию — `False`).

> ***Внимание!***
>
> **Подавлять исключения указанием параметра `IGNORE_EXCEPTIONS` со значением `True` следует, только если Redis используется исключительно для кэширования. Если же он служит и для хранения сессий, то следует разрешить исключения — в противном случае сессии сохраняться не будут, и сайт не будет работать как положено.**

Следующие параметры записываются непосредственно в настройках проекта:

- `DJANGO_REDIS_IGNORE_EXCEPTIONS` — значение `True` включает подавление исключений, возникающих в случае неполадок, для всех Redis-кэшей, описанных в параметре `CACHES`. По умолчанию: `False` (подавлять исключения в кэше или не подавлять, определяет дополнительный параметр `IGNORE_EXCEPTIONS`);
- `DJANGO_REDIS_LOG_IGNORED_EXCEPTIONS` — если `True`, то исключения, возбужденные в случае неполадок с Redis, в любом случае будут записываться в журнал, даже если в настройках кэша указано подавлять их. Если `False`, то подавленные исключения записываться в журнал не будут. Принимается во внимание, только если включено подавление исключений. По умолчанию — `False`;
- `DJANGO_REDIS_LOGGER` — имя класса регистратора, выполняющего журналирование всех исключений в кэшах Redis, в виде строки. Если не указан, то будет использоваться регистратор, заданный в настройках журналирования (см. *главу 29*).

26.3. Кэширование на стороне клиента

Для управления кэшированием на стороне клиента (в частности, для уменьшения объема пересылаемых по сети данных) Django предоставляет один посредник и несколько декораторов.

26.3.1. Автоматическая обработка заголовков

Посредник `django.middleware.http.ConditionalGetMiddleware` применяется в случае кэширования на стороне сервера и выполняет автоматическую обработку заголовков, управляющих кэшированием на стороне клиента.

Как только клиенту отправляется запрошенная им страница, в состав ответа добавляется заголовок `E-Tag`, хранящий хэш-сумму содержимого страницы. Эти сведения веб-обозреватель сохраняет в своем кэше.

Как только посетитель снова запрашивает загруженную ранее и сохраненную в локальном кэше страницу, веб-обозреватель посылает в составе запроса такие заголовки:

- `If-Match` или `If-None-Match` со значением полученного с ответом заголовка `E-Tag` — если ранее с ответом пришел заголовок `E-Tag`;
- `If-Modified-Since` или `If-Unmodified-Since` с временно́й отметкой последнего изменения страницы, полученной с заголовком `Last-Modified` — если ранее в составе ответа был получен заголовок `Last-Modified` или `Last-Unmodified`.

Получив запрос, посредник сравнивает значения:

- конечного ключа — полученное от клиента в заголовке `If-Match` или `If-None-Match` и находящееся в кэше;
- временно́й отметки последнего изменения страницы — полученное от клиента в заголовке `If-Modified-Since` или `If-Unmodified-Since` и хранящееся в кэше вместе с самой страницей.

Если эти значения равны, то Django предполагает, что кэшированная страница еще актуальна, и отправляет клиенту ответ с кодом статуса 304 (запрошенная страница не изменилась). Веб-обозреватель вместо того, чтобы повторно загружать страницу по сети, извлекает ее из локального кэша, что выполняется гораздо быстрее.

Если же эти значения не равны, значит, страница либо устарела и была удалена из кэша, либо еще не кэшировалась. Тогда веб-обозреватель получит полноценный ответ с кодом статуса 200, содержащий запрошенную страницу.

Посредник `django.middleware.http.ConditionalGetMiddleware` в списке зарегистрированных в проекте (параметр `MIDDLEWARE` настроек сайта) должен помещаться перед посредником `django.middleware.common.CommonMiddleware`. Автор книги обычно помещает его в самом начале списка зарегистрированных посредников:

```
MIDDLEWARE = [
    'django.middleware.http.ConditionalGetMiddleware',
    'django.middleware.security.SecurityMiddleware',
    . . .
]
```

Если кэширование на стороне сервера не задействовано, то посредник `django.middleware.http.ConditionalGetMiddleware` не дает никакого эффекта. Хотя описанные ранее заголовки добавляются в ответ, получаются из запроса и обрабатываются посредником, но, поскольку страница не сохранена в кэше, она при каждом запросе генерируется заново.

26.3.2. Управление кэшированием в контроллерах

26.3.2.1. Условная обработка запросов

Если кэширование на стороне сервера не используется, то для управления кэшированием на стороне клиента следует применять три декоратора из модуля `django.views.decorators.http`.

- `condition([etag_func=None][,][last_modified_func=None])` — выполняет обработку заголовков `E-Tag` и `Last-Modified`.

В параметре `etag_func` указывается ссылка на функцию, которая будет вычислять значение заголовка `E-Tag`. Эта функция должна принимать те же параметры, что и контроллер-функция, у которого указывается этот декоратор, и возвращать в качестве результата значение для упомянутого ранее заголовка, представленное в виде строки.

В параметре `last_modififed_func` указывается ссылка на аналогичную функцию, вычисляющую значение для заголовка `Last-Modified`. Эта функция должна возвращать значение заголовка в виде временно́й отметки (объекта типа `datetime` из модуля `datetime`).

Можно указать либо оба параметра сразу, либо, как поступают наиболее часто, лишь один из них.

Декоратор указывается у контроллера-функции и в дальнейшем управляет кэшированием генерируемой им страницы.

При формировании страницы функции, записанные в вызове декоратора, вычисляют значения заголовков `E-Tag` и (или) `Last-Modified`. Эти заголовки отсылаются клиенту в составе ответа вместе с готовой страницей.

Как только от того же клиента будет получен запрос на ту же страницу, декоратор извлечет из запроса значения заголовков `E-Tag` и (или) `Last-Modified` и сравнит их с величинами, возвращенными функциями, что указаны в его вызове. Если значения `E-Tag` не совпадают или если значение `Last-Modified`, вычисленное функцией, больше значения, полученного в заголовке, то будет выполнен контроллер, который сгенерирует страницу. В противном случае клиенту отправится ответ с кодом статуса 304 (запрошенная страница не изменилась).

Пример:

```
from django.views.decorators.http import condition
from .models import Bb

def bb_lmf(request, pk):
    return Bb.objects.get(pk=pk).published

@condition(last_modified_func=bb_lmf)
def detail(request, pk):
    . . .
```

Декоратор `condition()` можно указать и у контроллера-класса:

```
urlpatterns = [
    . . .
    path('detail/<int:pk>/',
         condition(last_modified_func=bb_lmf)(BbDetailView.as_view())),
    . . .
]
```

- `etag(<функция E-Tag>)` — обрабатывает только заголовок `E-Tag`;
- `last_modified(<функция Last-Modified>)` — обрабатывает только заголовок `Last_Modified`. Пример:

```
from django.views.decorators.http import last_modified

@last_modified(bb_lmf)
def detail(request, pk):
    . . .

urlpatterns = [
    . . .
    path('detail/<int:pk>/',
         last_modified(bb_lmf)(BbDetailView.as_view())),
    . . .
]
```

26.3.2.2. Прямое указание параметров кэширования

Параметры кэширования страницы на стороне клиента записываются в составе значения заголовка `Cache-Control`, который отсылается клиенту в составе ответа, включающего эту страницу. Указать эти параметры напрямую можно, воспользовавшись декоратором `cache_control(`*<параметры>*`)` из модуля `django.views.decorators.cache`. В его вызове указываются именованные *параметры*, которые и зададут настройки кэширования.

Чтобы вставить в заголовок `Cache-Control` ответа параметр, не имеющий значения, следует присвоить соответствующему именованному параметру декоратора значение `True`. Если параметр заголовка включает дефис, то в именованном параметре его следует заменить подчеркиванием.

Примеры:

```
from django.views.decorators.cache import cache_control

# Параметр max-age заголовка Cache-Control ответа указывает время
# кэширования страницы на стороне клиента, в секундах. Чтобы указать
# этот параметр в вызове декоратора cache_control, заменим дефис
# подчеркиванием
@cache_control(max_age=3600)
def detail(request, pk):
    ...

# Значение True, данное параметру private, указывает, что страница
# содержит конфиденциальные данные и может быть сохранена только в кэше
# веб-обозревателя. Промежуточные программы, наподобие прокси-серверов,
# кэшировать ее не будут
@cache_control(private=True)
def account(request, user_pk):
    ...
```

26.3.2.3. Запрет кэширования

Если необходимо запретить кэширование какой-либо страницы на уровне клиента (например, если страница содержит часто обновляющиеся или конфиденциальные

данные), достаточно использовать декоратор `never_cache()` из модуля `django.views.decorators.cache`. Пример:

```
from django.views.decorators.cache import never_cache

@never_cache
def fresh_news(request):
    ...
```

Этот декоратор добавляет в отсылаемый клиенту ответ следующий заголовок:

```
Cache-Control: max-age=0, no-cache, no-store, must-revalidate, private
```

Параметр `private` добавляется, начиная с Django 3.0.

26.3.3. Управление кэшированием в посредниках

Для указания параметров кэширования на стороне клиента в посредниках (о них рассказывалось в *главе 22*) применяются следующие функции, объявленные в модуле `django.utils.cache`:

- `patch_cache_control(<ответ>, <параметры кэширования>)` — непосредственно указывает *параметры кэширования* в заголовке `Cache-Control` заданного *ответа*. *Параметры кэширования* указываются в соответствующих именованных параметрах функции.

  Чтобы вставить в заголовок `Cache-Control` ответа параметр, не имеющий значения, следует присвоить соответствующему именованному параметру функции значение `True`. Если параметр кэширования включает дефис, то в именованном параметре его следует заменить подчеркиванием.

  Пример посредника, добавляющего в ответ заголовок `Cache-Control: max-age=0, no-cache`:

  ```
  from django.utils.cache import patch_cache_control

  def my_cache_control_middleware(next):

      def core_middleware(request):
          response = next(request)
          patch_cache_control(response, max_age=0, no_cache=True)
          return response

      return core_middleware
  ```

- `patch_response_headers(<ответ>[, cache_timeout=None])` — задает время кэширования страницы клиентом в заголовке `Expires` и параметре `max-age` заголовка `Cache-Control` заданного *ответа*. Время кэширования указывается в параметре `cache_timeout`; если оно не задано, то устанавливается время кэширования из параметра `CACHE_MIDDLEWARE_SECONDS` настроек проекта;

- `patch_vary_headers(<ответ>, <заголовки>)` — добавляет в заголовок `Vary` *ответа* заданные во втором параметре *заголовки* запроса. Последние указываются в виде списка или кортежа из строк. Пример:

```
from django.utils.cache import patch_vary_headers

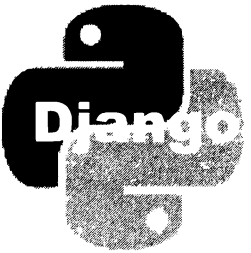
patch_vary_headers(response, ('User-Agent', 'Cookie'))
```

- `add_never_cache_headers(<ответ>)` — добавляет в *ответ* заголовок, запрещающий кэширование страниц на стороне клиента:

```
Cache-Control: max-age=0, no-cache, no-store, must-revalidate, private
```

  Параметр `private` добавляется, начиная с Django 3.0;

- `get_max_age(<ответ>)` — возвращает время кэширования страницы клиентом (в секундах), взятое из параметра `max-age` заголовка `Cache-Control` заданного *ответа*. Если такой параметр отсутствует или его значение не является целым числом, то возвращается `None`.



ГЛАВА 27

Административный веб-сайт Django

Django включает в свой состав полностью готовый к работе административный веб-сайт, предоставляющий доступ к любым внутренним данным, позволяющий пополнять, править и удалять их, гибко настраиваемый и исключительно удобный в использовании.

Доступ к административному сайту Django имеют только суперпользователь и пользователи со статусом персонала.

27.1. Подготовка административного веб-сайта к работе

Прежде чем работать с административным сайтом, необходимо осуществить следующие подготовительные операции:

- открыть модуль настроек проекта settings.py, находящийся в пакете конфигурации, и проверить:
  - присутствуют ли в списке зарегистрированных в проекте приложений (параметр `INSTALLED_APPS`) приложения `django.contrib.admin`, `django.contrib.auth`, `django.contrib.contenttypes`, `django.contrib.messages` и `django.contrib.sessions`;
  - присутствуют ли в списке зарегистрированных посредников (параметр `MIDDLEWARE`) посредники `django.contrib.auth.middleware.AuthenticationMiddleware`, `django.contrib.sessions.middleware.SessionMiddleware` и `django.contrib.messages.middleware.MessageMiddleware`;
  - проверить, присутствуют ли в списке зарегистрированных для используемого шаблонизатора обработчиков контекста (вложенный параметр `context_processors` параметра `OPTIONS`) обработчики `django.contrib.auth.context_processors.auth` и `django.contrib.messages.context_processors.messages`;
- добавить в список маршрутов уровня проекта (модуль urls.py пакета конфигурации) маршрут, который свяжет выбранный шаблонный путь (обычно использу-

ется **admin/**) со списком маршрутов, записанным в атрибуте `urls` объекта административного сайта, который хранится в переменной `site` из модуля `django.contrib.admin`:

```
from django.contrib import admin
. . .
urlpatterns = [
    . . .
    path('admin/', admin.site.urls),
]
```

Впрочем, во вновь созданном проекте все эти действия выполнены изначально;

- выполнить миграции;
- создать суперпользователя (см. *разд. 15.2.2*).

> ***На заметку***
>
> Для своих нужд Django создает в базе данных таблицу `django_admin_log`, хранящую "историю" манипуляций с содержимым базы данных посредством административного сайта.

27.2. Регистрация моделей на административном веб-сайте

Чтобы с данными, хранящимися в определенной модели, можно было работать посредством административного сайта, модель нужно зарегистрировать на сайте. Делается это вызовом метода `register(<модель>)` объекта административного сайта. Пример:

```
from django.contrib import admin
from .models import Bb, Rubric

admin.site.register(Rubric)
```

Код, выполняющий регистрацию моделей на административном сайте, следует помещать в модуль admin.py пакета приложения.

Это самый простой способ сделать модель доступной через административный сайт. Однако в этом случае записи, хранящиеся в модели, будут иметь представление по умолчанию: отображаться в виде их строкового представления (которое формируется методом `__str__()`), выводиться в порядке их добавления, не поддерживать специфических средств поиска и пр.

Если нужно, чтобы список записей представлялся в виде таблицы с колонками, в которых выводятся значения их отдельных полей, а также требуется получить доступ к специфическим средствам поиска и настроить внешний вид страниц добавления и правки записей, то придется создать для этой модели класс редактора.

27.3. Редакторы моделей

Редактор модели — это класс, указывающий параметры представления модели на административном сайте: набор полей, выводящихся на экран, порядок сортировки записей, применение специальных средств для их фильтрации, элементы управления, используемые для занесения значений в поля записей, и пр.

Класс редактора должен быть производным от класса `ModelAdmin` из модуля `django.contrib.admin`. Его объявление, равно как и регистрирующий его код, следует записывать в модуле admin.py пакета приложения. Пример объявления класса-редактора можно увидеть в листинге 1.14.

Различные параметры представления модели записываются в классе редактора в виде его атрибутов или методов.

27.3.1. Параметры списка записей

Страница списка записей выводит перечень записей, хранящихся в модели, позволяет осуществлять их фильтрацию, сортировку, выполнять различные действия над группой выбранных записей (в частности, их удаление) и даже, при указании соответствующих настроек, править записи, не заходя на страницу правки.

27.3.1.1. Параметры списка записей: состав выводимого списка

Параметры из этого раздела задают поля, выводимые в списке записей, и возможность правки записей непосредственно на странице списка:

- `list_display` — атрибут, задает набор выводимых в списке полей. Его значением должен быть список или кортеж, элементом которого может быть:
  - имя поля модели в виде строки:

```
class BbAdmin(admin.ModelAdmin):
    list_display = ('title', 'content', 'price', 'published', 'rubric')
```

  - имя функционального поля, объявленного в модели, в виде строки (о функциональных полях рассказывалось в *разд. 4.6*). Вот пример указания функционального поля `title_and_price` модели `Bb`:

```
class BbAdmin(admin.ModelAdmin):
    list_display = ('title_and_price', 'content', 'published', 'rubric')
```

    Также можно указать имя метода `__str__()`, который формирует строковое представление модели:

```
class RubricAdmin(admin.ModelAdmin):
    list_display = ('__str__', 'order')
```

  - имя функционального поля, объявленного непосредственно в классе редактора, также в виде строки. Такое поле реализуется методом, принимающим в качестве единственного параметра объект записи и возвращающим резуль-

тат, который и будет выведен на экран. Вот пример указания функционального поля title_and_rubric, объявленного в классе редактора BbAdmin:

```
class BbAdmin(admin.ModelAdmin):
    list_display = ('title_and_rubric', 'content', 'price', 'published')

    def title_and_rubric(self, rec):
        return '%s (%s)' % (rec.title, rec.rubric.name)
```

У созданного таким образом функционального поля можно указать название, присвоив его атрибуту short_description объекта метода, реализующего поле:

```
class BbAdmin(admin.ModelAdmin):
    . . .

    def title_and_rubric(self, rec):
        . . .
    title_and_rubric.short_description = 'Название и рубрика'
```

- ссылка на функциональное поле, объявленное в виде обычной функции. Такая функция должна принимать с единственным параметром объект записи и возвращать результат, который и будет выведен на экран. Для этого поля можно задать название, присвоив его атрибуту short_description объекта функции. Вот пример указания функционального поля title_and_rubric, реализованного в виде обычной функции:

```
def title_and_rubric(rec):
    return '%s (%s)' % (rec.title, rec.rubric.name)
title_and_rubric.short_description = 'Название и рубрика'

class BbAdmin(admin.ModelAdmin):
    list_display = [title_and_rubric, 'content', 'price', 'published']
```

Поля типа ManyToManyField не поддерживаются, и их содержимое не будет выводиться на экран.

По значениям функциональных полей невозможно выполнять сортировку, поскольку записи сортируются СУБД, которая не "знает" о существовании функциональных полей. Однако можно связать функциональное поле с обычным полем модели, и тогда щелчок на заголовке функционального поля на странице списка записей приведет к сортировке записей по значениям указанного обычного поля. Для этого достаточно присвоить строку с именем связываемого обычного поля модели атрибуту admin_order_field объекта метода (функции), реализующего функциональное поле. Вот пара примеров:

```
def title_and_rubric(rec):
    . . .
    def title_and_rubric(rec):
        return '%s (%s)' % (rec.title, rec.rubric.name)
```

```
    # Связываем с функциональным полем title_and_rubric
    # обычное поле title модели
    title_and_rubric.admin_order_field = 'title'

    # А теперь связываем с тем же полем поле name
    # связанной модели Rubric
    title_and_rubric.admin_order_field = 'rubric__name'
```

- `get_list_display(self, request)` — метод, должен возвращать набор выводимых в списке полей. Полученный с параметром `request` запрос можно использовать при формировании этого набора.

  В данном примере обычным пользователям показываются только название, описание и цена товара, а суперпользователю — также рубрика и временна́я отметка публикации:

```
class BbAdmin(admin.ModelAdmin):
    def get_list_display(self, request):
        ld = ['title', 'content', 'price']
        if request.user.is_superuser:
            ld += ['published', 'rubric']
        return ld
```

  В изначальной реализации метод возвращает значение атрибута `list_display`;

- `list_display_links` — атрибут, задает перечень полей, значения которых будут превращены в гиперссылки, указывающие на страницы правки соответствующих записей. В качестве значения указывается список или кортеж, элементами которого являются строки с именами полей. Эти поля должны присутствовать в перечне, заданном атрибутом `list_display`. Пример:

```
class BbAdmin(admin.ModelAdmin):
    list_display = ('title', 'content', 'price', 'published', 'rubric')
    list_display_links = ('title', 'content')
```

  Если в качестве значения атрибута задать "пустой" список или кортеж, то в гиперссылку будет превращено значение самого первого поля, выводящегося в списке, а если указать `None`, то такие гиперссылки вообще не будут создаваться. Значение по умолчанию — "пустой" список;

- `get_list_display_links(self, request, list_display)` — метод, должен возвращать перечень полей, значения которых будут превращены в гиперссылки на страницы правки соответствующих записей. Параметром `request` передается запрос, а параметром `list_display` — список с именами выводимых полей, который может быть использован для создания перечня полей-гиперссылок.

  Вот пример преобразования в гиперссылки всех полей, выводимых на экран:

```
class BbAdmin(admin.ModelAdmin):
    def get_list_display_links(self, request, list_display):
        return list_display
```

В изначальной реализации метод возвращает значение атрибута `list_display_links`, если его значение не равно "пустому" списку или кортежу. В противном случае возвращается список, содержащий первое поле из состава выводимых на экран;

- `list_editable` — атрибут, задает перечень полей, которые можно будет править непосредственно на странице списка записей, не переходя на страницу правки. В соответствующих столбцах списка записей будут присутствовать элементы управления, с помощью которых пользователь сможет исправить значения полей, указанных в перечне. После правки необходимо нажать расположенную в нижней части страницы кнопку **Сохранить**, чтобы внесенные правки были сохранены.

  В качестве значения атрибута указывается список или кортеж, элементами которого должны быть имена полей, представленные в виде строк. Если указать "пустой" список или кортеж, то ни одно поле модели не может быть исправлено на странице списка.

  > ***Внимание!***
  > Поля, указанные в перечне из атрибута `list_editable`, *не* должны присутствовать в перечне из атрибута `list_display_links`.

  Пример:

```
class BbAdmin(admin.ModelAdmin):
    list_display = ('title', 'content', 'price', 'published', 'rubric')
    list_display_links = None
    list_editable = ('title', 'content', 'price', 'rubric')
```

  Значение по умолчанию — "пустой" кортеж;

- `list_select_related` — атрибут, устанавливает набор первичных моделей, чьи связанные записи будут извлекаться одновременно с записями текущей модели посредством вызова метода `select_related()` (см. *разд. 16.1*). В качестве значения можно указать:

  - `True` — извлекать связанные записи всех первичных моделей, связанных с текущей моделью;
  - `False` — извлекать связанные записи только тех первичных моделей, что соответствуют полям типа `ForeignKey`, присутствующим в перечне из атрибута `list_display`;
  - список или кортеж, содержащий строки с именами полей типа `ForeignKey`, — извлекать связанные записи только тех первичных моделей, что соответствуют приведенным в этом списке (кортеже) полям;
  - "пустой" список или кортеж — вообще не извлекать связанные записи.

  Извлечение связанных записей первичных моделей одновременно с записями текущей модели повышает производительность, т. к. впоследствии Django не придется для получения связанных записей обращаться к базе еще раз.

  Значение по умолчанию — `False`;

- `get_list_select_related(self, request)` — метод, должен возвращать набор первичных моделей, чьи связанные записи будут извлекаться одновременно с записями текущей модели посредством вызова метода `select_related()`. Полученный с параметром `request` запрос можно использовать при формировании этого набора. В изначальной реализации возвращает значение атрибута `list_select_related`;
- `get_queryset(self, request)` — метод, должен возвращать набор записей, который и станет выводиться на странице списка. Параметр `request` передает запрос. Переопределив этот метод, можно установить какую-либо дополнительную фильтрацию записей, создать вычисляемые поля или изменить сортировку.

  Вот пример переопределения этого метода с целью дать возможность суперпользователю просматривать все объявления, а остальным пользователям — только объявления, не помеченные как скрытые:

```
class BbAdmin(admin.ModelAdmin):
    def get_queryset(self, request):
        qs = super().get_queryset(request)
        if request.user.is_superuser:
            return qs
        else:
            return qs.filter(is_hidden=False)
```

27.3.1.2. Параметры списка записей: фильтрация и сортировка

В этом разделе перечислены параметры, настраивающие средства для фильтрации и сортировки записей:

- `ordering` — атрибут, задает порядок сортировки записей. Значение указывается в том же формате, что и значение параметра модели `ordering` (см. *разд. 4.4*). Если указать значение `None`, то будет использована сортировка, заданная для модели. По умолчанию — `None`;
- `get_ordering(self, request)` — метод, должен возвращать параметры сортировки. Параметр `request` передает запрос. В изначальной реализации возвращает значение атрибута `ordering`;
- `sortable_by` — атрибут, задает перечень полей модели, по которым пользователь сможет выполнять сортировку записей. Перечень полей задается в виде списка, кортежа или множества. Если задать "пустую" последовательность, то пользователь не сможет сортировать записи по своему усмотрению. По умолчанию — `None` (разрешена сортировка записей по всем полям);
- `get_sortable_by(self, request)` — метод, должен возвращать перечень полей модели, по которым пользователь сможет сортировать записи. Параметр `request` передает запрос. В изначальной реализации возвращает значение атрибута `sortable_by`;

- `search_fields` — атрибут, указывает перечень полей модели, по которым будет выполняться фильтрация записей. Значением должен быть список или кортеж имен полей, заданных в виде строк. Пример:

```
class BbAdmin(admin.ModelAdmin):
    search_fields = ('title', 'content')
```

Фильтрация по указанным полям осуществляется занесением в расположенное вверху страницы поле ввода искомого слова или фразы и нажатием кнопки **Найти**. Чтобы отменить фильтрацию и вывести все записи, достаточно очистить поле ввода и снова нажать кнопку **Найти**.

По умолчанию Django выбирает записи, которые содержат все занесенные в поле ввода слова, независимо от того, в каком месте поля встретилось это слово и является оно отдельным словом или частью другого, более длинного. Фильтрация выполняется без учета регистра. Например, если ввести фразу "газ кирпич", будут отобраны записи, которые в указанных полях хранят слова "Газ" и "Кирпич", "керогаз" и "кирпичный" и т. п., но не "бензин" и "дерево".

Чтобы изменить поведение фреймворка при фильтрации, нужно предварить имя поля одним из поддерживаемых префиксов:

- `^` — искомое слово должно присутствовать в начале поля:

```
search_fields = ('^content',)
```

При задании слова "газ" будут отобраны записи со словами "газ", "газовый", но не "керогаз";

- `=` — точное совпадение, т. е. указанное слово должно полностью совпадать со значением поля. Регистр не учитывается. Пример:

```
search_fields = ('=content',)
```

Будут отобраны записи со словами "газ", но не "газовый" или "керогаз";

- `@` — полнотекстовый поиск. Поддерживается только базами данных MySQL.

Значение по умолчанию — "пустой" кортеж;

- `get_search_fields(self, request)` — метод, должен возвращать перечень полей, по которым будет выполняться фильтрация записей. Параметр `request` передает запрос. В изначальной реализации возвращает значение атрибута `search_fields`;

- `show_full_result_count` — атрибут. Если `True`, то после выполнения фильтрации в полях, заданных атрибутом `search_fields`, правее кнопки **Найти** будет выведено количество отфильтрованных записей и общее число записей в модели. Если `False`, то вместо общего числа записей в модели будет выведена гиперссылка **Показать все**. По умолчанию — `True`.

Для вывода общего количества записей Django выполняет дополнительный запрос к базе данных. Поэтому, если стоит задача всемерно уменьшить нагрузку на базу, этому атрибуту имеет смысл задать значение `False`;

- `list_filter` — атрибут, указывает перечень полей, по которым можно будет выполнять быструю фильтрацию. В правой части страницы списка записей появятся списки всех значений, занесенных в заданные поля, и при щелчке на таком значении будут выведены только те записи, у которых указанное поле хранит выбранное значение. Чтобы вновь вывести в списке все записи, следует щелкнуть пункт **Все**.

  Значением поля должен быть список или кортеж, каждый элемент которого представляет собой:

  - имя поля в виде строки:

    ```
    class BbAdmin(admin.ModelAdmin):
        list_display = ['title', 'content', 'price', 'published', 'rubric']
        list_filter = ('title', 'rubric__name',)
    ```

    Как видно из примера, в перечне можно указывать поля связанных моделей;

  - ссылку на подкласс класса `SimpleListFilter` из модуля `django.contrib.admin`, реализующий более сложную фильтрацию. В этом подклассе следует объявить:

    - `title` — атрибут, указывает заголовок, который будет выведен над перечнем доступных для выбора значений, по которым собственно и будет выполняться фильтрация. Значение задается в виде строки;
    - `parameter_name` — атрибут, указывает имя GET-параметра, посредством которого будет пересылаться внутренний идентификатор выбранного пользователем значения для фильтрации. Это имя также должно представлять собой строку;
    - `lookups(self, request, model_admin)` — метод, должен возвращать перечень доступных для выбора значений, по которым будет выполняться фильтрация. Через параметр `request` передается объект запроса, через параметр `model_admin` — объект редактора.

      Метод должен возвращать кортеж, каждый элемент которого задаст отдельное значение для фильтрации. Этот элемент должен представлять собой кортеж из двух строковых элементов: внутреннего идентификатора значения (того самого, что пересылается через GET-параметр, заданный атрибутом `parameter_name`) и названия, выводящегося на экран;
    - `queryset(self, request, queryset)` — метод, вызываемый после щелчка пользователя на одном из значений и возвращающий соответствующим образом отфильтрованный набор записей. Через параметр `request` передается объект запроса, через параметр `queryset` — изначальный набор записей.

    Чтобы получить внутренний идентификатор значения, на котором щелкнул пользователь, нужно обратиться к методу `value()`, унаследованному классом редактора от суперкласса.

    Далее приведен код класса, реализующего фильтрацию объявлений по цене: является она низкой (менее 500 руб.), средней (от 500 до 5000 руб.) или высокой (более 5000 руб.).

```
class PriceListFilter(admin.SimpleListFilter):
    title = 'Категория цен'
    parameter_name = 'price'

    def lookups(self, request, model_admin):
        return (
            ('low', 'Низкая цена'),
            ('medium', 'Средняя цена'),
            ('high', 'Высокая цена'),
        )

    def queryset(self, request, queryset):
        if self.value() == 'low':
            return queryset.filter(price__lt=500)
        elif self.value() == 'medium':
            return queryset.filter(price__gte=500,
                                   price__lte=5000)
        elif self.value() == 'high':
            return queryset.filter(price__gt=5000)
```

Теперь мы можем использовать этот класс для быстрой фильтрации объявлений по категории цен:

```
class BbAdmin(admin.ModelAdmin):
    . . .
    list_filter = (PriceListFilter,)
```

Если в качестве значения атрибута указать "пустой" список или кортеж, то быстрая сортировка будет отключена. Значение по умолчанию — "пустой" кортеж;

- `get_list_filter(self, request)` — метод, должен возвращать перечень полей, по которым будет выполняться быстрая фильтрация. Параметр `request` передает запрос. В изначальной реализации возвращает значение атрибута `list_filter`;
- `date_hierarchy` — атрибут, включающий быструю фильтрацию по датам или временным отметкам. Если указать в качестве его значения строку с именем поля типа `DateField` или `DateTimeField`, то над списком записей будет выведен набор значений, хранящихся в этом поле, в виде гиперссылок. После щелчка на такой гиперссылке в списке будут выведены только те записи, в указанном поле которых хранится выбранное значение.

  Пример:

```
class BbAdmin(admin.ModelAdmin):
    . . .
    date_hierarchy = 'published'
```

  Значение по умолчанию — `None` (быстрая фильтрация по дате не выполняется);
- `preserve_filters` — атрибут. Если `True`, то после сохранения добавленной или исправленной записи все заданные пользователем условия фильтрации про-

должают действовать. Если False, то фильтрация записей в списке после этого будет отменена. По умолчанию — True.

27.3.1.3. Параметры списка записей: прочие

Описанные здесь параметры затрагивают по большей части внешний вид списка записей:

- list_per_page — атрибут, задает количество записей в части пагинатора, применяемого при выводе списка записей, в виде целого числа (по умолчанию: 100);
- list_max_show_all — атрибут, задает количество записей, выводимых после щелчка на гиперссылке **Показать все** (она находится под списком), если общее количество записей меньше или равно количеству, заданному этим атрибутом. Значение атрибута указывается в виде целого числа. По умолчанию: 200;
- actions_on_top — атрибут. Если True, то над списком записей будет выведен раскрывающийся список всех действий, доступных для выполнения над группой выбранных записей. Если False, то список действий над списком записей выведен не будет. По умолчанию — True;
- actions_on_bottom — атрибут. Если True, то под списком записей будет выведен раскрывающийся список всех действий, доступных для выполнения над группой выбранных записей. Если False, то список действий под списком записей выведен не будет. По умолчанию — False;
- actions_selection_counter — атрибут. Если True, то возле раскрывающегося списка действий будут выведены количество выбранных записей и общее количество записей на текущей странице списка. Если False, то эти сведения не будут выводиться. По умолчанию — True;
- empty_value_display — атрибут, указывает символ или строку, которая будет выводиться вместо пустого значения поля (таким считается "пустая" строка и None). По умолчанию: "-" (дефис). Пример:

```
class BbAdmin(admin.ModelAdmin):
    . . .
    empty_value_display = '---'
```

Если в редакторе объявлено функциональное поле, то можно указать подобного рода значение только для этого поля, присвоив его атрибуту empty_value_display объекта метода, который реализует это поле:

```
class RubricAdmin(admin.ModelAdmin):
    fields = ('name', 'super_rubric')

    def super_rubric(self, rec):
        return rec.super_rubric.name
    super_rubric.empty_value_display = '[нет]'
```

Наконец, можно указать "замещающее" значение для всего административного сайта:

```
admin.site.empty_value_display = '(пусто)'
```

- `paginator` — атрибут, задает ссылку на класс пагинатора, используемого для вывода списка записей. По умолчанию — ссылка на класс `Paginator` из модуля `django.core.paginator`;
- `get_paginator()` — метод, должен возвращать объект пагинатора, используемого для вывода списка записей. Формат объявления:

```
get_paginator(self, request, queryset, per_page, orphans=0,
                                       allow_empty_first_page=True)
```

Параметр `request` передает запрос, параметр `queryset` — набор записей, который должен разбиваться на части, параметр `per_page` — количество записей в части. Параметры `orphans` и `allow_empty_first_page` описаны в *разд. 12.1*.

В реализации по умолчанию возвращает экземпляр класса пагинатора, заданного в атрибуте `paginator`.

27.3.2. Параметры страниц добавления и правки записей

Эти страницы содержат формы для добавления и правки записей и также довольно гибко настраиваются.

27.3.2.1. Параметры страниц добавления и правки записей: набор выводимых полей

Эти параметры указывают набор выводимых в форме полей модели, перечни наборов полей (будут описаны далее) и некоторые другие параметры:

- `fields` — атрибут, задает последовательность (список или кортеж) имен полей, выводимых в форме (не указанные поля выведены не будут). Поля будут выведены в том порядке, в котором они перечислены в последовательности.

Пример вывода в форме объявления только полей названия, цены и описания (именно в таком порядке):

```
class BbAdmin(admin.ModelAdmin):
    fields = ('title', 'price', 'content')
```

В наборе можно указать поля, доступные для чтения, которые указаны в атрибуте `readonly_fields` (он будет описан далее).

По умолчанию поля в форме выводятся по вертикали сверху вниз. Чтобы вывести какие-либо поля в одной строчке по горизонтали, нужно заключить их имена во вложенный кортеж. Пример вывода полей названия и цены в одной строчке:

```
class BbAdmin(admin.ModelAdmin):
    fields = (('title', 'price'), 'content')
```

Если указать для атрибута значение `None`, то в форме будут выведены все поля, кроме имеющих типы `AutoField`, `SmallAutoField`, `BigAutoField` и тех, у которых

в параметре `editable` конструктора класса было явно или неявно задано значение `True`.

Значение по умолчанию — `None`;

- `get_fields(self, request, obj=None)` — метод, должен возвращать последовательность имен полей, которые следует вывести в форме. В параметре `request` передается запрос, а в необязательном параметре `obj` может быть получен объект исправляемой записи или `None`, если запись в данный момент добавляется.

  Для примера сделаем так, чтобы у создаваемого объявления можно было указать рубрику, а у исправляемого — уже нет:

```
class BbAdmin(admin.ModelAdmin):
    def get_fields(self, request, obj=None):
        f = ['title', 'content', 'price']
        if not obj:
            f.append('rubric')
        return f
```

  В изначальной реализации метод возвращает значение атрибута `fields`, если его значение отлично от `None`. В противном случае возвращается список, включающий все поля модели и все поля, доступные только для чтения (указываются в атрибуте `readonly_fields`, описанном далее);

- `exclude` — атрибут, задает последовательность имен полей, наоборот, *не* выводимых в форме (не указанные поля будут выведены). Пример вывода в форме объявления всех полей, кроме рубрики и типа объявления:

```
class BbAdmin(admin.ModelAdmin):
    exclude = ('rubric', 'kind')
```

  Если задать значение `None`, то ни одно поле модели не будет исключено из числа выводимых в форме. Значение по умолчанию — `None`;

- `get_exclude(self, request, obj=None)` — метод, должен возвращать последовательность имен полей модели, *не* выводимых в форме. В параметре `request` передается запрос, а в необязательном параметре `obj` может быть получен объект исправляемой записи или `None`, если запись в данный момент добавляется. В изначальной реализации возвращает значение атрибута `exclude`;

  > ***Внимание!***
  >
  > **Следует задавать либо перечень выводимых в форме полей, либо перечень *не* выводимых полей. Если указать их оба, возникнет ошибка.**

- `readonly_fields` — атрибут, задает последовательность имен полей, которые должны быть доступны только для чтения. Значения таких полей будут выведены в виде обычного текста. Пример:

```
class BbAdmin(admin.ModelAdmin):
    fields = ('title', 'content', 'price', 'published')
    readonly_fields = ('published',)
```

Указание в списке атрибута `readonly_fields` — единственный способ вывести на странице правки записи значение поля, у которого при создании параметр `editable` конструктора был установлен в `False`.

Также в этом атрибуте можно указать функциональные поля, объявленные в классе редактора.

Значение по умолчанию — "пустой" кортеж;

- `get_readonly_fields(self, request, obj=None)` — метод, должен возвращать перечень полей, которые должны быть доступны только для чтения. В параметре `request` передается запрос, а в необязательном параметре `obj` может быть получен объект исправляемой записи или `None`, если запись в данный момент добавляется. В изначальной реализации возвращает значение атрибута `readonly_fields`;
- `inlines` — атрибут, задает список встроенных редакторов, присутствующих в текущем редакторе (будут рассмотрены позже). По умолчанию — "пустой" список;
- `fieldsets` — атрибут, задает перечень наборов полей, которые будут созданы в форме.

  Набор полей, как и следует из его названия, объединяет указанные поля формы. Он может вообще никак не выделяться на экране (*основной набор полей*), а может представляться в виде спойлера, изначально свернутого или развернутого.

  Перечень наборов полей записывается в виде кортежа, в котором каждый элемент представляет один набор и также должен являться кортежем из двух элементов: заголовка набора полей (если задать `None`, будет создан основной набор полей) и словаря со следующими дополнительными параметрами:

  - `fields` — кортеж из строк с именами полей модели, которые должны выводиться в наборе. Этот параметр обязателен для указания.

    Здесь можно указать также доступные для чтения поля, заданные в атрибуте `readonly_fields`, в том числе и функциональные поля.

    По умолчанию поля в наборе выводятся по вертикали сверху вниз. Чтобы вывести какие-либо поля в одной строчке по горизонтали, нужно заключить их имена во вложенный кортеж;

  - `classes` — список или кортеж с именами стилевых классов, которые будут привязаны к набору, представленными строками. Поддерживаются два стилевых класса: `collapse` (набор полей, представленный в виде спойлера, будет изначально свернут) и `wide` (поля в наборе займут все пространство окна по ширине);

  - `description` — строка с поясняющим текстом, который будет выведен вверху набора форм, непосредственно под его названием. Вместо обычного текста можно указать HTML-код.

Пример указания набора полей:

```
class BbAdmin(admin.ModelAdmin):
    fieldsets = (
        (None, {
            'fields': (('title', 'rubric'), 'content'),
            'classes': ('wide',),
        }),
        ('Дополнительные сведения', {
            'fields': ('price',),
            'description': 'Параметры, необязательные для указания.',
        })
    )
```

- `get_fieldsets(self, request, obj=None)` — метод, должен возвращать перечень наборов полей, которые будут выведены в форме. В параметре `request` передается запрос, а в необязательном параметре `obj` может быть получен объект исправляемой записи или `None`, если запись в данный момент добавляется.

  В изначальной реализации возвращает значение атрибута `fieldsets`, если его значение отлично от "пустого" словаря. В противном случае возвращается перечень из одного основного набора, содержащего все имеющиеся в форме поля;

- `form` — атрибут, задает класс связанной с моделью формы, на основе которой будет создана окончательная форма, применяемая для работы с записью. Значение по умолчанию — ссылка на класс `ModelForm`;

- `get_form(self, request, obj=None, **kwargs)` — метод, должен возвращать класс формы, которая будет использоваться для занесения данных в создаваемую или исправляемую запись.

  В параметре `request` передается запрос, в необязательном параметре `obj` может быть получен объект исправляемой записи или `None`, если запись в данный момент добавляется. Параметр `kwargs` хранит словарь, элементы которого будут переданы функции `modelform_factory()`, применяемой методом для создания класса формы, в качестве дополнительных параметров.

  Вот пример использования для добавления записи формы `BbAddModelForm`, а для правки существующей записи — формы `BbModelForm`:

```
class BbAdmin(admin.ModelAdmin):
    def get_form(self, request, obj=None, **kwargs):
        if obj:
            return BbModelForm
        else:
            return BbAddModelForm
```

  В изначальной реализации метод возвращает класс формы, сгенерированный вызовом функции `modelform_factory()`.

27.3.2.2. Параметры страниц добавления и правки записей: элементы управления

Эти параметры настраивают внешнее представление отдельных полей в форме, в частности, задают для них элементы управления:

- `radio_fields` — атрибут, задает перечень полей типа `ForeignKey` и полей со списком, для отображения которых вместо раскрывающегося списка будет использован набор переключателей. В качестве значения задается словарь, ключами элементов которого должны быть имена полей, а значениями — одна из переменных, объявленных в модуле `django.contrib.admin`: `HORIZONTAL` (переключатели в наборе располагаются по горизонтали) или `VERTICAL` (по вертикали). Пример:

```
class BbAdmin(admin.ModelAdmin):
    radio_fields = {'rubric': admin.VERTICAL}
```

  Значение по умолчанию — "пустой" словарь;

- `autocomplete_fields` — атрибут, задает перечень полей типов `ForeignKey` и `ManyToManyField`, которые должны отображаться на экране в виде списка с возможностью поиска. Такой список будет включать в свой состав поле для указания текста искомого пункта или его части. Значение атрибута указывается в виде списка или кортежа, включающего строки с именами полей.

  > ***Внимание!***
  > **Чтобы список с возможностью поиска успешно работал, необходимо в классе редактора, представляющем связанную модель, задать перечень полей, по которым можно выполнять поиск записей (атрибут `search_fields`).**

  Пример:

```
class RubricAdmin(admin.ModelAdmin):
    search_fields = ('name',)

class BbAdmin(admin.ModelAdmin):
    autocomplete_fields = ('rubric',)
```

  Значение по умолчанию — "пустой" кортеж;

- `get_autocomplete_fields(self, request)` — метод, должен возвращать перечень полей, отображаемых в виде списка с возможностью поиска. В параметре `request` передается запрос. В изначальной реализации возвращает значение атрибута `autocomplete_fields`;

- `filter_horizontal` — атрибут, указывает кортеж строк с именами полей типа `ManyToManyField`, которые должны быть отображены в виде пары расположенных по горизонтали списков с возможностью поиска пунктов.

  Пример такого элемента управления, надо сказать, исключительно удобного в использовании, можно увидеть на рис. 15.1. В левом списке выводятся только записи ведомой модели, не связанные с текущей записью ведущей модели,

а в правом — только записи, связанные с ней. Связывание записей ведомой модели с текущей записью ведущей модели выполняется переносом их из левого списка в правый щелчком на кнопке со стрелкой, направленной вправо. Аналогично удаление записей из числа связанных с текущей записью производится переносом из правого списка в левый, для чего нужно щелкнуть на кнопке со стрелкой влево.

Вот пример вывода поля `spares` модели `Machine` (см. листинг 4.2) в виде подобного рода элемента управления:

```
class MachineAdmin(admin.ModelAdmin):
    filter_horizontal = ('spares',)
```

Поля типа `ManyToManyField`, не указанные в этом атрибуте, будут выводиться в виде обычного списка с возможностью выбора произвольного количества пунктов. Такой элемент управления не очень удобен в использовании, особенно если записей в ведомой модели достаточно много.

Значение по умолчанию — "пустой" кортеж;

- `filter_vertical` — то же самое, что `filter_horizontal`, только списки выводятся не по горизонтали, а по вертикали, друг над другом: сверху — список несвязанных записей, а под ним — список связанных записей;
- `formfield_overrides` — атрибут, позволяет переопределить параметры полей формы, которая будет выводиться на страницах добавления и правки. В качестве значения указывается словарь, ключами элементов которого выступают ссылки на классы полей модели, а значения указывают параметры соответствующих им полей формы и также записываются в виде словарей.

  Чаще всего этот атрибут применяется для задания других элементов управления для полей формы. Вот пример указания для поля типа `ForeignKey` в качестве элемента управления обычного списка вместо применяемого по умолчанию раскрывающегося:

```
from django import forms
from django.db import models

class BbAdmin(admin.ModelAdmin):
    formfield_overrides = {
        models.ForeignKey: {'widget': forms.widgets.Select(
                                       attrs={'size': 8})},
    }
```

  Значение по умолчанию — "пустой" словарь;

- `prepopulated_fields` — атрибут, устанавливает набор полей, значения которых должны формироваться на основе значений из других полей. В качестве значения указывается словарь, ключи элементов которого должны соответствовать полям, значения которых будут формироваться описанным ранее образом, а значениями станут кортежи имен полей, откуда будут браться данные для формирования значений.

Основное назначение этого атрибута — указание сведений для формирования слагов. Обычно слаг создается из названия какой-либо позиции путем преобразования букв кириллицы в символы латиницы, удаления знаков препинания и замены пробелов на дефисы. Также можно формировать слаги на основе нескольких значений (например, названия и рубрики) — в этом случае отдельные значения объединяются.

Пример:

```
class BbAdmin(admin.ModelAdmin):
    prepopulated_fields = {"slug": ("title",)}
```

Значение по умолчанию — "пустой" словарь;

- `get_prepopulated_fields(self, request, obj=None)` — метод, должен возвращать набор полей, значения которых должны формироваться на основе значений из других полей. В параметре `request` передается запрос, в параметре `obj` — объект исправляемой записи или `None`, если запись в данный момент добавляется. В изначальной реализации возвращает значение атрибута `prepopulated_fields`;
- `raw_id_fields` — атрибут, задает перечень полей типа `ForeignKey` или `ManyToManyField`, для отображения которых вместо списка нужно использовать обычное поле ввода. В такое поле вводится ключ связанной записи или сразу несколько ключей через запятую. Значение атрибута указывается в виде списка или кортежа, в котором приводятся строки с именами полей. Значение по умолчанию — "пустой" кортеж.

27.3.2.3. Параметры страниц добавления и правки записей: прочие

Эти, весьма немногочисленные, параметры управляют внешним видом и поведением страниц добавления и правки записей:

- `view_on_site` — атрибут, указывает, будет ли на странице правки записи выводиться гиперссылка **Смотреть на сайте**, при щелчке на которой осуществляется переход по интернет-адресу модели (см. *разд. 4.5*). В качестве значения атрибута можно указать:
  - `True` — выводить гиперссылку, если в модели объявлен метод `get_absolute_url()`, формирующий интернет-адрес модели. Если такого метода нет, то гиперссылка выводиться не будет;
  - `False` — не выводить гиперссылку в любом случае.

  Значение по умолчанию — `True`.

  Аналогичного результата можно достичь, объявив непосредственно в классе редактора метод `view_on_site()`, который в качестве единственного параметра будет получать объект записи и возвращать строку с интернет-адресом модели. Вот пример:

```
from django.urls import reverse

class BbAdmin(admin.ModelAdmin):
    def view_on_site(self, rec):
        return reverse('bboard:detail', kwargs={'pk': rec.pk})
```

- `save_as` — атрибут. Если `True`, то на странице будет выведена кнопка **Сохранить как новый объект**, если `False`, то вместо нее будет присутствовать кнопка **Сохранить и добавить другой объект**. По умолчанию — `False`;
- `save_as_continue` — атрибут. Принимается во внимание только в том случае, если для атрибута `save_as` задано значение `True`. Если `True`, то после сохранения новой записи выполняется перенаправление на страницу правки той же записи, если `False` — возврат на страницу списка записей. По умолчанию — `True`;
- `save_on_top` — атрибут. Если `True`, кнопки сохранения записи будут присутствовать и вверху, и внизу страницы, если `False` — только внизу. По умолчанию — `False`.

> ***На заметку***
>
> Помимо рассмотренных здесь, редакторы Django поддерживают ряд более развитых инструментов: обработку сохранения записей и наборов записей, удаления записей, указание элементов управления для полей в зависимости от различных условий (например, является ли текущий пользователь суперпользователем), замену шаблонов страниц и др. Эти инструменты достаточно сложны в использовании и применяются относительно редко, поэтому не рассматриваются в этой книге. За инструкциями по их применению обращайтесь на страницу **https://docs.djangoproject.com/en/3.0/ref/contrib/admin/**.

27.3.3. Регистрация редакторов на административном веб-сайте

Чтобы административный сайт смог использовать в работе редактор, последний должен быть соответствующим образом зарегистрирован. Сделать это можно двумя способами:

- применить расширенный формат вызова метода `register()` объекта административного сайта:

```
register(<модель>, <редактор>)
```

Пример:

```
from django.contrib import admin
from .models import Bb, Rubric

class BbAdmin(admin.ModelAdmin):
    . . .

admin.site.register(Bb, BbAdmin)
```

- применить декоратор `register()`, объявленный в модуле `django.contrib.admin`, формат вызова которого следующий:

  `register(<модель 1>, <модель 2> . . . <модель n>)`

  Этот декоратор указывается непосредственно у объявления класса редактора.

  Пример:

```
from django.contrib import admin
from .models import Bb, Rubric

@admin.register(Bb)
class BbAdmin(admin.ModelAdmin):
    . . .
```

  Применив декоратор, один и тот же класс редактора можно указать сразу для нескольких моделей:

```
@admin.register(Bb, Rubric, Machine, Spare)
class UniversalAdmin(admin.ModelAdmin)
    . . .
```

27.4. Встроенные редакторы

Встроенный редактор по назначению аналогичен встроенному набору форм (см. *разд. 14.5*). Он создает на странице добавления или правки записи первичной модели набор форм для работы со связанными записями вторичной модели.

27.4.1. Объявление встроенного редактора

Класс встроенного редактора должен быть производным от одного из следующих классов, объявленных в модуле `django.contrib.admin`:

- `StackedInline` — элементы управления располагаются по вертикали;
- `TabularInline` — элементы управления располагаются по горизонтали. Для формирования набора форм применяется таблица HTML.

В объявлении встроенного редактора указывается модель, которую он должен обслуживать. После объявления он связывается с классом основного редактора.

В листинге 27.1 приведен код, объявляющий и регистрирующий редактор `RubricAdmin`, который предназначен для работы с моделью рубрик `Rubric`. Этот редактор связан со встроенным редактором `BbInline`, обслуживающим модель объявлений `Bb` и выводящим объявления из текущей рубрики.

Листинг 27.1. Пример использования встроенного редактора

```
from django.contrib import admin
from .models import Bb, Rubric
```

```
class BbInline(admin.StackedInline):
    model = Bb

class RubricAdmin(admin.ModelAdmin):
    inlines = [BbInline]

admin.site.register(Rubric, RubricAdmin)
```

27.4.2. Параметры встроенного редактора

Оба класса встроенных редакторов наследуют от класса `ModelAdmin` атрибуты `ordering`, `fields`, `exclude`, `fieldsets`, `radio_fields`, `filter_horizontal`, `filter_vertical`, `formfield_overrides`, `readonly_fields`, `prepopulated_fields`, `raw_id_fields` и методы `get_ordering()`, `get_queryset()`, `get_fields()`, `get_exclude()`, `get_fieldsets()`, `get_readonly_fields()`, `get_prepopulated_fields()`, описанные ранее.

Кроме того, встроенные редакторы поддерживают следующие дополнительные атрибуты и методы:

- `model` — атрибут, указывает ссылку на класс вторичной модели, которая будет обслуживаться встроенным редактором. Единственный обязательный для указания атрибут. По умолчанию — `None`;
- `fk_name` — атрибут, задает имя поля внешнего ключа у вторичной модели в виде строки. Обязателен для указания, если вторичная модель связана с разными первичными моделями и, соответственно, включает несколько полей внешнего ключа. Если задать значение `None`, то Django использует самое первое из объявленных в модели поле внешнего ключа. По умолчанию — `None`;
- `extra` — атрибут, указывает количество пустых форм, предназначенных для создания новых записей, которые будут присутствовать в редакторе. Количество форм должно быть задано в виде целого числа. По умолчанию: `3`;
- `get_extra(self, request, obj=None, **kwargs)` — метод, должен возвращать количество пустых форм, предназначенных для создания новых записей.

  В параметре `request` передается запрос, в параметре `obj` — объект исправляемой записи первичной модели или `None`, если запись еще не создана. Назначение параметра `kwargs` автором не установлено — вероятно, он оставлен на будущее.

  Вот пример указания десяти "пустых" форм при создании записи первичной модели и трех при ее правке:

```
class BbInline(admin.StackedInline):
    model = Bb

    def get_extra(self, request, obj=None, **kwargs):
        if obj:
            return 3
        else:
            return 10
```

  В изначальной реализации метод возвращает значение атрибута `extra`;

- `can_delete` — атрибут. Если `True`, то редактор разрешит пользователю удалять записи, если `False` — не разрешит. По умолчанию — `True`;
- `show_change_link` — атрибут. Если `True`, то в каждой из форм встроенного редактора будет выведена гиперссылка **Изменить**, ведущая на страницу правки соответствующей записи. Если `False`, то такая гиперссылка выводиться не будет. По умолчанию — `False`;
- `min_num` — атрибут, указывает минимальное допустимое количество форм в редакторе в виде целого числа. Если задать `None`, редактор может включать сколько угодно форм. По умолчанию — `None`;
- `get_min_num(self, request, obj=None, **kwargs)` — метод, должен возвращать минимальное допустимое количество форм в редакторе. В параметре `request` передается запрос, в параметре `obj` — объект исправляемой записи первичной модели или `None`, если запись еще не создана. Назначение параметра `kwargs` автором не установлено — вероятно, он оставлен на будущее. В изначальной реализации метод возвращает значение атрибута `min_num`;
- `max_num` — атрибут, указывает максимальное допустимое количество форм в редакторе в виде целого числа. Если задать `None`, то редактор может включать сколько угодно форм. По умолчанию — `None`;
- `get_max_num(self, request, obj=None, **kwargs)` — метод, должен возвращать максимальное допустимое количество форм в редакторе. В параметре `request` передается запрос, в параметре `obj` — объект исправляемой записи первичной модели или `None`, если запись еще не создана. Назначение параметра `kwargs` автором не установлено — вероятно, он оставлен на будущее. В изначальной реализации метод возвращает значение атрибута `max_num`;
- `classes` — атрибут, задает набор стилевых классов, которые будут привязаны к встроенному редактору. Значение указывается в виде списка или кортежа, содержащего строки с именами стилевых классов. Поддерживаются два стилевых класса: `collapse` (редактор, представленный в виде спойлера, будет изначально свернут) и `wide` (редактор займет все пространство окна по ширине). По умолчанию — `None`;
- `verbose_name` — атрибут, задает название сущности, хранящейся в записи вторичной модели, в виде строки. Если задать значение `None`, то будет использовано название, записанное в одноименном параметре обслуживаемой редактором модели. По умолчанию — `None`;
- `verbose_name_plural` — атрибут, задает название набора сущностей, хранящихся во вторичной модели, в виде строки. Если задать значение `None`, то будет использовано название, записанное в одноименном параметре обслуживаемой редактором модели. По умолчанию — `None`;
- `formset` — атрибут, указывает набор форм, связанный с моделью, на основе которого будет создан окончательный набор форм, выводимый на страницу. По умолчанию — ссылка на класс `BaseInlineFormSet`;

- `get_formset(self, request, obj=None, **kwargs)` — метод, должен возвращать класс встроенного набора форм, который будет использован в редакторе.

  В параметре `request` передается запрос, в параметре `obj` — объект исправляемой записи первичной модели или `None`, если запись еще не создана. Параметр `kwargs` хранит словарь, элементы которого будут переданы функции `inlineformset_factory()`, создающей класс набора форм, в качестве параметров.

  В изначальной реализации метод возвращает класс встроенного набора форм, сгенерированный вызовом функции `inlineformset_factory()`;

- `form` — атрибут, задает класс связанной с моделью формы, которая будет применяться в создаваемом наборе форм. По умолчанию — ссылка на класс `ModelForm`.

27.4.3. Регистрация встроенного редактора

Для регистрации встроенных редакторов в редакторе, обслуживающем первичную модель, могут быть использованы:

- `inlines` — атрибут, задает список или кортеж со ссылками на классы встроенных редакторов, регистрируемых в текущем редакторе. По умолчанию — "пустой" список.

  Регистрация встроенного редактора `BbInline`, обслуживающего вторичную модель `Bb`, в редакторе `RubricAdmin` первичной модели:

```
class RubricAdmin(admin.ModelAdmin):
    . . .
    inlines = (BbInline,)
```

- `get_inlines(self, request, obj)` (начиная с Django 3.0) — метод, должен возвращать последовательность ссылок на классы регистрируемых встроенных редакторов. В параметре `request` передается запрос, в параметре `obj` — объект исправляемой записи первичной модели или `None`, если такой записи еще нет.

  Пример вывода набора форм для ввода объявлений только на странице добавления рубрики (на странице правки рубрики он выводиться не будет):

```
class RubricAdmin(admin.ModelAdmin):
    . . .
    def get_inlines(self, request, obj=None):
        if obj:
            return ()
        else:
            return (BbInline,)
```

  В изначальной реализации метод возвращает значение атрибута `inlines`.

27.5. Действия

Действие в терминологии административного сайта Django — это операция, выбираемая из раскрывающегося списка **Действие** и выполняемая применительно к выбранным в списке записям. Список **Действие** находится над перечнем записей, а увидеть его можно на рис. 1.8 и 1.10.

Изначально в этом списке присутствует лишь действие **Удалить выбранные *<название набора сущностей>***, однако можно добавить туда свои собственные действия.

Сначала необходимо объявить функцию, которая, собственно, и реализует действие. В качестве параметров она должна принимать:

- экземпляр класса редактора, к которому будет привязано действие;
- запрос;
- набор записей, содержащий записи, которые были выбраны пользователем.

Никакого результата она возвращать не должна.

Далее нужно задать для действия название, которое будет выводиться в списке **Действие**. Строка с названием присваивается атрибуту `short_description` объекта функции, реализующей это действие.

По завершении выполнения действия, равно как и при возникновении ошибки, рекомендуется вывести всплывающее сообщение (см. *разд. 23.3*). Делается это вызовом метода `message_user()` класса `ModelAdmin`:

```
message_user(<запрос>, <текст сообщения>[, level=messages.INFO][,
                    extra_tags=''][, fail_silently=False])
```

Запрос представляется экземпляром класса `HttpRequest`, *текст сообщения* — строкой. Необязательный параметр `level` указывает уровень сообщения.

Необязательный параметр `extra_tags` задает перечень дополнительных стилевых классов, привязываемых к HTML-тегу с текстом выводимого всплывающего сообщения. Параметру присваивается строка со стилевыми классами, разделенными пробелами.

Если присвоить необязательному параметру `fail_silently` значение `True`, то в случае невозможности создания нового всплывающего сообщения (например, если соответствующая подсистема отключена) ничего не произойдет. Используемое по умолчанию значение `False` указывает в таком случае возбудить исключение `MessageFailure` из модуля `django.contrib.messages`.

В листинге 27.2 приведен код функции `discount()`, реализующей действие, которое уменьшит цены в выбранных объявлениях вдвое и уведомит о завершении всплывающим сообщением.

Листинг 27.2. Пример создания действия в виде функции

```
from django.db.models import F

def discount(modeladmin, request, queryset):
    f = F('price')
    for rec in queryset:
        rec.price = f / 2
        rec.save()
    modeladmin.message_user(request, 'Действие выполнено')
discount.short_description = 'Уменьшить цену вдвое'
```

Для регистрации действий в редакторе предусмотрен атрибут `actions`, поддерживаемый классом `ModelAdmin`. Атрибуту присваивается список или кортеж, содержащий ссылки на функции, что реализуют регистрируемые в редакторе действия. Значение атрибута `actions` по умолчанию — "пустой" список.

Вот так в редакторе `BbAdmin` регистрируется действие `discount()` (см. листинг 27.2):

```
class BbAdmin(admin.ModelAdmin):
    . . .
    actions = (discount,)
```

Действие можно реализовать в виде метода того же класса редактора, в котором оно будет зарегистрировано. Имя метода, реализующего действие, в списке атрибута `actions` следует указать в виде строки.

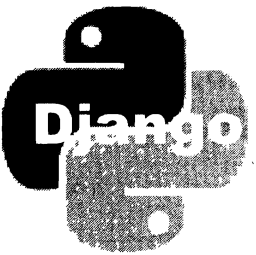
Пример действия, реализованного в виде метода `discount()` редактора `BbAdmin`:

```
class BbAdmin(admin.ModelAdmin):
    . . .
    actions = ('discount',)

    def discount(self, request, queryset):
        f = F('price')
        for rec in queryset:
            rec.price = f / 2
            rec.save()
        self.message_user(request, 'Действие выполнено')
    discount.short_description = 'Уменьшить цену вдвое'
```

На заметку

Django поддерживает создание более сложных действий, выводящих какие-либо промежуточные страницы (наподобие страницы подтверждения), доступных во всех редакторах, а также временную деактивацию действий. Соответствующие руководства находятся здесь: https://docs.djangoproject.com/en/3.0/ref/contrib/admin/actions/.



ГЛАВА 28

Разработка веб-служб REST. Библиотека Django REST framework

Многие современные веб-сайты предоставляют программные интерфейсы, предназначенные для использования сторонними программами: настольными, мобильными приложениями и другими веб-сайтами. С помощью таких интерфейсов, называемых *веб-службами*, сторонние программы могут получать информацию или, наоборот, заносить ее на сайт.

Интерфейсы подобного рода строятся согласно принципам *REST* (Representational State Transfer, репрезентативная передача состояния). К числу этих принципов относится, в частности, идентификация запрашиваемых интернет-ресурсов посредством обычных интернет-адресов. Так, для получения перечня рубрик сторонней программе нужно обратиться по интернет-адресу, скажем, **http://www.bboard.ru/api/rubrics/**.

Совокупность веб-служб (*бэкенд*) отсылает сторонним программам данные, закодированные в каком-либо компактном формате, обычно JSON. Сторонние клиентские программы (*фронтенды*) получают эти данные и обрабатывают нужным им образом, например, выводят на экран.

Веб-службу можно реализовать исключительно средствами Django. Еще в *разд. 9.8.3* описывался класс `JsonResponse`, отправляющий клиенту данные в формате JSON. Однако для этих целей удобнее применять библиотеку Django REST framework. Она самостоятельно извлечет данные из базы, закодирует в JSON, отправит фронтенду, получит данные от фронтенда, проведет их валидацию, занесет в базу и даже реализует разграничение доступа.

> ***Внимание!***
>
> Django REST framework — это полноценный фреймворк, базирующийся на Django. Этому фреймворку впору посвящать отдельную книгу. Здесь же будут описаны лишь его основные возможности и способы применения.
>
> Полная документация по Django REST framework находится по интернет-адресу: **http://www.django-rest-framework.org/**.

28.1. Установка и подготовка к работе Django REST framework

Установка этой библиотеки выполняется подачей команды:

```
pip install djangorestframework
```

Изначально Django обрабатывает лишь запросы, пришедшие с того же домена, на котором располагается веб-служба. Чтобы разрешить фреймворку обрабатывать запросы с других доменов, понадобится дополнительная библиотека django-cors-headers. Установить ее можно подачей команды:

```
pip install django-cors-headers
```

Программными ядрами библиотек Django REST framework и django-cors-headers являются приложения `rest_framework` и `corsheaders` соответственно. Их необходимо добавить в список зарегистрированных в проекте (параметр `INSTALLED_APPS` модуля settings.py из пакета конфигурации):

```
INSTALLED_APPS = [
    . . .
    'rest_framework',
    'corsheaders',
]
```

Кроме того, в список зарегистрированных в проекте (параметр `MIDDLEWARE`) нужно добавить посредник `corsheaders.middleware.CorsMiddleware`, расположив его перед посредником `django.middleware.common.CommonMiddleware`:

```
MIDDLEWARE = [
    . . .
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
    . . .
]
```

Библиотека Django REST framework для успешной работы не требует обязательного указания каких-либо настроек. Необходимые настройки библиотеки django-cors-headers весьма немногочисленны:

- `CORS_ORIGIN_ALLOW_ALL` — если `True`, то Django будет обрабатывать запросы, приходящие с любого домена. Если `False`, то будут обрабатываться только запросы с текущего домена и с доменов, заданных в параметрах `CORS_ORIGIN_WHITELIST` и `CORS_ORIGIN_REGEX_WHITELIST`. По умолчанию — `False`;
- `CORS_ORIGIN_WHITELIST` — список или кортеж доменов, запросы с которых разрешено обрабатывать. Домены задаются в виде строк. Пример:

```
CORS_ORIGIN_WHITELIST = [
    'http://www.bboard.ru',
    'https://www.bboard.ru',
```

```
    'https://admin.bboard.ru',
    'http://www.bb.net',
]
```

Значение по умолчанию — "пустой" список;

- `CORS_ORIGIN_REGEX_WHITELIST` — список или кортеж с регулярными выражениями, с которыми должны совпадать "разрешенные" домены:

```
CORS_ORIGIN_REGEX_WHITELIST = [
    r'^https?://(www|admin)\.bboard\.ru$',
    r'^http://(www\.)?bb\.net$',
]
```

Значение по умолчанию — "пустой" список;

- `CORS_URLS_REGEX` — регулярное выражение, с которым должен совпадать путь, запрос по которому будет допущен к обработке, в виде строки. По умолчанию: `"^.*$"` (регулярное выражение, совпадающее с любым путем).

Например: чтобы разрешить обработку запросов, приходящих с любых доменов, но только к тем путям, что включают префикс `api`, следует добавить в модуль settings.py пакета конфигурации такие выражения:

```
CORS_ORIGIN_ALLOW_ALL = True
CORS_URLS_REGEX = r'^/api/.*$'
```

> ***На заметку***
>
> Полное руководство по библиотеке django-cors-headers находится здесь: **https://github.com/ottoyiu/django-cors-headers/**.

28.2. Введение в Django REST framework. Вывод данных

28.2.1. Сериализаторы

Сериализатор в Django REST framework выступает аналогом формы. Сериализаторы, связанные с моделями, самостоятельно извлекают данные из модели и "умеют" сохранять в ней данные, полученные от фронтенда.

Код сериализаторов обычно записывается в модуле serializers.py пакета приложения. Этот модуль изначально отсутствует, и его придется создать самостоятельно.

Класс сериализатора, связанного с моделью, должен быть производным от класса `ModelSerializer` из модуля `rest_framework.serializers`. В остальном он мало отличается от формы, связанной с моделью (см. *главу 13*).

В листинге 28.1 приведен код сериализатора `RubricSerializer`, связанного с моделью `Rubric` и обрабатывающего рубрики.

Листинг 28.1. Сериализатор RubricSerializer

```
from rest_framework import serializers
from .models import Rubric

class RubricSerializer(serializers.ModelSerializer):
    class Meta:
        model = Rubric
        fields = ('id', 'name')
```

Проверим наш первый сериализатор в действии. Добавим в модуль views.py контроллер-функцию `api_rubrics()`, который будет выдавать клиентам список рубрик, закодированный в формат JSON. Код контроллера приведен в листинге 28.2.

Листинг 28.2. Контроллер, использующий сериализатор для вывода набора рубрик

```
from django.http import JsonResponse
from .models import Rubric
from .serializers import RubricSerializer

def api_rubrics(request):
    if request.method == 'GET':
        rubrics = Rubric.objects.all()
        serializer = RubricSerializer(rubrics, many=True)
        return JsonResponse(serializer.data, safe=False)
```

Конструктору класса сериализатора мы передали набор записей, который следует сериализовать, и параметр `many` со значением `True`, говоря тем самым, что сериализовать нужно именно набор записей, а не единичную запись. А ответ мы сформировали с помощью класса `JsonResponse`.

Наконец, добавим в список маршрутов уровня приложения (модуль urls.py пакета приложения) маршрут, указывающий на только что написанный нами контроллер:

```
from .views import api_rubrics
. . .
urlpatterns = [
    . . .
    path('api/rubrics/', api_rubrics),
    . . .
]
```

Запустим отладочный веб-сервер Django, откроем веб-обозреватель и перейдем по интернет-адресу **http://localhost:8000/api/rubrics/**. Мы увидим на экране список рубрик в формате JSON:

```
[{"id": 3, "name": "\u0411\u044b\u0442\u043e\u0432\u0430\u044f
\u0442\u0435\u0445\u043d\u0438\u043a\u0430"}, {"id": 5, "name":
```

```
"\u041c\u0435\u0431\u0435\u043b\u044c"},
. . . Остальной вывод пропущен
]
```

28.2.2. Веб-представление JSON

Если активно *веб-представление JSON*, то Django REST framework будет выводить JSON-данные на обычной веб-странице отформатированными для удобства чтения и с некоторыми дополнительными сведениями. Используем его, чтобы проверить, действительно ли сериализатор `RubricSerializer` выводит нам список рубрик.

Чтобы задействовать веб-представление, достаточно:

- указать у контроллера-функции декоратор `api_view(<допустимые HTTP-методы>)` из модуля `rest_framework.decorators`. *Допустимые HTTP-методы* задаются в виде списка со строковыми наименованиями этих методов;
- для формирования ответа вместо класса `JsonResponse` использовать класс `Response` из модуля `rest_framework.response`. Конструктор этого класса вызывается в формате: `Response(<отправляемые данные>)`.

В листинге 28.3 приведен полный код обновленной версии контроллера-функции `api_rubrics()`, которая реализует веб-представление.

Листинг 28.3. Пример контроллера, реализующего веб-представление

```
from rest_framework.response import Response
from rest_framework.decorators import api_view
from .models import Rubric
from .serializers import RubricSerializer

@api_view(['GET'])
def api_rubrics(request):
    if request.method == 'GET':
        rubrics = Rubric.objects.all()
        serializer = RubricSerializer(rubrics, many=True)
        return Response(serializer.data)
```

Сохраним исправленный код и попробуем наведаться по тому же интернет-адресу **http://localhost:8000/api/rubrics/**. На этот раз веб-обозреватель покажет нам веб-представление JSON (его часть можно увидеть на рис. 28.1).

Здесь выведены, прежде всего, сами JSON-данные в удобном для изучения виде и сведения о полученном ответе (код статуса, MIME-тип содержимого и пр.). Кнопка **GET** позволит вывести обычный JSON-код — для этого достаточно щелкнуть на расположенной в ее правой части стрелке, направленной вниз, и выбрать в появившемся на экране меню пункт **json**. Вернуть веб-представление данных можно выбором в том же меню пункта **api** или нажатием непосредственно кнопки **GET**, не затрагивая стрелки. Кнопка **OPTIONS** выводит сведения о самой веб-службе.

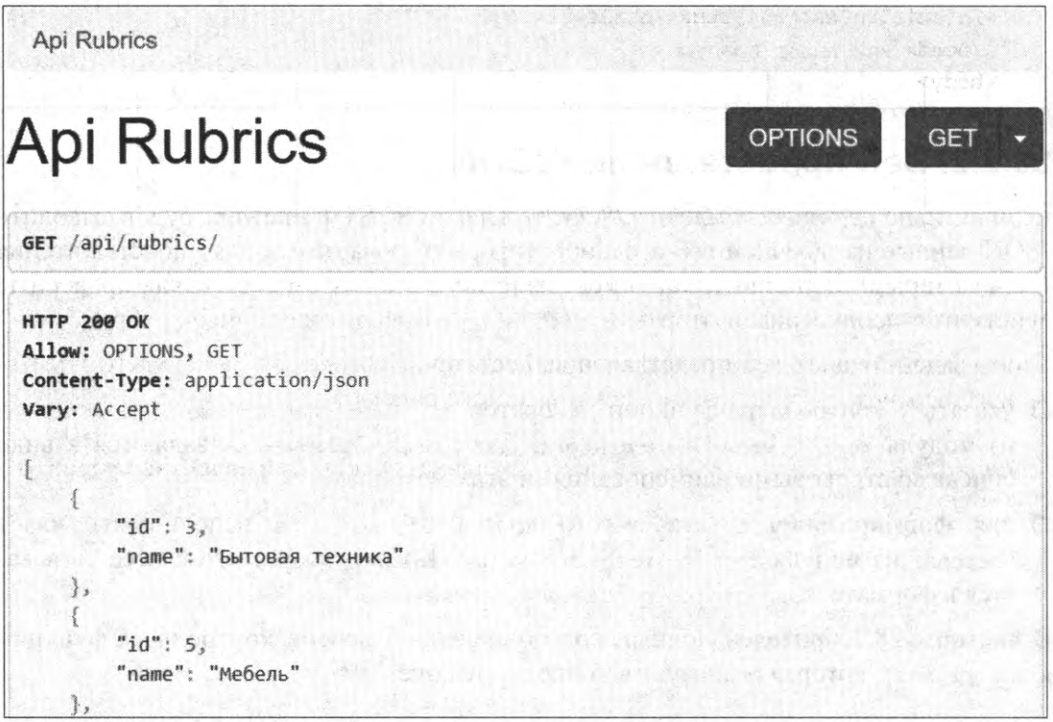


Рис. 28.1. Веб-представление JSON для модели рубрик `Rubric` (показана верхняя часть)

28.2.3. Вывод данных на стороне клиента

Полученные от бэкенда JSON-данные можно обработать и вывести на веб-странице средствами DOM и AJAX.

> ***Внимание!***
>
> Веб-обозреватель может загружать данные по технологии AJAX только с веб-сервера, но не с локального диска. Кроме того, многие веб-обозреватели блокируют AJAX-загрузку данных на страницах, открытых с локального диска.
>
> Автор использовал для обслуживания тестового фронтенда (написание которого описывается далее в этой главе) сторонний веб-сервер Apache HTTP Server с настройками по умолчанию.

В листинге 28.4 приведен код веб-страницы rubrics.html, на которой будет выводиться перечень рубрик, полученный от веб-службы.

Листинг 28.4. Веб-страница rubrics.html, выводящая полученный от веб-службы перечень рубрик

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
```

```
        <title>Список рубрик</title>
    </head>
    <body>
        <div id="list"></div>
    </body>
</html>
<script type="text/javascript" src="rubrics.js"></script>
```

Блок (блочный контейнер, тег) с якорем `list` будет использован для вывода маркированного списка с перечнем рубрик.

Листинг 28.5 представляет код файла веб-сценария rubrics.js, загружающего и выводящего перечень рубрик в виде маркированного списка.

Листинг 28.5. Веб-сценарий rubrics.js, загружающий и выводящий перечень рубрик на странице rubrics.html

```
const domain = 'http://localhost:8000/';

let list = document.getElementById('list');
let listLoader = new XMLHttpRequest();

listLoader.addEventListener('readystatechange', () => {
    if (listLoader.readyState == 4) {
        if (listLoader.status == 200) {
            let data = JSON.parse(listLoader.responseText);
            let s = '<ul>', d;
            for (let i = 0; i < data.length; i++) {
                d = data[i];
                s += '<li>' + d.name + '</li>';
            }
            s += '</ul>';
            list.innerHTML = s;
        } else
            window.alert(listLoader.statusText);
    }
});

function listLoad() {
    listLoader.open('GET', domain + 'api/rubrics/', true);
    listLoader.send();
}

listLoad();
```

Код, запускающий загрузку перечня рубрик, оформлен в виде функции `listLoad()`. Это позволит впоследствии, после добавления, правки или удаления рубрики, выполнить обновление перечня простым вызовом этой функции.

Сохраним файлы rubrics.html и rubrics.js в корневой папке стороннего веб-сервера. Запустим отладочный веб-сервер Django и сторонний веб-сервер. В веб-обозревателе выполним обращение по интернет-адресу **http://localhost/rubrics.html**. На открывшейся странице будет выведен перечень рубрик наподобие того, что показан на рис. 28.2.

- Бытовая техника
- Мебель
- Недвижимость
- Растения
- Сантехника
- Сельхозинвентарь
- Транспорт

Рис. 28.2. Список рубрик, полученный от веб-службы

28.2.4. Первый принцип REST: идентификация ресурса по интернет-адресу

Согласно первому принципу REST, любой ресурс, выдаваемый бэкендом, идентифицируется интернет-адресом — как и обычная веб-страница. Например, у нас ресурс "перечень рубрик" идентифицируется интернет-адресом **/api/rubrics/** — именно по нему фронтенд обращался к бэкенду для получения этого перечня.

Логично ресурс "сведения о рубрике" с заданным *ключом* идентифицировать интернет-адресом формата **/api/rubrics/<*ключ рубрики*>/**.

В листинге 28.6 приведен код контроллера, выдающий фронтенду сведения о рубрике с указанным ключом. Этот код мы добавим в модуль views.py.

Листинг 28.6. Контроллер, выдающий сведения об отдельной рубрике

```
@api_view(['GET'])
def api_rubric_detail(request, pk):
    if request.method == 'GET':
        rubric = Rubric.objects.get(pk=pk)
        serializer = RubricSerializer(rubric)
        return Response(serializer.data)
```

В вызове конструктора класса сериализатора `RubricSerializer` не следует указывать параметр `many` со значением `True`, т. к. сериализовать нужно лишь одну запись.

Добавим в список маршрутов уровня приложения маршрут, который укажет на новый контроллер:

```
from .views import api_rubric_detail
. . .
urlpatterns = [
    . . .
```

```
    path('api/rubrics/<int:pk>/', api_rubric_detail),
    path('api/rubrics/', api_rubrics),
    . . .
]
```

На страницу rubrics.html, непосредственно под блоком `list`, поместим веб-форму, в которой будут выводиться сведения о выбранной рубрике. Эту форму мы потом используем для добавления и правки рубрик. Вот HTML-код, создающий ее:

```
<form id="rubric_form" method="post">
    <input type="hidden" name="id" id="id">
    <p>Название: <input name="name" id="name"></p>
    <p><input type="reset" value="Очистить">
    <input type="submit" value="Сохранить"></p>
</form>
```

В форме присутствует скрытое поле `id`, хранящее ключ исправляемой рубрики. Он понадобится нам впоследствии.

Теперь исправим веб-сценарий, хранящийся в файле rubrics.js. Сначала сделаем так, чтобы рядом с названием каждой рубрики присутствовала гиперссылка **Вывести**, выводящая сведения о рубрике в только что созданной веб-форме. Вот правки, которые нам нужно внести в код:

```
loader.addEventListener('readystatechange', () => {
    if (loader.readyState == 4) {
        if (loader.status == 200) {
            . . .
            for (let i = 0; i < data.length; i++) {
                d = data[i];
                s += '<li>' + d.name + ' <a href="' + domain +
                    'api/rubrics/' + d.id +
                    '/" class="detail">Вывести</a></li>';
            }
            s += '</ul>';
            list.innerHTML = s;
            let links = list.querySelectorAll('ul li a.detail');
            links.forEach((link) =>
                {link.addEventListener('click', rubricLoad);});
        } else
            window.alert(listLoader.statusText);
    }
});
```

Здесь нужно пояснить три момента. Во-первых, мы привязали к каждой из созданных гиперссылок стилевой класс `detail` — это упростит задачу привязки к гиперссылкам обработчика события `click`. Во-вторых, записали интернет-адреса для загрузки рубрик непосредственно в тегах `<a>`, создающих гиперссылки, — это также упростит нам дальнейшее программирование. В-третьих, привязали к созданным гиперссылкам обработчик события `click` — функцию `rubricLoad()`.

Теперь допишем в файл rubrics.js код, выводящий сведения о выбранной рубрике:

```
let id = document.getElementById('id');
let name = document.getElementById('name');
let rubricLoader = new XMLHttpRequest();

rubricLoader.addEventListener('readystatechange', () => {
    if (rubricLoader.readyState == 4) {
        if (rubricLoader.status == 200) {
            let data = JSON.parse(rubricLoader.responseText);
            id.value = data.id;
            name.value = data.name;
        } else
            window.alert(rubricLoader.statusText);
    }
});

function rubricLoad(evt) {
    evt.preventDefault();
    rubricLoader.open('GET', evt.target.href, true);
    rubricLoader.send();
}
```

Функция `rubricLoad()` — обработчик события `click` гиперссылок **Вывести** — извлекает из атрибута `href` тега `<a>` гиперссылки, на которой был выполнен щелчок мышью, интернет-адрес и запускает процесс загрузки с этого адреса сведений о рубрике. Полученные сведения — название рубрики — выводятся в веб-форме.

Запустим отладочный веб-сервер Django и сторонний веб-сервер и перейдем по интернет-адресу **http://localhost/rubrics.html**. Когда на странице появится перечень рубрик, щелкнем на гиперссылке **Вывести** любой из них и проверим, выводятся ли в веб-форме сведения об этой рубрике.

28.3. Ввод и правка данных

28.3.1. Второй принцип REST: идентификация действия по HTTP-методу

Согласно второму принципу REST, действие, выполняемое над ресурсом (идентифицируемым уникальным интернет-адресом), обозначается HTTP-методом, указанным в запросе. В веб-службах REST применяются следующие методы:

- GET — выдача ресурса. Ресурс может представлять собой как перечень каких-либо сущностей, так и отдельную сущность (в нашем случае в качестве сущностей выступают рубрики);
- POST — создание нового ресурса;
- PUT — исправление значений всех полей у ресурса. Отметим, что при использовании этого метода фронтенд должен отправить бэкенду значения всех полей;

- PATCH — исправление отдельных полей у ресурса. В этом случае фронтенд может отправить бэкенду значения только тех полей, которые нужно исправить.

  На практике методы PUT и PATCH часто обозначают одно и то же действие — исправление либо всех полей ресурса, либо отдельных его полей (это зависит от конкретной реализации);
- DELETE — удаление ресурса.

Обычно создание нового ресурса реализует тот же контроллер, который выдает ресурс-перечень сущностей, а исправление и удаление — тот же контроллер, который выдает ресурс–отдельную сущность. Благодаря этому для реализации всех этих действий достаточно записать всего два маршрута.

В листинге 28.7 приведен исправленный код контроллеров `api_rubrics()` и `api_rubric_detail()`, которые получили поддержку добавления, правки и удаления рубрик.

Листинг 28.7. Контроллеры `api_rubrics()` и `api_rubric_detail()`, поддерживающие добавление, правку и удаление рубрик

```
from rest_framework import status

@api_view(['GET', 'POST'])
def api_rubrics(request):
    if request.method == 'GET':
        rubrics = Rubric.objects.all()
        serializer = RubricSerializer(rubrics, many=True)
        return Response(serializer.data)
    elif request.method == 'POST':
        serializer = RubricSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
                            status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
                        status=status.HTTP_400_BAD_REQUEST)

@api_view(['GET', 'PUT', 'PATCH', 'DELETE'])
def api_rubric_detail(request, pk):
    rubric = Rubric.objects.get(pk=pk)
    if request.method == 'GET':
        serializer = RubricSerializer(rubric)
        return Response(serializer.data)
    elif request.method == 'PUT' or request.method == 'PATCH':
        serializer = RubricSerializer(rubric, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
```

```
        return Response(serializer.errors,
                        status=status.HTTP_400_BAD_REQUEST)
    elif request.method == 'DELETE':
        rubric.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

Сохранение и удаление записей с помощью сериализаторов, связанных с моделями, выполняется точно так же, как и в случае применения связанных с моделями форм. Мы вызываем методы `is_valid()`, `save()` и `delete()`, а все остальное выполняют фреймворк Django и библиотека Django REST framework.

Теперь отметим три важных момента. Во-первых, мы добавили в вызовы декоратора `api_view()` обозначения HTTP-методов POST, PUT, PATCH и DELETE, указав тем самым, что контроллеры поддерживают запросы, выполненные с применением этих методов. Если этого не сделать, мы получим ошибку.

Во-вторых, чтобы занести в сериализатор данные, полученные от фронтенда, мы присваиваем их именованному параметру `data` конструктора класса сериализатора. Сами эти данные можно извлечь из атрибута `data` объекта запроса. Вот пример:

```
serializer = RubricSerializer(data=request.data)
```

В-третьих, после успешного создания рубрики мы отсылаем фронтенду ответ с кодом `201` (ресурс успешно создан). Это выполняется передачей конструктору класса `Response` числового кода статуса посредством именованного параметра `status`:

```
return Response(serializer.data, status=status.HTTP_201_CREATED)
```

При успешном исправлении рубрики отправленный клиенту ответ будет иметь код статуса по умолчанию: `200`. После успешного удаления рубрики клиент получит ответ с кодом статуса `204` (ресурс удален). Если же отправленные данные некорректны, то фронтенд получит ответ с кодом `400` (некорректный запрос).

Указанные коды статуса хранятся в переменных `HTTP_201_CREATED`, `HTTP_204_NO_CONTENT` и `HTTP_400_BAD_REQUEST`, объявленных в модуле `rest_framework.status`.

Настало время "научить" фронтенд создавать и править рубрики. Откроем файл rubrics.js и добавим в него следующий код:

```
let rubricUpdater = new XMLHttpRequest();
rubricUpdater.addEventListener('readystatechange', () => {
    if (rubricUpdater.readyState == 4) {
        if ((rubricUpdater.status == 200) ||
            (rubricUpdater.status == 201)) {
            listLoad();
            name.form.reset();
            id.value = '';
        } else
            window.alert(rubricUpdater.statusText);
    }
});
```

```
name.form.addEventListener('submit', (evt) => {
    evt.preventDefault();
    let vid = id.value, url, method;
    if (vid) {
        url = 'api/rubrics/' + vid + '/';
        method = 'PUT';
    } else {
        url = 'api/rubrics/';
        method = 'POST';
    }
    let data = JSON.stringify({id: vid, name: name.value});
    rubricUpdater.open(method, domain + url, true);
    rubricUpdater.setRequestHeader('Content-Type', 'application/json');
    rubricUpdater.send(data);
});
```

Обработчик события `submit` веб-формы проверяет, хранится ли в скрытом поле `id` ключ рубрики. Если скрытое поле хранит ключ, значит, выполняется правка уже имеющей рубрики, в противном случае в список добавляется новая рубрика. Исходя из этого, вычисляется интернет-адрес ресурса, по которому следует выполнить запрос, и выбирается HTTP-метод, применяемый для его отсылки. Введенные в форму данные кодируются в формат JSON, для чего они сначала представляются в виде объекта класса `Object`, а потом "пропускаются" через статический метод `stringify()` класса `JSON`. У готовящегося к отправке запроса в заголовке `Content-Type` указывается тип отсылаемых данных: `application/json`, — для чего используется метод `setRequestHeader()` класса `XMLHttpRequest` (это нужно, чтобы Django REST framework подобрала подходящий парсер — о парсерах мы поговорим позже). Наконец, готовый запрос, хранящий введенные данные, отсылается веб-службе.

Обработчик события `readystatechange` после получения ответа с кодом `200` или `201` (т. е. после успешного исправления или добавления рубрики) обновляет перечень рубрик, очищает форму и заносит в скрытое поле `id` веб-формы "пустую" строку. Так он сообщает пользователю, что все прошло нормально, и готовит форму для ввода новой рубрики.

Сразу же добавим в файл rubrics.js код, реализующий удаление рубрик. Сначала сделаем так, чтобы в перечне рубрик выводились гиперссылки **Удалить**:

```
listLoader.addEventListener('readystatechange', () => {
    if (listLoader.readyState == 4) {
        if (listLoader.status == 200) {
            . . .
            for (let i = 0; i < data.length; i++) {
                d = data[i];
                s += '<li>' + d.name + ' <a href="' + domain +
                    'api/rubrics/' + d.id +
                    '/" class="detail">Вывести</a> <a href="' + domain +
```

```
                    'api/rubrics/' + d.id +
                    '/" class="delete">Удалить</a></li>';
            }
            . . .
            links = list.querySelectorAll('ul li a.delete');
            links.forEach((link) =>
                {link.addEventListener('click', rubricDelete);});
        } else
            . . .
    }
});
```

К созданным гиперссылкам привязывается обработчик события `click` — функция `rubricDelete()`, которая и запустит удаление выбранной рубрики.

Допишем код, удаляющий рубрики:

```
let rubricDeleter = new XMLHttpRequest();
rubricDeleter.addEventListener('readystatechange', () => {
    if (rubricDeleter.readyState == 4) {
        if (rubricDeleter.status == 204)
            listLoad();
        else
            window.alert(rubricDeleter.statusText);
    }
});

function rubricDelete(evt) {
    evt.preventDefault();
    rubricDeleter.open('DELETE', evt.target.href, true);
    rubricDeleter.send();
}
```

Здесь обработчик события `readystatechange` ожидает ответа с кодом `204`, чтобы удостовериться, что рубрика была успешно удалена.

Перезапустим отладочный веб-сервер, обновим страницу rubrics.html в веб-обозревателе и проверим, как все работает.

28.3.2. Парсеры веб-форм

Получив от фронтенда какие-либо данные, библиотека Django REST framework пытается разобрать их, используя подходящий парсер.

Парсер — это класс, выполняющий разбор переданных фронтендом данных и их преобразование в объекты языка Python, пригодные для дальнейшей обработки. Парсер задействуется программным ядром библиотеки перед вызовом контроллера — таким образом, последний получит уже обработанные данные.

В составе Django REST framework поставляются три наиболее интересных для нас класса парсеров:

- `JSONParser` — обрабатывает данные, представленные в формате JSON (MIME-тип `application/json`);
- `FormParser` — обрабатывает данные из обычных веб-форм (MIME-тип `application/x-www-form-urlencoded`);
- `MultiPartParser` — обрабатывает данные из веб-форм, выгружающих файлы (MIME-тип `multipart/form-data`).

По умолчанию активны все эти три класса парсеров. Библиотека выбирает нужный парсер, основываясь на MIME-типе переданных фронтендом данных, который записывается в заголовке `Content-Type` запроса. Именно поэтому в коде фронтенда перед отправкой данных необходимо указать их MIME-тип.

Ранее мы пересылали от клиента данные, закодированные в формате JSON, и обрабатывались они парсером `JSONParser`. Однако мы можем переслать данные в формате обычных форм:

```
data = 'id=' + encodeURIComponent(vid);
data += '&name=' + encodeURIComponent(name.value);
data += '&order=' + encodeURIComponent(order.value);
rubricUpdater.setRequestHeader('Content-Type',
                               'application/x-www-form-urlencoded');
```

Такие данные будут обработаны парсером `FormParser`.

28.4. Контроллеры-классы Django REST framework

28.4.1. Контроллер-класс низкого уровня

Контроллер-класс `APIView` из модуля `rest_framework.views` всего лишь, как и аналогичный ему класс `View` (см. *разд. 10.2.1*) — кстати, его суперкласс, — осуществляет диспетчеризацию по HTTP-методу: при получении запроса по методу GET вызывает метод `get()`, при получении запроса по методу POST — метод `post()` и т. д. Если был получен запрос по методу PATCH, а метод `patch()` отсутствует, будет выполнен метод `put()`.

Класс `APIView` реализует веб-представление (см. *разд. 28.2.2*) самостоятельно, так что декоратор `api_view()` указывать не нужно.

В листинге 28.8 приведен код контроллера-класса `APIRubrics`, производного от `APIView` и выполняющего вывод перечня рубрик и добавление новой рубрики.

Листинг 28.8. Пример использования класса `APIView`

```
from rest_framework.views import APIView

class APIRubrics(APIView):
    def get(self, request):
        rubrics = Rubric.objects.all()
```

```
        serializer = RubricSerializer(rubrics, many=True)
        return Response(serializer.data)
    def post(self, request):
        serializer = RubricSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
                            status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
                        status=status.HTTP_400_BAD_REQUEST)
```

Маршрут, указывающий на такой контроллер-класс, записывается аналогично маршрутам на контроллеры-классы, рассмотренные в *главе 10*:

```
urlpatterns = [
    . . .
    path('api/rubrics/', APIRubrics.as_view()),
    . . .
]
```

Код контроллера-класса `APIRubrics` очень похож на код контроллеров-функций из листинга 28.7, выполняющих те же задачи. Так что контроллер-класс `APIRubricDetail`, который реализует вывод сведений о выбранной рубрике, правку и удаление рубрик, вы, уважаемые читатели, можете написать самостоятельно.

28.4.2. Контроллеры-классы высокого уровня: комбинированные и простые

Еще Django REST framework предоставляет набор высокоуровневых классов, объявленных в модуле `rest_framework.generics`. Прежде всего, это четыре комбинированных контроллера-класса, которые могут выполнять сразу два или три действия, в зависимости от HTTP-метода, которым был отправлен запрос:

- `ListCreateAPIView` — выполняет выдачу ресурса-перечня сущностей и создание нового ресурса (т. е. обрабатывает HTTP-методы GET и POST);
- `RetrieveUpdateDestroyAPIView` — выполняет выдачу ресурса–отдельной сущности, правку и удаление ресурса (обрабатывает методы GET, PUT, PATCH и DELETE);
- `RetrieveUpdateAPIView` — выполняет выдачу ресурса–отдельной сущности и правку ресурса (методы GET, PUT и PATCH);
- `RetrieveDestroyAPIView` — выполняет выдачу ресурса–отдельной сущности и удаление ресурса (методы GET и DELETE).

Как минимум, в таких классах нужно задать набор записей, который будет обрабатываться, и сериализатор, который будет применяться для пересылки данных фронтенду. Набор записей указывается в атрибуте `queryset`, а сериализатор — в атрибуте `serializer_class`.

Листинг 28.9 показывает новую реализацию контроллеров-классов `APIRubrics` и `APIRubricDetail` — основанную на классах `ListCreateAPIView` и `RetrieveUpdateDestroyAPIView`.

Листинг 28.9. Пример использования классов `ListCreateAPIView` и `RetrieveUpdateDestroyAPIView`

```
from rest_framework import generics

class APIRubrics(generics.ListCreateAPIView):
    queryset = Rubric.objects.all()
    serializer_class = RubricSerializer

class APIRubricDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Rubric.objects.all()
    serializer_class = RubricSerializer
```

Если интерфейс веб-службы предусматривает выполнение только какого-либо одного действия (например, вывода списка рубрик), комбинированные классы будут избыточными. В таких случаях удобнее задействовать более простые классы, выполняющие только одно действие:

- `ListAPIView` — выполняет выдачу ресурса-перечня сущностей (т. е. обрабатывает HTTP-метод GET);
- `RetrieveAPIView` — выполняет выдачу ресурса–отдельной сущности (метод GET);
- `CreateAPIView` — выполняет создание нового ресурса (метод POST);
- `UpdateAPIView` — выполняет правку ресурса (методы PUT и PATCH);
- `DestroyAPIView` — выполняет удаление ресурса (метод DELETE).

Используются они точно так же, как и комбинированные контроллеры-классы. Например, код контроллера, выводящего список рубрик, может быть таким:

```
class APIRubricList(generics.ListAPIView):
    queryset = Rubric.objects.all()
    serializer_class = RubricSerializer
```

28.5. Метаконтроллеры

Метаконтроллер — это комбинированный контроллер-класс, выполняющий сразу все возможные действия: выдачу ресурсов-перечней, ресурсов–отдельных сущностей, добавление, правку и удаление ресурсов. Он может заменить два обычных комбинированных контроллера-класса, наподобие показанных в листинге 28.9. Помимо этого, метаконтроллер предоставляет средства для генерирования всех необходимых маршрутов.

Метаконтроллер, связанный с моделью, создается как подкласс класса `ModelViewSet` из модуля `rest_framework.viewsets`. В нем с применением атрибутов `queryset` и

`serializer_class` указываются набор записей, с которым будет выполняться работа, и сериализатор, управляющий отправкой данных фронтенду.

В листинге 28.10 приведен код метаконтроллера `APIRubricViewSet`, работающего со списком рубрик. Как видим, он очень прост и компактен.

Листинг 28.10. Метаконтроллер `APIRubricViewSet`, работающий со списком рубрик

```
from rest_framework.viewsets import ModelViewSet

class APIRubricViewSet(ModelViewSet):
    queryset = Rubric.objects.all()
    serializer_class = RubricSerializer
```

Чтобы сгенерировать маршруты, указывающие на отдельные функции метаконтроллера, нужно выполнить три действия:

- получить объект *генератора* таких *маршрутов*, представляющий собой экземпляр класса `DefaultRouter` из модуля `rest_framework.routers`. Конструктор этого класса вызывается без параметров;
- зарегистрировать в генераторе маршрутов метаконтроллер, связав его с выбранным префиксом. Это выполняется вызовом метода `register()` класса `DefaultRouter` в формате:

  `register(<строка с префиксом>, <ссылка на класс метаконтроллера>)`

- добавить сгенерированные маршруты в список уровня приложения или проекта, воспользовавшись функцией `include()` (см. *разд. 8.3*). Сами маршруты можно извлечь из атрибута `urls` генератора маршрутов.

Вот пример генерирования набора маршрутов для метаконтроллера `APIRubricViewSet` из листинга 28.10:

```
from rest_framework.routers import DefaultRouter
from django.urls import path, include

router = DefaultRouter()
router.register('rubrics', APIRubricViewSet)

urlpatterns = [
    . . .
    path('api/', include(router.urls)),
    . . .
]
```

В результате в список будут добавлены два следующих маршрута:

- **api/rubrics/** — выполняет при запросе с применением HTTP-метода:
  - GET — выдачу списка рубрик;
  - POST — добавление новой рубрики;

- **api/rubrics/<*ключ*>** — выполняет при запросе с применением HTTP-метода:
  - GET — выдачу рубрики с указанным *ключом*;
  - PUT или PATCH — правку рубрики с указанным *ключом*;
  - DELETE — удаление рубрики с указанным *ключом*.

Помимо класса `ModelViewSet`, библиотека Django REST framework предлагает класс `ReadOnlyModelViewSet`, объявленный в том же модуле `rest_framework.viewsets`. Он реализует функциональность только по выдаче ресурса-списка сущностей и ресурса–отдельной сущности и подходит для случаев, когда фронтенды должны только получать данные от бэкенда, но не добавлять и править их. Пример метаконтроллера, обрабатывающего список рубрик и позволяющего только считывать данные, приведен в листинге 28.11.

Листинг 28.11. Метаконтроллер, реализующий только выдачу рубрик

```
from rest_framework.viewsets import ReadOnlyModelViewSet

class APIRubricViewSet(ReadOnlyModelViewSet):
    queryset = Rubric.objects.all()
    serializer_class = RubricSerializer
```

28.6. Разграничение доступа в Django REST framework

28.6.1. Третий принцип REST: данные клиента хранятся на стороне клиента

В сайтах, построенных по традиционной архитектуре, данные клиента, включая признак, выполнил ли пользователь вход на сайт, хранятся на стороне сервера. Благодаря этому сервер всегда может проверить, выполнил ли текущий пользователь вход на сайт или он является гостем.

Но в веб-службах REST данные клиента должны храниться на стороне клиента. Фронтенд должен сохранить введенные пользователем имя и пароль: в локальном хранилище DOM, cookie или в обычных переменных.

Чтобы бэкенд смог проверить, имеет ли текущий пользователь права на доступ к данным, фронтенд должен посылать ему сохраненные имя и пароль в каждом запросе. Они записываются в заголовке `Authorization` запроса в виде строки формата `Basic` *<имя и пароль>*, где *имя и пароль* представляются в формате *<имя>:<пароль>* и кодируются в кодировке base64. Для кодирования можно использовать метод `btoa()`, поддерживаемый классом `Window`.

Вот пример кода фронтенда, отправляющего имя и пароль пользователя бэкенду:

```
const username = 'editor';
const password = '1988win1993';
```

```
const credentials = window.btoa(username + ':' + password);
. . .
listLoader.open('GET', domain + 'api/rubrics/', true);
listLoader.setRequestHeader('Authorization', 'Basic ' + credentials);
listLoader.send();
```

Таким образом реализуется *основная аутентификация* (basic authentication), при которой в каждом клиентском запросе бэкенду пересылаются непосредственно имя и пароль пользователя.

28.6.2. Классы разграничения доступа

Чтобы в коде бэкенда указать, какие пользователи имеют доступ к определенному контроллеру, следует задать у этого контроллера набор необходимых классов разграничения доступа.

Классы разграничения доступа объявлены в модуле `rest_framework.permissions`. Наиболее часто используемые из них приведены далее:

- `AllowAny` — разрешает доступ к данным всем — и зарегистрированным пользователям, и гостям. Установлен по умолчанию;
- `IsAuthenticated` — разрешает доступ к данным только зарегистрированным пользователям;
- `IsAuthenticatedOrReadOnly` — предоставляет полный доступ к данным только зарегистрированным пользователям, гости получат доступ лишь на чтение;
- `IsAdminUser` — разрешает доступ к данным только зарегистрированным пользователям со статусом персонала;
- `DjangoModelPermissions` — разрешает доступ к данным только зарегистрированным пользователям, имеющим необходимые права на работу с этими данными (о правах пользователей рассказывалось в *главе 15*);
- `DjangoModelPermissionsOrAnonReadOnly` — предоставляет полный доступ к данным только зарегистрированным пользователям, имеющим необходимые права на работу с ними. Все прочие пользователи, включая гостей, получат доступ только на чтение.

Перечень классов разграничения доступа записывается в виде кортежа. Указать его можно:

- в контроллере-функции — с помощью декоратора `permission_classes(`*<перечень классов>*`)` из модуля `rest_framework.decorators`:

```
from rest_framework.decorators import api_view, permission_classes
from rest_framework.permissions import IsAuthenticated

@api_view(['GET', 'POST'])
@permission_classes((IsAuthenticated,))
def api_rubrics(request):
    . . .
```

- в контроллере-классе — в атрибуте `permission_classes`:

```
from rest_framework.permissions import IsAuthenticated

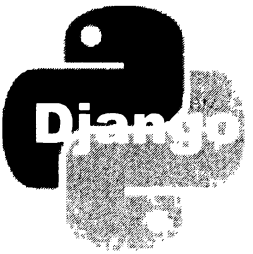
class APIRubricViewSet(ModelViewSet):
    . . .
    permission_classes = (IsAuthenticated,)
```

Перечень классов разграничения доступа, используемых по умолчанию, указывается в настройках проекта (в модуле settings.py пакета конфигурации) следующим образом:

```
# По умолчанию разрешаем доступ только зарегистрированным пользователям
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    )
}
```

Чтобы пользователь получил доступ к данным, он должен "пройти" через все классы разграничения доступа, указанные в перечне. Так, если указать классы `IsAdminUser` и `DjangoModelPermissions`, то доступ к данным получат только пользователи со статусом персонала, имеющие права на работу с этими данными.

> ***На заметку***
>
> Библиотека Django REST framework поддерживает множество других программных инструментов: сериализаторы, не связанные с моделями, иные способы аутентификации (жетонную, при которой от клиента серверу пересылаются не имя и пароль пользователя, а идентифицирующий его электронный жетон, и сессионную, традиционную, при которой сведения о клиенте сохраняются на стороне сервера), дополнительные классы разграничения доступа и т. п. К сожалению, ограниченный объем книги не позволяет рассказать обо всем этом.



ГЛАВА 29

Средства журналирования и отладки

Django предоставляет ряд отладочных инструментов, которые пригодятся как при отладке разрабатываемого сайта, так и в процессе его эксплуатации, для фиксации возникающих ошибок.

29.1. Средства журналирования

Журналирование — это фиксация каких-либо событий, происходящих в работающем сайте, обычно — ошибок, возникающих в его коде. Подсистема журналирования, встроенная в Django, может выводить сообщения о произошедших событиях на экран, отправлять их по электронной почте или записывать в файл журнала.

29.1.1. Настройка подсистемы журналирования

Настройки подсистемы журналирования записываются в модуле settings.py пакета конфигурации. Все они указываются в параметре `LOGGING` в виде словаря, ключи элементов которого задают названия различных параметров, а значения элементов — значения этих параметров.

Вот параметры, доступные для указания:

- `version` — номер версии стандарта, в котором записываются настройки подсистемы журналирования, в виде целого числа. На данный момент поддерживается только версия 1;
- `formatters` — перечень доступных для использования форматировщиков. *Форматировщик* определяет формат сообщений о произошедших событиях, выводимых подсистемой;
- `filters` — перечень доступных фильтров сообщений. *Фильтры* отбирают для вывода только сообщения, удовлетворяющие определенным условиям;
- `handlers` — перечень доступных обработчиков. *Обработчики* непосредственно выполняют вывод определенным способом (на консоль, в файл, по электронной

почте и др.) сообщений, прошедших через фильтры, в формате, заданном форматировщиком;

- `loggers` — перечень доступных регистраторов. *Регистратор* собирает все сообщения, отправленные заданными подсистемами Django, и передает их указанным обработчикам для вывода;
- `disable_existing_loggers` — если `True`, то регистраторы, используемые по умолчанию, работать не будут, если `False` — будут (по умолчанию — `True`).

29.1.2. Объект сообщения

Каждое сообщение о произошедшем событии представляется в виде экземпляра класса `LogRecord` из модуля `logging` стандартной библиотеки Python.

Вот атрибуты класса `LogRecord`, хранящие полезную для нас информацию о сообщении:

- `message` — текст сообщения в виде строки;
- `levelname` — обозначение уровня сообщения в виде строки: `"DEBUG"`, `"INFO"`, `"WARNING"`, `"ERROR"` или `"CRITICAL"`. Все эти уровни сообщений описаны в табл. 23.1;
- `levelno` — обозначение уровня сообщения в виде целого числа;
- `pathname` — полный путь выполняемого в данный момент файла в виде строки;
- `filename` — имя выполняемого в данный момент файла в виде строки;
- `module` — имя выполняемого в данный момент модуля, полученное из имени файла путем удаления у него расширения, в виде строки;
- `lineno` — порядковый номер выполняемой в данный момент строки программного кода в виде целого числа;
- `funcName` — имя выполняемой в данный момент функции в виде строки;
- `asctime` — временна́я отметка создания сообщения в виде строки;
- `created` — временна́я отметка создания сообщения в виде вещественного числа, представляющего собой количество секунд, что прошли с полуночи 1 января 1970 года. Для формирования этой величины применяется функция `time()` из модуля `time` Python;
- `msecs` — миллисекунды из времени создания сообщения в виде целого числа;
- `relativeCreated` — количество миллисекунд, прошедших между запуском регистратора и созданием текущего сообщения, в виде целого числа;
- `exc_info` — кортеж из трех значений: ссылки на класс исключения, самого объекта исключения и объекта, хранящего стек вызова. Для формирования этого кортежа применяется функция `exc_info()` из модуля `sys` Python;
- `stack_info` — объект, хранящий стек вызовов;
- `process` — идентификатор процесса в виде целого числа (если таковой удается определить);

- `processName` — имя процесса в виде строки (если таковое удается определить);
- `thread` — идентификатор потока в виде целого числа (если таковой удается определить);
- `threadName` — имя потока в виде строки (если таковое удается определить);
- `name` — имя регистратора, оставившего это сообщение, в виде строки.

29.1.3. Форматировщики

Форматировщик задает формат, в котором представляется сообщение о произошедшем событии.

Перечень форматировщиков в параметре `formatters` указывается в виде словаря. Ключами его элементов служат имена объявляемых форматировщиков, а значения элементов задают значения параметров соответствующих форматировщиков.

Доступны следующие параметры форматировщиков:

- `format` — строка формата для формирования текста сообщения. Для вставки в текст значений атрибутов объекта сообщения (они были приведены в *разд. 29.1.2*) применяются языковые конструкции вида `%(`*<имя атрибута>*`)s`;
- `datefmt` — строка формата для формирования временны́х отметок. В ней должны присутствовать специальные символы, поддерживаемые функцией `strftime()` из модуля `time`. По умолчанию: `"%Y-%m-%d %H:%M:%S,uuu"`.

Пример объявления простого форматировщика с именем `simple`:

```
LOGGING = {
    . . .
    'formatters': {
        'simple': {
            'format': '[%(asctime)s] %(levelname)s: %(message)s',
            'datefmt': '%Y.%m.%d %H:%M:%S',
        },
    },
    . . .
}
```

Он выводит сообщения в формате `[`*<временна́я отметка>*`]` *<уровень>*`:` *<текст>*, где *временна́я отметка* создания события имеет формат *<год>*`.`*<месяц>*`.`*<число>* *<часы>*`:` *<минуты>*`:` *<секунды>*.

29.1.4. Фильтры

Фильтр отбирает для вывода только те сообщения, которые удовлетворяют определенным условиям.

Перечень фильтров записывается в таком же формате, как и перечень форматировщиков (см. *разд. 29.1.3*). У каждого объявленного фильтра следует задать обяза-

тельный параметр () (две круглые скобки), указывающий строку с именем класса фильтра. Если конструктор этого класса принимает какие-либо параметры, то они задаются там же — в настройках фильтра.

Все доступные классы фильтров объявлены в модуле `django.utils.log`:

- `RequireDebugTrue` — выводит сообщения только в том случае, если включен отладочный режим (параметру `DEBUG` настроек проекта присвоено значение `True`). Об отладочном и эксплуатационном режимах сайта рассказывалось в *разд. 3.3.1*;
- `RequireDebugFalse` — выводит сообщения только в том случае, если включен эксплуатационный режим (параметру `DEBUG` настроек проекта присвоено значение `False`).

  Пример использования этих классов фильтров:

```
LOGGING = {
    . . .
    'filters': {
        'require_debug_false': {
            '()': 'django.utils.log.RequireDebugFalse',
        },
        'require_debug_true': {
            '()': 'django.utils.log.RequireDebugTrue',
        },
    },
    . . .
}
```

- `CallbackFilter(callback=`*<функция>*`)` — выводит только те сообщения, для которых указанная в параметре `callback` *функция* вернет `True`. *Функция* должна в качестве единственного параметра принимать сообщение, представленное экземпляром класса `LogRecord` (см. *разд. 29.1.2*).

  Пример объявления фильтра `info_filter`, отбирающего только сообщения уровня `INFO`:

```
def info_filter(message):
    return message.levelname == 'INFO'
. . .
LOGGING = {
    . . .
    'filters': {
        'info_filter': {
            '()': 'django.utils.log.CallbackFilter',
            'callback': info_filter,
        },
    },
    . . .
}
```

29.1.5. Обработчики

Обработчики непосредственно выполняют вывод сообщений на поддерживаемые ими устройства: в командной строке, в файл или куда-либо еще.

Перечень обработчиков записывается в таком же формате, что и перечень форматировщиков (см. *разд. 29.1.3*). У каждого из обработчиков можно задать такие параметры:

- `class` — строка с именем класса обработчика, который будет выполнять вывод сообщений. Поддерживаемые Django классы обработчиков будут рассмотрены позже;
- `level` — минимальный уровень сообщений в виде строкового обозначения. Обработчик станет выводить сообщения, уровень которых не меньше заданного, сообщения меньшего уровня выводиться не будут. Если параметр не указан, то обработчик будет выводить сообщения всех уровней;
- `formatter` — форматировщик, который будет применяться для формирования сообщений. Если параметр не указан, сообщения будут иметь формат по умолчанию: *<текст сообщения>*;
- `filters` — список фильтров, через которые будут проходить выводимые обработчиком сообщения. Чтобы сообщение было выведено, оно должно пройти через все включенные в список фильтры. Если параметр не указан, фильтры использоваться не будут;
- именованные параметры, принимаемые конструктором класса обработчика (если таковые есть).

Пример указания фильтра, выводящего сообщения уровня `ERROR` и выше на консоль с применением фильтра `require_debug_true` и форматировщика `simple`:

```
LOGGING = {
    . . .
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'level': 'ERROR',
            'formatter': 'simple',
            'filters': ['require_debug_true'],
        },
    },
    . . .
}
```

Вот наиболее часто используемые классы обработчиков:

- `logging.StreamHandler` — выводит сообщения в консоли;
- `logging.FileHandler(filename=`*<путь к файлу>*`[, mode='a'][, encoding=None][, delay=False])` — сохраняет сообщения в файле с заданным *путем*. Размер получающегося файла не ограничен.

Параметр `mode` задает режим открытия файла; если он не указан, файл открывается для добавления. Параметр `encoding` указывает кодировку файла; если он опущен, то Python сам выберет кодировку.

Если параметру `delay` задать значение `True`, файл будет открыт только в момент вывода самого первого сообщения. Если же присвоить ему значение `False`, файл будет открыт непосредственно при инициализации класса обработчика (поведение по умолчанию).

Пример использования этого класса:

```
LOGGING = {
    . . .
    'handlers': {
        'file': {
            'class': 'logging.FileHandler',
            'level': 'INFO',
            'filename': 'd:/logs/django-site.log',
        },
    },
    . . .
}
```

- `logging.handlers.RotatingFileHandler` — то же самое, что `FileHandler`, но вместо одного большого файла создает набор файлов ограниченного размера. Как только размер очередного файла приближается к указанному пределу, создается новый файл. Формат вызова конструктора этого класса:

```
RotatingFileHandler(filename=<путь к файлу>[, maxBytes=0][,
                    backupCount=0][, mode='a'][, encoding=None][,
                    delay=False])
```

Параметр `maxBytes` устанавливает размер файла, при превышении которого будет создан новый файл с сообщениями, в байтах. Если задать значение `0`, то класс обработчика будет сохранять все сообщения в один файл неограниченного размера, т. е. вести себя, как класс `FileHandler`.

Параметр `backupCount` указывает количество ранее созданных файлов, которые будут сохраняться на диске. К расширениям этих файлов будут добавляться последовательно увеличивающиеся целые числа. Так, если сообщения записываются в файл django-site.log, предыдущие файлы получат имена django-site.log.1, django-site.log.2 и т. д. Если количество таких файлов превысит заданную в параметре величину, наиболее старые файлы будут удалены.

Если параметру `backupCount` присвоить значение `0`, то все сообщения станут сохраняться в один файл неограниченного размера (при этом значение параметра `maxBytes` будет проигнорировано).

О назначении остальных параметров конструктора говорилось в описании класса `FileHandler`.

Пример:

```
LOGGING = {
    . . .
    'handlers': {
        'file': {
            'class': 'logging.handlers.RotatingFileHandler',
            'level': 'INFO',
            'filename': 'd:/logs/django-site.log',
            'maxBytes': 1048576,
            'backupCount': 10,
        },
    },
    . . .
}
```

- `logging.handlers.TimedRotatingFileHandler` — то же самое, что `RotatingFileHandler`, только начинает запись в новый файл не при превышении указанного размера файла, а по прошествии заданного временного промежутка. Формат вызова конструктора:

```
TimedRotatingFileHandler(filename=<путь к файлу>[, when='H'][,
                         interval=1][, utc=False][, atTime=None][,
                         backupCount=0][, encoding=None][,
                         delay=False])
```

Параметр `when` указывает разновидность промежутка времени, через который следует создать новый файл. Доступны значения:

- `"S"` — секунды;
- `"M"` — минуты;
- `"H"` — часы;
- `"D"` — дни;
- `"W<номер дня недели>"` — каждый день недели с указанным *номером*. В качестве *номера дня недели* нужно указать целое число от `0` (понедельник) до `6` (воскресенье);
- `"midnight"` — каждый день в полночь.

Параметр `interval` задает количество промежутков времени заданной разновидности, после которых нужно начинать запись в новый файл (по умолчанию: `1`).

Примеры:

```
# Создавать новый файл каждый день
'when': 'D',
# Создавать новый файл каждые шесть часов
'interval': 6,
# Создавать новый файл каждые десять дней
'when': 'D',
'interval': 10,
```

```
# Создавать новый файл каждую субботу
'when': 'W5',
```

К расширениям ранее созданных файлов с сообщениями будут добавляться строки формата *<год>-<месяц>-<число>*[_*<часы>-<минуты>-<секунды>*], причем вторая половина, с *часами*, *минутами* и *секундами*, может отсутствовать, если задан временной интервал, превышающий один день.

Если параметру `utc` присвоить значение `True`, будет применяться всемирное координированное время (UTC). Присвоение значения `False` укажет Django использовать местное время.

Параметр `atTime` принимается во внимание только в том случае, если параметру `when` дано значение "`W`*<номер дня недели>*" или "`midnight`". Значением параметра `atTime` должна быть отметка времени в виде объекта типа `time` из модуля `datetime`, которая укажет время, в которое следует начать запись в новый файл.

О назначении остальных параметров конструктора говорилось в описании классов `FileHandler` и `RotatingFileHandler`.

Пример:

```
LOGGING = {
    . . .
    'handlers': {
        'file': {
            'class': 'logging.handlers.TimedRotatingFileHandler',
            'level': 'INFO',
            'filename': 'd:/logs/django-site.log',
            'when': 'D',
            'interval': 10,
            'utc': True,
            'backupCount': 10,
        },
    },
    . . .
}
```

- `django.utils.log.AdminEmailHandler([include_html=False][,][email_backend=None])` — отправляет сообщения по электронной почте по адресам, приведенным в списке параметра `ADMINS` настроек проекта (см. *разд. 25.3.3*).

  Если параметру `include_html` присвоить значение `True`, то в письмо будет вложена веб-страница с полным текстом сообщения об ошибке. Значение `False` приводит к отправке обычного сообщения.

  Посредством параметра `email_backend` можно выбрать другой класс, реализующий отправку электронных писем. Список доступных классов такого назначения, равно как и формат значения параметра, приведены в *разд. 25.1*, в описании параметра `EMAIL_BACKEND` настроек проекта.

- `logging.handlers.SMTPHandler` — отправляет сообщения по электронной почте на произвольный адрес. Формат конструктора:

```
SMTPHandler(mailhost=<интернет-адрес SMTP-сервера>,
            fromaddr=<адрес отправителя>,
            toaddrs=<адреса получателей>,
            subject=<тема>[, credentials=None][, secure=None][,
            timeout=1.0])
```

Интернет-адрес SMTP-сервера может быть задан в виде:

- строки — если сервер работает через стандартный TCP-порт;
- кортежа из собственно интернет-адреса и номера TCP-порта — если сервер работает через нестандартный порт.

Адреса получателей указываются в виде списка. *Адрес отправителя* и *тему* отправляемого письма нужно задать в виде строк.

Параметр `credentials` указывает кортеж из имени и пароля для подключения к SMTP-серверу. Если сервер не требует аутентификации, параметр нужно опустить.

Доступные значения для параметра `secure`:

- `None` — если протоколы SSL и TLS не используются (значение по умолчанию);
- "пустой" кортеж — если используется протокол SSL или TLS;
- кортеж из пути к файлу с закрытым ключом;
- кортеж из пути к файлу с закрытым ключом и пути к файлу с сертификатом.

Параметр `timeout` указывает промежуток времени (в виде целого числа в секундах), в течение которого класс-отправитель будет пытаться установить соединение с SMTP-сервером.

Пример:

```
LOGGING = {
    . . .
    'handlers': {
        'file': {
            'class': 'logging.handlers.SMTPHandler',
            'mailhost': 'mail.supersite.ru',
            'fromaddr': 'site@supersite.ru',
            'toaddr': ['admin@supersite.ru',
                       'webmaster@othersite.ru'],
            'subject': 'Проблема с сайтом!',
            'credentials': ('site', 's1i2t3e4'),
        },
    },
    . . .
}
```

- `logging.NullHandler` — вообще не выводит сообщения. Применяется для подавления вывода сообщений определенного уровня.

> ***На заметку***
>
> В данном подразделе были описаны не все доступные классы обработчиков. Более полное описание находится здесь:
> **https://docs.python.org/3/library/logging.handlers.html**.

29.1.6. Регистраторы

Регистраторы занимаются сбором всех сообщений, отправляемых указанными подсистемами Django.

Перечень регистраторов записывается в виде словаря. В качестве ключей его элементов указываются имена регистраторов, а значениями элементов должны быть словари, задающие настройки этих регистраторов.

Django поддерживает следующие регистраторы:

- `django` — собирает сообщения от всех подсистем фреймворка;
- `django.request` — собирает сообщения от подсистемы обработки запросов и формирования ответов. Ответы с кодами статуса `5XX` создают сообщения с уровнем `ERROR`, сообщения с кодами `4XX` — сообщения уровня `WARNING`.

  Объект сообщения, в дополнение к приведенным в *разд. 29.1.2*, получит следующие атрибуты:

  - `status_code` — числовой код статуса ответа;
  - `request` — объект запроса;
- `django.server` — то же самое, что `django.request`, но работает только под отладочным веб-сервером Django;
- `django.template` — собирает сообщения об ошибках, присутствующих в коде шаблонов. Такие сообщения получают уровень `DEBUG`;
- `django.db.backends` — собирает сообщения обо всех операциях с базой данных сайта. Такие сообщения получают уровень `DEBUG`.

  Объект сообщения, в дополнение к приведенным в *разд. 29.1.2*, получит следующие атрибуты:

  - `sql` — SQL-код команды, отправленной СУБД;
  - `duration` — продолжительность выполнения этой команды;
  - `params` — параметры, переданные вместе с этой командой;
- `django.db.backends.schema` — собирает сообщения обо всех операциях, производимых над базой данных в процессе выполнения миграций.

  Объект сообщения, в дополнение к приведенным в *разд. 29.1.2*, получит следующие атрибуты:

- `sql` — SQL-код команды, отправленной СУБД;
- `params` — параметры, переданные вместе с этой командой;

- `django.security.<класс исключения>` — собирает сообщения о возникновении исключений указанного *класса*. Поддерживаются только класс исключения `SuspiciousOperation` и все его подклассы (`DisallowedHost`, `DisallowedModelAdminLookup`, `DisallowedModelAdminToField`, `DisallowedRedirect`, `InvalidSessionKey`, `RequestDataTooBig`, `SuspiciousFileOperation`, `SuspiciousMultipartForm`, `SuspiciousSession` и `TooManyFieldsSent`);
- `django.security.csrf` — собирает сообщения о несовпадении электронных жетонов безопасности, указанных в веб-формах посредством тега `csrf_token`, с ожидаемыми.

Параметры, поддерживаемые всеми регистраторами:

- `handlers` — список обработчиков, которым регистратор будет пересылать собранные им сообщения для вывода;
- `propagate` — если `True`, то регистратор будет передавать собранные сообщения более универсальным регистраторам (обычно это регистратор `django`). Если `False`, то сообщения передаваться не будут. По умолчанию — `False`;
- `level` — минимальный уровень сообщений в виде строкового обозначения. Регистратор станет собирать сообщения, уровень которых не меньше заданного, сообщения меньшего уровня будут отклоняться. Если параметр не указан, регистратор будет собирать сообщения всех уровней.

  У универсального регистратора, принимающего сообщения от регистраторов более специализированных, значение параметра `level`, судя по всему, во внимание не принимается. Следовательно, он собирает сообщения любого уровня;
- `filters` — список фильтров, через которые будут проходить собираемые регистратором сообщения. Чтобы сообщение было воспринято, оно должно пройти через все включенные в список фильтры. Если параметр не указан, фильтры использоваться не будут.

29.1.7. Пример настройки подсистемы журналирования

В листинге 29.1 приведен пример кода, задающего настройки подсистемы журналирования, который можно использовать на практике. Эти настройки записываются в модуле settings.py пакета конфигурации.

Листинг 29.1. Пример настройки диагностических средств Django

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
```

```
    'filters': {
        'require_debug_false': {
            '()': 'django.utils.log.RequireDebugFalse',
        },
        'require_debug_true': {
            '()': 'django.utils.log.RequireDebugTrue',
        },
    },
    'formatters': {
        'simple': {
            'format': '[%(asctime)s] %(levelname)s: %(message)s',
            'datefmt': '%Y.%m.%d %H:%M:%S',
        }
    },
    'handlers': {
        'console_dev': {
            'class': 'logging.StreamHandler',
            'formatter': 'simple',
            'filters': ['require_debug_true'],
        },
        'console_prod': {
            'class': 'logging.StreamHandler',
            'formatter': 'simple',
            'level': 'ERROR',
            'filters': ['require_debug_false'],
        },
        'file': {
            'class': 'logging.handlers.RotatingFileHandler',
            'filename': 'd:/django-site.log',
            'maxBytes': 1048576,
            'backupCount': 10,
            'formatter': 'simple',
        },
    },
    'loggers': {
        'django': {
            'handlers': ['console_dev', 'console_prod'],
        },
        'django.server': {
            'handlers': ['file'],
            'level': 'INFO',
            'propagate': True,
        },
    }
}
```

Все регистраторы, используемые по умолчанию, отключены указанием у параметра `disable_existing_loggers` значения `True`. Фактически проект задействует свои собственные средства журналирования.

Объявлены два фильтра: `require_debug_false`, пропускающий сообщения только в эксплуатационном режиме, и `require_debug_true`, который будет пропускать сообщения только в отладочном режиме.

Форматировщик `simple` выводит сообщения в формате `[`*<временна́я отметка создания>*`] `*<уровень>*`: `*<текст>*.

Обработчиков в нашей конфигурации целых три:

- `console_dev` — выводит в командной строке сообщения любого уровня, прошедшие через фильтр `require_debug_true`, посредством форматировщика `simple`;
- `console_prod` — выводит в командной строке сообщения уровня `ERROR`, прошедшие через фильтр `require_debug_false`, посредством форматировщика `simple`;
- `file` — сохраняет в файл d:\django-site.log сообщения любого уровня посредством форматировщика `simple`. При превышении файлом размера в 1 Мбайт (1 048 576 байт) будет создан новый файл. Всего будет одновременно храниться 10 таких файлов с сообщениями.

Наконец, объявлены два регистратора:

- `django` — универсальный, собирает сообщения из всех подсистем фреймворка и выводит их посредством обработчиков `console_dev` и `console_prod`;
- `django.server` — собирает сообщения уровней `INFO` и выше от подсистемы обработки запросов, когда запущен отладочный веб-сервер, и выводит их через обработчик `file`.

В результате, если включен отладочный режим, в командной строке будут выводиться все сообщения, а при активном эксплуатационном режиме — только сообщения о критических ошибках. А сообщения от подсистемы обработки запросов в любом случае будут дополнительно записываться в файл.

29.2. Средства отладки

29.2.1. Веб-страница сообщения об ошибке

Если при исполнении программного кода произошла ошибка, то Django выведет стандартную веб-страницу с соответствующим сообщением. Верхнюю, наиболее полезную часть этой страницы можно увидеть на рис. 29.1.

Содержимое страницы разделено на отдельные области, представляющие различную информацию:

- Общие сведения об ошибке — наиболее важная, выделенная на странице желтым (на рис. 29.1 — светло-серым) фоном. В ней присутствуют следующие полезные для нас сведения:

- имя класса исключения, возбужденного при возникновении ошибки, и запрошенный клиентом интернет-адрес;
- текстовое описание ошибки;
- **Request Method** — HTTP-метод, посредством которого был выполнен запрос;
- **Request URL** — полный интернет-адрес, запрошенный клиентом, с указанием домена и номера порта;
- **Exception Type** — имя класса исключения;
- **Exception Value** — текстовое описание ошибки;
- **Exception Location** — полный путь к программному файлу, в коде которого была допущена ошибка, с указанием номера строки кода.

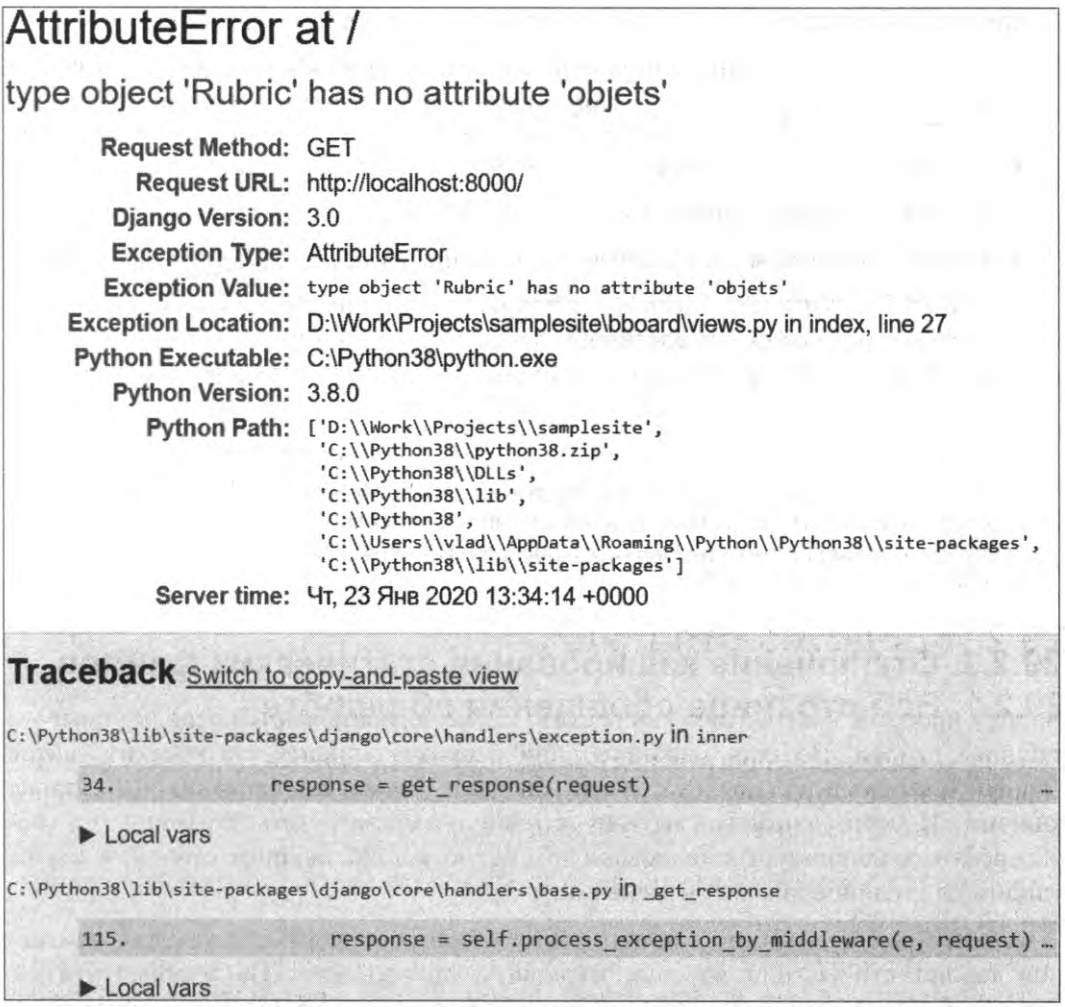


Рис. 29.1. Веб-страница с сообщением об ошибке

Также в этой области выводятся номера версий Django и Python, путь к файлу исполняющей среды Python, список путей, по которым исполняющая среда ищет библиотеки, и время обнаружения ошибки.

- **Traceback** — стек вызовов. Организован в виде набора разделов, в каждом из которых выводятся полный путь к программному файлу, строка исходного кода и список локальных переменных с их значениями.

  Каждая строка исходного кода в таком разделе представляет собой заголовок спойлера (раскрывающейся панели). Если щелкнуть на расположенном в его правой части многоточии, появится фрагмент исходного кода, в котором находится эта строка.

  Во вложенных спойлерах с заголовками **Local vars** перечислены все локальные переменные и их значения.

- **Request information** — сведения о полученном запросе:
  - **USER** — имя текущего пользователя или **AnonymousUser**, если это гость;
  - **GET** — GET-параметры и их значения;
  - **POST** — POST-параметры и их значения;
  - **FILES** — отправленные посетителем файлы;
  - **COOKIES** — cookie и их значения;
  - **META** — значения заголовков запроса, сведения об исполняющей среде Python и операционной системе;
  - **Settings** — настройки проекта.

Внимание!

Описанная здесь страница со сведениями об ошибке выводится только в том случае, если активен отладочный режим. Если сайт переведен в эксплуатационный режим, будет выведена обычная страница с сообщением об ошибке 503 (внутренняя ошибка сервера).

29.2.2. Отключение кэширования статических файлов

Часто в процессе программирования сайта также дорабатываются используемые им таблицы стилей. Но есть проблема: при отправке клиенту статических сайтов Django устанавливает для них очень большое время кэширования на стороне клиента. И если исправить таблицу стилей и перезагрузить страницу, то веб-обозреватель использует кэшированную старую копию таблицы стилей, и мы не увидим на странице никаких изменений.

В качестве решения можно очищать кэш веб-обозревателя после каждого изменения таблиц стилей или вообще отключить кэширование. Но удобнее указать Django, чтобы он запрещал клиенту кэшировать статические файлы. Для этого достаточно выполнить всего три простых действия:

- добавить в модуль urls.py пакета конфигурации такой код (выделен полужирным шрифтом):

```
from django.contrib.staticfiles.views import serve
from django.views.decorators.cache import never_cache

urlpatterns = [
    . . .
]

if settings.DEBUG:
    urlpatterns.append(path('static/<path:path>', never_cache(serve)))
```

Этот код создает маршрут, связывающий шаблонный путь вида **static/*<путь к статическому файлу>*/** с контроллером-функцией `serve()` из модуля `django.contrib.staticfiles.views`, который занимается обработкой запросов к статическим файлам.

У контроллера `serve()` указан декоратор `never_cache()`, запрещающий кэширование на стороне клиента (за подробностями — к *разд. 26.2.4*). Созданный таким образом маршрут добавляется в список маршрутов уровня проекта.

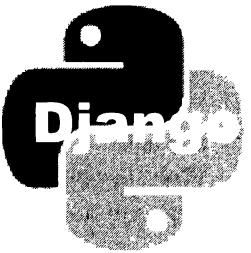
Отметим, что завершающий прямой слеш в шаблонном пути этого маршрута не ставится. Если его все же поставить, то интернет-адреса файлов, ссылки на которые присутствуют в загружаемых таблицах стилей (например, фоновых изображений), будут сформированы некорректно, и эти файлы не загрузятся;

> ***Внимание!***
>
> Здесь предполагается, что в качестве префикса, добавляемого к интернет-адресу статического файла, используется значение по умолчанию: `'/static/'` (этот префикс указывается в параметре `STATIC_URL` настроек проекта и описан в *разд. 11.6.1*). Если в настройках проекта задан другой префикс, шаблонный путь нужно исправить соответствующим образом.
>
> Если сайт работает в эксплуатационном режиме (см. *разд. 3.3.1*), функция `serve()` возбуждает исключение `Http404`.

- обязательно очистить кэш веб-обозревателя. Если этого не сделать, веб-обозреватель продолжит использовать копии статических файлов, сохраненные в своем кэше;
- запустить отладочный веб-сервер с ключом `--nostatic`, отменяющим обработку статических файлов по умолчанию:

```
manage.py runserver --nostatic
```

Как только работа над таблицами стилей сайта будет закончена, рекомендуется вновь включить кэширование статических файлов. Для этого достаточно закомментировать строки, добавленные в модуль urls.py пакета конфигурации на первом шаге, и перезапустить отладочный веб-сервер уже без ключа `--nostatic`.



ГЛАВА 30

Публикация веб-сайта

Разработка веб-сайта — процесс долгий и по-своему увлекательный. Но рано или поздно он подходит к концу. Сайт написан, проверен, возможно, наполнен какими-либо рабочими данными — и теперь его предстоит опубликовать в Сети.

30.1. Подготовка веб-сайта к публикации

Перед публикацией веб-сайта предварительно нужно выполнить некоторые подготовительные работы.

30.1.1. Написание шаблонов веб-страниц с сообщениями об ошибках

Эти шаблоны будут применяться для генерирования страниц с сообщениями об ошибках при работе в эксплуатационном режиме:

- 404.html — шаблон страницы с сообщением об ошибке с кодом статуса 404 (запрошенная страница отсутствует). Обычно такая страница содержит текст вида "Страница не найдена" и гиперссылку на главную страницу сайта.

  Служебный контроллер, выводящий эту страницу, создает в контексте шаблона две переменные:

  - `request_path` — путь, выделенный из интернет-адреса, который был получен в составе запроса;
  - `exception` — строка с текстом сообщения об отсутствии запрошенной страницы.

  Помимо этого, шаблон 404.html имеет доступ ко всем переменным, добавленным в контекст шаблона зарегистрированными обработчиками контекста (о них рассказывалось в *разд. 11.1*);

- 500.html — шаблон страницы с сообщением об ошибке 500 (внутренняя ошибка сервера). Обычно такая страница содержит текст "Внутренняя ошибка сервера" и предложение попытаться обновить страницу спустя некоторое время.

Служебный контроллер, выводящий эту страницу, передает шаблонизатору пустой контекст шаблона без каких-либо переменных;

- 403.html — шаблон страницы с сообщением об ошибке с кодом статуса 403 (доступ к запрошенной странице запрещен. В частности, эта ошибка возникает при обращении гостя к странице, к которой имеют доступ только зарегистрированные пользователи). Обычно такая страница содержит текст вида "Страница недоступна", предложение выполнить процедуру входа на сайт и гиперссылки на страницу входа и главную страницу сайта.

  Служебный контроллер, выводящий эту страницу, создает в контексте шаблона переменную `exception`, в которой хранится строка с текстом сообщения о недоступности запрошенной страницы;

- 400.html — шаблон страницы с сообщением об ошибке 400 (клиентский запрос некорректно сформирован). Обычно такая страница содержит текст вида "Некорректный запрос".

  Служебный контроллер, выводящий эту страницу, передает шаблонизатору пустой контекст шаблона без каких-либо переменных.

Все эти шаблоны помещаются непосредственно в папку templates пакета приложения или в одну из папок, чей путь указан в параметре `DIRS` настроек текущего шаблонизатора (см. *разд. 11.1*).

> ***Внимание!***
>
> Стандартное приложение `django.contrib.admin` (административный веб-сайт) содержит в своем составе шаблоны *404.html* и *500.html*. Чтобы наш сайт использовал созданные нами шаблоны, а не принадлежащие этому приложению, мы можем прибегнуть к переопределению шаблонов (см. *разд. 19.3*).

Если какой-либо из упомянутых шаблонов отсутствует, то Django отправит клиенту пустой ответ с кодом статуса, соответствующим возникшей ошибке. В результате веб-обозреватель выведет встроенную в него страницу с описанием ошибки.

30.1.2. Указание настроек эксплуатационного режима

Следующий шаг — указание настроек проекта, которые будут действовать в эксплуатационном режиме:

- `DEBUG` — этому параметру, указывающему режим работы сайта, нужно присвоить значение `False`, задающее эксплуатационный режим;
- `ALLOWED_HOSTS` — очень важный параметр, указывающий перечень хостов, с которых Django будет принимать отправленные клиентами данные. Если сайт получит данные с хоста, отсутствующего в этом перечне, то он возбудит исключение `SuspiciousOperation` из модуля `django.core.exceptions`, что приведет к выдаче страницы с сообщением об ошибке 400 (некорректно сформированный запрос).

Перечень должен быть представлен в виде списка, элементами которого должны быть строки, указывающие разрешенные хосты. Эти строки могут быть:

- доменными именами;
- IP-адресами в формате IPv4 или IPv6;
- шаблонами доменных имен. В таких шаблонах можно применять специальный символ * (звездочка), который обозначает произвольное количество любых знаков.

Пример:

```
ALLOWED_HOSTS = ['www.supersite.ru', 'blog.supersite.ru',
                                     '*.shop.supersite.ru']
```

Здесь в список разрешенных занесены хосты **www.supersite.ru**, **blog.supersite.ru** и все хосты вида ***<произвольные символы>*.shop.supersite.ru** (**technics.shop.supersite.ru**, **furniture.shop.supersite.ru** и т. п.).

Значения этого параметра по умолчанию:

- в отладочном режиме — список `['localhost', '127.0.0.1', '[::1]']` (т. е. локальный хост, представленный доменным именем и IP-адресами стандартов IPv4 и IPv6);
- в эксплуатационном режиме — "пустой" список. Поэтому перед запуском сайта в эксплуатацию этот параметр обязательно следует задать, иначе сайт работать не будет;

- `DATABASES` — необходимо указать параметры базы данных, которая будет использоваться сайтом в эксплуатационном режиме (подробнее об их указании рассказывалось в *разд. 3.3.2*). Поскольку сайт, как правило, публикуется на компьютере, отличном от того, на котором он разрабатывался, параметры базы данных там, скорее всего, будут иными;
- `STATIC_ROOT` — возможно, понадобится изменить путь к папке, в которой хранятся статические файлы сайта (за подробностями — к *разд. 11.6*);
- `MEDIA_ROOT` — возможно, понадобится изменить путь к папке, в которой хранятся выгруженные файлы (см. *главу 20*);
- настройки подсистемы отправки электронных писем — следует изменить их на те, что будут использоваться сайтом в режиме эксплуатации (описание этих параметров см. в *разд. 25.1*);
- `CACHES` — следует указать параметры подсистемы кэширования уровня сервера (см. *главу 26*), которая будет использоваться при эксплуатации сайта;
- `LOGGING` — понадобится задать окончательные настройки для подсистемы диагностики (она была описана в *разд. 29.1*);
- `ADMINS` — здесь нужно задать перечень адресов электронной почты, принадлежащих администраторам;

- MANAGERS — и адреса редакторов.

  Как указываются перечни электронных адресов администраторов и редакторов, было рассказано в *разд. 25.3.3*;

- SECRET_KEY — не помешает удостовериться, что секретный ключ, задаваемый этим параметром, кроме данного сайта, не применяется более нигде.

30.1.3. Удаление ненужных данных

Часть данных, генерируемых работающим Django-сайтом, либо являются временными (устаревшие CAPTCHA, сессии и пр.), либо впоследствии могут быть созданы повторно (например, миниатюры). Перед публикацией сайта такие данные лучше удалить для уменьшения его объема, особенно если сайт будет переноситься на целевой компьютер по сети.

Вот список команд утилиты manage.py, служащих для удаления ненужных и временных данных:

- captcha_clean — удаляет просроченные CAPTCHA из хранилища (подробности — в *разд. 17.4.4*);
- thumbnail_cleanup — удаляет файлы с миниатюрами: все или сгенерированные в течение указанного количества дней (см. *разд. 20.6.5*);
- clearsessions — удаляет устаревшие сессии (см. *разд. 23.2.3*).

30.1.4. Окончательная проверка веб-сайта

По окончании подготовительных работ неплохо выполнить проверку, всё ли мы сделали как надо. Провести ее нам поможет команда check утилиты manage.py:

```
manage.py check [<псевдоним приложения 1> <псевдоним приложения 2> . . .
<псевдоним приложения n>] [--tag|-t <группа проверок>] [--list-tags]
[--deploy] [--fail-level <уровень неполадки>]
```

По умолчанию выполняется проверка всех приложений, имеющихся в проекте. Но можно задать проверку только приложений с заданными *псевдонимами*, перечислив их через пробел. Пример:

```
manage.py check bboard testapp restapi
```

Поддерживаемые командные ключи:

- --tag или -t — указывает *группу проверок*, которые необходимо провести. Доступны следующие *группы*:
  - admin — всё связанное с административным веб-сайтом Django (редакторы, обычные и встроенные, действия и др.);
  - caches — настройки подсистемы кэширования;
  - compatibility — потенциальные проблемы при переходе на следующую версию Django;

- `database` — настройки используемых баз данных;
- `models` — объявления моделей, диспетчеров записей и наборов записей;
- `security` — настройки безопасности;
- `signals` — объявления сигналов и привязка к ним обработчиков;
- `staticfiles` — настройки подсистемы, обрабатывающей статические файлы;
- `templates` — настройки шаблонизаторов;
- `urls` — списки маршрутов.

Пример:

```
manage.py check --tag urls
```

Можно указать произвольное количество *групп проверок* — каждую в отдельном ключе:

```
manage.py check --tag database --tag staticfiles --tag urls
```

Если ключ не задан, выполняется проверка по всем группам, за исключением `database`;

- `--list-tags` — выводит список всех поддерживаемых групп проверок;
- `--deploy` — выполняет дополнительные проверки, актуальные только для сайтов, предназначенных для публикации;
- `--fail-level` — указывает *уровень* найденной *неполадки*, после которого проверка прекращается. Доступны уровни неполадок `DEBUG`, `INFO`, `WARNING`, `ERROR` и `CRITICAL`. Если ключ не указан, то проверка завершается по выявлении неполадки уровня `ERROR`.

30.1.5. Настройка веб-сайта для работы по протоколу HTTPS

Любой Django-сайт может работать по защищенному протоколу HTTPS без какого-либо дополнительного конфигурирования. Однако настоятельно рекомендуется указать в модуле settings.py пакета конфигурации следующие настройки, затрагивающие безопасность сайта и защиту от сетевых атак:

- `SECURE_SSL_REDIRECT` — если `True`, то сайт при попытке доступа к нему по незащищенному протоколу HTTP станет выполнять перенаправление по тому же интернет-адресу, но с использованием протокола HTTPS (по умолчанию — `False`).

  Если сайт должен работать исключительно по протоколу HTTPS, то необходимо установить этот параметр в `True`;

- `SECURE_REDIRECT_EXEMPT` — задает перечень шаблонных путей, которые должны быть доступны по протоколу HTTP (соответственно, сайт не будет выполнять перенаправление с применением HTTPS при запросах по этим путям). Перечень

задается в виде списка, каждый элемент которого указывает отдельный путь в виде регулярного выражения. Шаблонные пути не должны содержать начального слеша. Пример:

```
SECURE_REDIRECT_EXEMPT = [r'^no-ssl/$', r'^public/$']
```

Принимается во внимание, только если параметру `SECURE_SSL_REDIRECT` задано значение `True`.

Значение по умолчанию — "пустой" список;

- `SECURE_SSL_HOST` — задает интернет-адрес хоста, на который сайт будет выполнять перенаправление с использованием HTTPS. Если `None`, перенаправление будет выполняться на изначальный хост.

  Принимается во внимание, только если параметру `SECURE_SSL_REDIRECT` задано значение `True`.

  Значение по умолчанию — `None`;

- `SECURE_HSTS_SECONDS` — если задано значение, отличное от `0`, сайт будет вставлять в каждый отправляемый ответ заголовок формата:

  ```
  Strict-Transport-Security: max-age=<время>
  ```

  Одним фактом своего наличия в ответе такой заголовок сообщает веб-обозревателю, что сайт доступен исключительно по протоколу HTTPS, и при попытке получить к нему доступ по HTTP веб-обозревателю следует самостоятельно выполнить перенаправление с применением HTTPS.

  Языковая конструкция `max-age`, содержащаяся в значении этого заголовка, задает *время* в секундах, в течение которого веб-обозреватель должен "помнить", что сайт доступен исключительно по протоколу HTTPS. В качестве этого *времени* указывается значение параметра `SECURE_HSTS_SECONDS`.

  Значение по умолчанию: `0` (упомянутый ранее заголовок в ответы не вставляется).

  Этот параметр нужно указывать, если сайт должен работать исключительно по протоколу HTTPS, чтобы усилить защиту от сетевых атак. Сначала в целях проверки работоспособности имеет смысл задать относительно небольшое значение, например `3600` (1 час), а потом, удостоверившись, что сайт полностью работоспособен, увеличить его, скажем, до `31536000` (1 года);

- `SECURE_HSTS_INCLUDE_SUBDOMAINS` — если `True`, то сайт будет добавлять в значение заголовка `Strict-Transport-Security` конструкцию `includeSubDomains`:

  ```
  Strict-Transport-Security: max-age=<время> includeSubDomains
  ```

  Она предписывает веб-обозревателям блокировать доступ по HTTP также и к поддоменам.

  Значение по умолчанию — `False` (не добавлять конструкцию `includeSubDomains`).

  Этот параметр принимается во внимание, если параметру `SECURE_HSTS_SECONDS` задано значение, отличное от `0`;

- SECURE_HSTS_PRELOAD — если True, то сайт будет добавлять в значение заголовка Strict-Transport-Security конструкцию preload:

  Strict-Transport-Security: max-age=<*время*> preload

  Она предписывает веб-обозревателю уведомлять веб-службы Google, что текущий сайт либо находится в статическом списке безопасных сайтов, поддерживаемом упомянутой ранее корпорацией, либо является кандидатом на включение туда.

  Значение по умолчанию — False (не добавлять конструкцию preload).

  Параметрам SECURE_HSTS_INCLUDE_SUBDOMAINS и SECURE_HSTS_PRELOAD можно одновременно дать значение True. В таком случае в ответы будет добавляться заголовок:

  Strict-Transport-Security: max-age=<*время*> includeSubDomains preload

- SECURE_CONTENT_TYPE_NOSNIFF — если True, то сайт будет вставлять в каждый отправляемый ответ заголовок:

  X-Content-Type-Options: nosniff

  Он запрещает веб-обозревателю определять тип загруженного файла по его содержимому, а, наоборот, предписывает всегда использовать тип, заданный в заголовке Content-Type полученного ответа.

  Значение по умолчанию, начиная с Django 3.0, — True (в более старых версиях — False, указывающее не вставлять в ответы такой заголовок).

  Установка значения True позволяет предотвратить некоторые типы сетевых атак, связанных с загрузкой клиентом небезопасных файлов (например, веб-страниц с вредоносными веб-сценариями), замаскированных под безопасные (например, изображения или архивы);

- SECURE_BROWSER_XSS_FILTER — если True, то сайт будет вставлять в каждый ответ заголовок:

  X-XSS-Protection: 1; mode=block

  Он указывает веб-обозревателю блокировать любой пришедший с сервера ответ, содержащий веб-сценарии, которые могут оказаться вредоносными (т. е. использоваться для атак *XSS* — Cross-Site Scripting, межсайтовый скриптинг).

  Значение по умолчанию — False (не вставлять такой заголовок);

- SECURE_REFERRER_POLICY (начиная с Django 3.0) — если задано значение, отличное от None, то сайт будет помещать в каждый ответ заголовок Referrer-Policy, указывающий веб-обозревателю, вставлять ли при переходах на другую страницу в запросы заголовок Referrer с интернет-адресом предыдущей страницы. Доступны следующие значения параметра:

  - "no-referrer" — веб-обозреватель не должен вставлять в запросы заголовок Referrer;

- "no-referrer-when-downgrade" — вставлять этот заголовок только в том случае, если выполняется переход на сайт, работающий через HTTPS;
- "origin" — вставлять заголовок, но отправлять в нем интернет-адрес хоста, а не страницы;
- "origin-when-cross-origin" — вставлять заголовок, но отправлять в нем интернет-адрес страницы только при переходе на страницу того же сайта, в противном случае отправлять интернет-адрес хоста;
- "same-origin" — вставлять заголовок с интернет-адресом страницы только при переходе на страницу того же сайта, в противном случае не вставлять этот заголовок;
- "strict-origin" — вставлять заголовок, но отправлять в нем интернет-адрес хоста и только при переходе на сайт, работающий через HTTPS, в противном случае не вставлять заголовок;
- "strict-origin-when-cross-origin" — вставлять заголовок, отправлять в нем интернет-адрес страницы при переходе на страницу того же сайта, работающего через HTTPS, интернет-адрес хоста — при переходе на страницу другого сайта, также работающего через HTTPS, и вообще не вставлять заголовок в остальных случаях;
- "unsafe-url" — всегда вставлять заголовок с интернет-адресом страницы;
- None — заголовок Referrer-Policy вообще не будет вставляться в ответы.

Например, при указании параметра:

```
SECURE_REFERRER_POLICY = 'same-origin'
```

сайт будет отправлять в ответах заголовок:

```
Referrer-Policy: same-origin
```

В параметре можно указать сразу несколько значений — на тот случай, если какое-то из них не будет "знакомо" веб-обозревателю:

- либо в одной строке, перечислив их через запятую:

```
SECURE_REFERRER_POLICY = 'no-referrer, same-origin'
```

- либо в виде списка или кортежа:

```
SECURE_REFERRER_POLICY = ('no-referrer', 'same-origin')
```

Последнее значение из перечисленных будет трактоваться как предпочтительное.

Значение по умолчанию — None;

- CSRF_COOKIE_SECURE — если True, то электронные жетоны в веб-формах для идентификации получаемых данных будут пересылаться в подписанных cookie (по умолчанию — False).

Этим параметрам также нужно задать значение True, чтобы обезопасить сайт и его посетителей от сетевых атак;

- SESSION_COOKIE_SECURE — этот параметр уже рассматривался в *разд. 23.2.1*. Ему нужно задать значение True, чтобы cookie сессий загружались только по протоколу HTTPS;
- X_FRAME_OPTIONS — указывает, разрешает ли сайт веб-обозревателям открывать свои страницы во фреймах. Доступны два строковых значения:
  - "SAMEORIGIN" — разрешается открывать страницы только во фреймах, что находятся на страницах того же сайта;
  - "DENY" — полный запрет на открытие страниц текущего сайта во фреймах.

  Значение по умолчанию, начиная с Django 3.0, — "DENY" (в более старых версиях — "SAMEORIGIN").

  Указывать значение "SAMEORIGIN" этого параметра следует только в случаях, если на страницах сайта не заносится какая-либо важная информация (например, номер кредитной карты);
- SECURE_PROXY_SSL_HEADER — задает пару "заголовок-значение", чье присутствие в запросе указывает на то, что запрос был выполнен по защищенному протоколу HTTPS. Значение параметра задается в виде кортежа из двух строковых элементов: первый элемент укажет заголовок (присутствующие в нем дефисы следует заменить подчеркиваниями, а сам заголовок — предварить символами HTTP_), а второй элемент — значение. Пример:

  ```
  SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'secured')
  ```

  В этом случае присутствие в полученном запросе заголовка:

  ```
  X-Forwarded-Proto: secured
  ```

  сообщит Django о том, что запрос пришел по защищенному протоколу.

  Значение по умолчанию: None (Django будет выяснять, пришел ли запрос по защищенному протоколу, проверяя, присутствует ли в начале интернет-адреса обозначение **https://**).

  Этот параметр необходимо указывать, если Django-сайт соединяется с Интернетом через прокси-сервер. В таком случае Django не сможет определить, по какому протоколу клиент соединился с прокси-сервером: например, если сайт и прокси-сервер соединяются по протоколу HTTP, то даже при получении от клиента запроса по HTTPS Django будет "считать", что соединение небезопасно.

  Помимо указания параметра SECURE_PROXY_SSL_HEADER, необходимо сконфигурировать прокси-сервер таким образом, чтобы, получив запрос по незащищенному протоколу, он вырезал из запроса указанный в параметре SECURE_PROXY_SSL_HEADER заголовок, а получив запрос по защищенному протоколу — добавлял его. Однако в таком случае необходимо иметь доступ к прокси-серверу и возможность задавать его настройки.

30.2. Публикация веб-сайта

30.2.1. Публикация посредством Uvicorn

Uvicorn — "легкий" и быстрый веб-сервер, который специально предназначен для публикации сайтов, написанных на Python, в том числе и с применением Django.

> ***Внимание!***
> Uvicorn может обслуживать только сайты, написанные на Django 3.0 или более новых версиях этого фреймворка.

Преимущество Uvicorn в том, что для подготовки сайта к публикации с его помощью достаточно добавить в код всего несколько выражений. Недостаток — невысокая производительность, вследствие чего этот сервер не стоит применять для обслуживания высоконагруженных решений.

> ***На заметку***
> Полная документация по Uvicorn находится здесь: **https://www.uvicorn.org/**.

Установка Uvicorn выполняется подачей команды:

```
pip install uvicorn
```

Помимо веб-сервера, будут установлены библиотеки websockets, click и h11, необходимые для работы.

30.2.1.1. Подготовка веб-сайта к публикации посредством Uvicorn

Веб-сайт, написанный с применением Django, успешно работает под управлением Uvicorn. За одним исключением: этот веб-сервер не обрабатывает статические и выгруженные файлы. Поэтому в код сайта необходимо внести некоторые правки.

В список маршрутов уровня проекта (что хранится в модуле urls.py пакета конфигурации) следует добавить два маршрута — для обработки статических и выгруженных файлов. Оба маршрута создаются вызовом функции `path()`, описанной в *разд. 8.2*. Различаются они только контроллером-функцией, который указывается во втором параметре этой функции.

- Обработка статических файлов — будет осуществляться контроллером `serve()` из модуля `django.contrib.staticfiles.views`. Этот контроллер "умеет" искать статические файлы во всех папках, указанных в настройках проекта (о настройках подсистемы статических файлов рассказывалось в *разд. 11.6.1*).

  К сожалению, контроллер `serve()` работает только в отладочном режиме — в эксплуатационном он возбуждает исключение `Http404`. Но, к счастью, он поддерживает необязательный параметр `insecure`: если задать ему значение `True`, то контроллер успешно работает и в эксплуатационном режиме. Передать значения для параметров контроллера можно, указав их в словаре, который передается функции `path()` в третьем параметре.

- Обработка выгруженных файлов — будет выполняться контроллером-функцией `serve()` из модуля `django.views.static` (не перепутайте с одноименным контроллером из модуля `django.contrib.staticfiles.views`!). Он более универсален, нежели описанный ранее, и может выдавать файлы из произвольной папки, путь к которой передается ему через необязательный параметр `document_root`. Значение этого параметра можно передать также в словаре, указываемом в третьем параметре функции `path()`.

Пример задания обоих путей в модуле urls.py пакета конфигурации:

```
from django.contrib.staticfiles.views import serve
from django.views.static import serve as media_serve
from django.conf import settings

urlpatterns = [
    . . .
]

if not settings.DEBUG:
    urlpatterns.append(path('static/<path:path>', serve,
                                                  {'insecure': True}))
    urlpatterns.append(path('media/<path:path>', media_serve,
                            {'document_root': settings.MEDIA_ROOT}))
```

30.2.1.2. Запуск и остановка Uvicorn

Чтобы запустить Uvicorn, следует перейти в папку проекта и задать в командной строке команду формата:

```
uvicorn <имя пакета конфигурации>.asgi:application
[--port <номер TCP-порта>]
[--no-access-log] [--ssl-keyfile <путь к файлу с закрытым ключом>]
[--ssl-certfile <путь к файлу сертификата>]
```

Uvicorn "общается" с Django-сайтом через интерфейс *ASGI* (Asynchronous Server Gateway Interface, асинхронный интерфейс серверного шлюза), поддержка которого появилась в Django 3.0. "Связкой" между Uvicorn и сайтом выступает модуль asgi.py пакета конфигурации; `application` — это переменная модуля asgi.py, хранящая объект, который представляет сайт.

Поддерживаются следующие наиболее полезные ключи:

- `--port` — указывает *номер TCP-порта*, через который будет работать веб-сервер (по умолчанию: 8000);
- `--no-access-log` — запрещает выводить журнал работы сервера (по умолчанию журнал выводится непосредственно в командной строке).

Пример запуска Uvicorn для обслуживания сайта через стандартный TCP-порт № 80 без вывода журнала:

```
uvicorn samplesite.asgi:application --port 80 --no-access-log
```

Чтобы запустить Uvicorn для работы через защищенный протокол HTTPS, следует дополнительно задать параметры `--ssl-keyfile` и `--ssl-certfile`, указывающие пути к файлам соответственно закрытого ключа и сертификата. Пример:

```
uvicorn samplesite.asgi:application --port 443 ↳
--ssl-keyfile c:\keys\samplesitesite.pki ↳
--ssl-keyfile c:\keys\samplesitesite.crt
```

Чтобы остановить Uvicorn, достаточно переключиться в окно командной строки, в которой он запущен, и нажать комбинацию клавиш <Ctrl>+<Break>.

30.2.2. Публикация посредством Apache HTTP Server

Apache HTTP Server — высокопроизводительный, универсальный и популярнейший на данный момент веб-сервер.

При публикации сайта с его помощью обработка статических и выгруженных файлов будет выполняться самим веб-сервером, не затрагивая Django. Если сайт содержит много файлов такого рода, это заметно повысит производительность.

> ***Внимание!***
>
> К сожалению, из-за ошибки в одном из модулей стандартной библиотеки Python 3.8 (автор протестировал версии 3.8.0 и 3.8.1) сайт, публикуемый с помощью Apache HTTP Server, не работает. Возможно, в более новых версиях Python эта ошибка будет устранена.
>
> Однако под более старыми версиями Python (автор использовал 3.7.6) сайт полностью функционирует.

30.2.2.1. Подготовка веб-сайта к публикации посредством Apache HTTP Server

Подготовка сайта к публикации включает три шага:

1. Создание папки, в которой будут собраны все статические файлы, имеющиеся в составе Django-сайта. Обычно этой папке дают имя static.
2. Указание пути к этой папке в параметре `STATIC_ROOT` настроек проекта. Вот пример задания пути к папке static, находящейся непосредственно в папке проекта:

   ```
   STATIC_ROOT = os.path.join(BASE_DIR, 'static')
   ```

3. Сбор всех статических файлов в указанной папке подачей команды `collectstatic` утилиты manage.py:

   ```
   manage.py collectstatic [--ignore|-i <шаблон>] [--clear|-c]
   [--link|-l] [--noinput|--no-input] [--no-default-ignore] [--dry-run]
   ```

   По умолчанию все статические файлы, найденные в папках static пакетов приложений и папках, пути к которым указаны в параметре `STATICFILES_DIRS`, копируются в папку, заданную в параметре `STATIC_ROOT` настроек проекта. Структура папок, в которые вложены эти файлы, при этом сохраняется.

Приложения просматриваются в том порядке, в котором они перечислены в списке зарегистрированных в проекте. Если в разных папках static присутствуют файлы с одним и тем же именем, будет использован файл, найденный первым.

При последующих вызовах команды `collectstatic` в папку будут скопированы только новые и изменившиеся после предыдущего копирования файлы. Перед перезаписью имеющегося в папке назначения файла утилита выдаст предупреждение и предложит ввести слово "yes" для перезаписи или "no" для отказа от этого.

По умолчанию копируются все статические файлы, за исключением файлов с именами CVS, а также именами, которые совпадают с шаблонами .* и *~.

Поддерживаемые командные ключи:

- `--ignore` или `-i` — указывает *шаблон* для имен файлов, которые не должны копироваться:

  ```
  manage.py collectstatic --ignore *.tmp
  ```

  Можно задать произвольное количество таких *шаблонов* — каждый в своем ключе:

  ```
  manage.py collectstatic --ignore *.tmp --ignore *.bak
  ```

  Начиная с Django 2.2, можно указывать шаблоны имен папок, содержимое которых не должно копироваться;

- `--clear` или `-c` — перед началом копирования очистить папку назначения;
- `--link` или `-l` — вместо копирования файла создать символическую ссылку на него;
- `--noinput` или `--no-input` — имеющийся в папке назначения файл будет перезаписан без выдачи предупреждения;
- `--no-default-ignore` — также копировать файлы с именами CVS, .* и *~;
- `--dry-run` — выводит на экран сведения о файлах, подлежащих копированию, но не копирует их.

  Внимание!

  Статических файлов в папке назначения может оказаться довольно много, особенно если в проекте используются сложные приложения и библиотеки, наподобие административного веб-сайта Django и Django REST framework. Поэтому во многих случаях имеет смысл рассмотреть вариант с созданием символических ссылок на статические файлы вместо их копирования.

Также может оказаться полезной команда `findstatic` утилиты manage.py, которая ищет статические файлы с указанными *именами* и выводит на экран полные пути к ним:

```
manage.py findstatic <имя файла 1> <имя файла 2> . . . <имя файла n>
[--first]
```

Если задать командный ключ `--first`, то будет выведен только путь к первому обнаруженному файлу.

30.2.2.2. Подготовка платформы для публикации посредством Apache HTTP Server

Подготовка платформы для публикации сайта с помощью Apache HTTP Server заключается в выполнении следующих шагов:

1. Поиск и загрузка программного модуля mod_wsgi, выступающего в качестве коннектора между веб-сервером и Django-сайтом.

   Дистрибутивные комплекты различных редакций этого модуля находятся по интернет-адресу **https://www.lfd.uci.edu/~gohlke/pythonlibs/#mod_wsgi**. Имена файлов с этими комплектами включают в себя следующие фрагменты символов:

   - `ap<две цифры>` — где *две цифры* обозначают версию Apache;
   - `vc<одна или две цифры>` — *одна или две цифры* показывают внутреннюю версию среды разработки Microsoft Visual C++, в которой компилировался модуль-коннектор;
   - `cp<две цифры>` — *две цифры* обозначают версию Python;
   - `win32` — если это 32-разрядная редакция модуля;
   - `win_amd64` — если это 64-разрядная редакция модуля.

   Например, mod_wsgi-4.7.0+ap24vc14-cp36-cp36m-win32.whl — это редакция для 32-разрядного Python 3.6 и Apache 2.4, откомпилированного в Microsoft Visual C++ версии VC14.

   Следует выбрать ту редакцию mod_wsgi, которая соответствует версии и редакции установленного на платформе Python. Так, если установлен 64-разрядный Python 3.7, следует загрузить файл mod_wsgi-4.7.0+ap24vc15-cp37-cp37m-win_amd64.whl.

   С расширением whl сохраняются файлы формата WHL (от англ. wheel) — дистрибутивные пакеты дополнительных библиотек для Python.

2. Установка модуля из загруженного файла WHL отдачей команды формата:

   ```
   pip install <путь к WHL-файлу с дистрибутивом>
   ```

3. Загрузка и установка веб-сервера Apache HTTP Server. Найти его дистрибутивный комплект можно на сайте **https://www.apachelounge.com/** или **https://www.apachehaus.com/**.

   > ***На заметку***
   >
   > Полная документация по Apache находится на его домашнем сайте: **http://httpd.apache.org/**.

   При выборе дистрибутива сервера необходимо учесть две очень важные вещи:

   - Следует выбрать ту редакцию, которая откомпилирована в той же версии Microsoft Visual C++, что и установленный ранее модуль-коннектор mod_wsgi. Так, если был установлен модуль mod_wsgi-4.7.0+ap24vc15-cp37-

cp37m-win_amd64.whl, откомпилированный в Microsoft Visual C++ версии VC15, следует выбрать дистрибутив Apache 2.4.x OpenSSL 1.1.1 VC15.

- В случае установки 32-разрядной редакции Python нужно выбирать только 32-разрядную редакцию Apache, а в случае 64-разрядной редакции Python — только 64-разрядную редакцию Apache.

Внимание!

Если установить не подходящие друг другу редакции Python, mod_wsgi и Apache, скорее всего, ничего не заработает.

Дистрибутив Apache поставляется в виде обычного архива формата ZIP, содержимое которого следует распаковать в корневую папку диска.

4. Получение настроек, необходимых для связывания веб-сервера и публикуемого сайта посредством mod_wsgi. Для этого следует задать в командной строке команду:

```
mod_wsgi-express module-config
```

Утилита mod_wsgi-express.exe устанавливается в составе модуля mod_wsgi, а команда `module-config` этой утилиты выведет на экран строки, содержащие необходимые настройки. Например, автор книги получил такие строки:

```
LoadFile "c:/python37/python37.dll"
LoadModule wsgi_module ↳
"c:/python37/lib/site-packages/mod_wsgi/server/↳
mod_wsgi.cp37-win_amd64.pyd"
WSGIPythonHome "c:/python37"
```

Первая строка предписывает веб-серверу при запуске загрузить программное ядро исполняющей среды Python, необходимое для успешной работы mod_wsgi. Вторая строка выполняет загрузку самого этого модуля-коннектора. Третья строка указывает коннектору путь, по которому установлен Python.

5. Добавление полученных ранее строк с настройками в файл конфигурации Apache. Этот файл носит имя httpd.conf и находится по пути *<путь, по которому установлен Apache>*\conf.

30.2.2.3. Конфигурирование веб-сайта для работы под Apache HTTP Server

В файл конфигурации httpd.conf также нужно внести параметры самого публикуемого Django-сайта. Они указываются посредством следующих директив:

- `Alias` *<префикс интернет-адреса статических файлов>* ↳
  "*<путь к папке статических файлов>*"

Директива `Alias` указывает серверу искать файлы, интернет-адреса которых имеют заданный *префикс*, в папке с указанным *путем*. После ее указания веб-сервер станет сам обслуживать статические файлы, минуя Django.

- ```
  <Directory "<путь к папке статических файлов>">
      Require all granted
  </Directory>
  ```

  Директива `Directory` устанавливает права Apache на содержимое папки с заданным *путем*. Команда `Require all granted` дает веб-серверу доступ ко всем файлам из этой папки. Если данную директиву не указать, то сервер не сможет загрузить ни один файл из папки.

- ```
  Alias <префикс интернет-адреса выгруженных файлов> ↳
  "<путь к папке выгруженных файлов>"
  <Directory "<путь к папке выгруженных файлов>">
      Require all granted
  </Directory>
  ```

  Аналогичным образом конфигурируется обработка файлов, выгруженных посетителями.

- ```
  WSGIScriptAlias <префикс для сайта> ↳
  "<путь к папке пакета конфигурации проекта>/wsgi.py"
  ```

  Эта директива свяжет указанный *префикс* с самим Django-сайтом. Например, если задать префикс **/bboard**, то сайт станет доступен по интернет-адресу **http://*<интернет-адрес хоста>*/bboard/**. Чтобы связать сайт с "корнем", следует задать префикс / (слеш).

  Модуль wsgi.py, хранящийся в пакете конфигурации проекта, служит "связкой" между mod_wsgi и сайтом. Взаимодействие между ними выполняется через интерфейс WSGI (Web Server Gateway Interface, интерфейс шлюза веб-сервера), поддерживаемый Django начиная с самых первых версий.

- ```
  WSGIPythonPath "<путь к папке проекта>"
  ```

  Очень важная директива, задающая *путь к папке проекта*, чтобы исполняющая среда Python "знала", где находятся все пакеты и модули, составляющие сайт;

- ```
  <Directory "<путь к папке пакета конфигурации проекта>">
      <Files wsgi.py>
          Require all granted
      </Files>
  </Directory>
  ```

  Эта директива разрешает веб-серверу доступ к модулю wsgi.py. Если ее не указать, то Apache не сможет запустить этот файл;

- ```
  WSGIPassAuthorization On
  ```

  Эта директива указывает модулю-коннектору пропускать любые заголовки, содержащие имя и пароль пользователя, а не удалять их, как это он делает по умолчанию. Ее следует указать, если в состав сайта входит веб-служба, обрабатывающая AJAX-запросы и реализующая аутентификацию.

В листинге 30.1 приведен готовый код, конфигурирующий Django-сайт. Его можно использовать как шаблон для написания своей конфигурации.

Листинг 30.1. Пример конфигурации Django-сайта

```
Alias /static/ "c:/sites/samplesite/static/"
Alias /media/ "c:/sites/samplesite/media/"

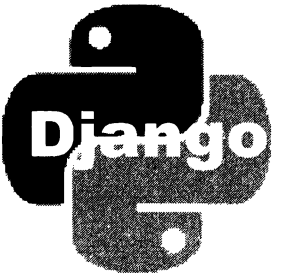
<Directory "c:/sites/samplesite/static/">
    Require all granted
</Directory>

<Directory "c:/sites/samplesite/media/">
    Require all granted
</Directory>

WSGIScriptAlias / "c:/sites/samplesite/samplesite/wsgi.py"
WSGIPythonPath "c:/sites/samplesite"

<Directory "c:/sites/samplesite/samplesite">
    <Files wsgi.py>
        Require all granted
    </Files>
</Directory>

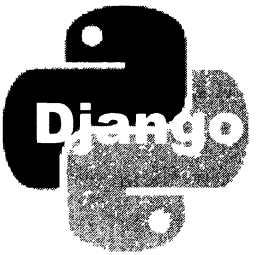
WSGIPassAuthorization On
```



ЧАСТЬ IV

Практическое занятие: разработка веб-сайта

ГЛАВА 31



Дизайн. Вспомогательные веб-страницы

На протяжении трех частей книги мы изучали теорию, "разбавляя" ее небольшими практическими заданиями. Четвертая же часть представляет собой полностью практическое упражнение — разработку полнофункционального и, в принципе, готового к публикации веб-сайта электронной доски объявлений.

31.1. План веб-сайта

Наша электронная доска объявлений позволит зарегистрированным пользователям публиковать объявления о продаже чего-либо. Объявления будут разноситься по рубрикам, причем структура рубрик будет иметь два уровня иерархии: на первом уровне расположатся рубрики общего плана ("недвижимость", "транспорт" и пр.), а на втором — более конкретные ("жилье", "гаражи", "дачи", "легковой", "грузовой", "специальный").

Для вывода списка объявлений мы применим пагинацию, т. к. объявлений может оказаться очень много, и страница, содержащая все объявления, будет слишком большой. Также мы предусмотрим возможность поиска объявлений по введенному посетителем слову.

Под любым объявлением (на странице сведений об объявлении) может быть оставлено произвольное количество комментариев. Оставлять комментарии будет позволено любому пользователю, в том числе и гостю.

В составе объявления пользователь может поместить основную графическую иллюстрацию, которая будет выводиться и в списке объявлений, и в составе сведений об объявлении, а также произвольное количество дополнительных иллюстраций, которые можно будет увидеть лишь на странице сведений об объявлении. И основная, и дополнительные иллюстрации не являются обязательными к размещению.

Процедура регистрации нового пользователя на сайте будет разбита на два этапа. На первом этапе посетитель вводит свои данные на странице регистрации, после чего на указанный им адрес электронной почты приходит письмо с гиперссылкой,

ведущей на страницу активации. На втором этапе посетитель переходит по гиперссылке, полученной в письме, попадает на страницу активации и становится полноправным пользователем.

Сайт доски объявлений включит в себя следующие страницы:

- главная — показывающая десять последних опубликованных объявлений без разбиения их на рубрики;
- страница списка объявлений — показывающая (с использованием пагинации) объявления из определенной рубрики. Также она будет содержать форму для поиска объявления по введенному слову;
- страница сведений о выбранном объявлении — выведет еще все оставленные для него комментарии и форму для добавления нового комментария;
- страницы регистрации и активации нового пользователя;
- страницы входа и выхода;
- страница профиля зарегистрированного пользователя — выведет список объявлений, оставленных текущим пользователем;
- страницы добавления, правки, удаления объявлений;
- страницы изменения пароля, правки и удаления пользовательского профиля;
- страницы сведений о сайте, о правах его разработчика, пользовательского соглашения и пр.

Таков, в самых общих чертах, план разрабатываемого нами сайта. Всевозможные мелочи мы уточним по ходу дела.

31.2. Подготовка проекта и приложения *main*

Сейчас мы создадим проект нашего сайта, незатейливо назвав его `bboard`, и приложение `main`, которое реализует всю функциональность сайта, за исключением веб-службы (последняя будет удостоена особого рассмотрения — в *главе 36* мы создадим под нее отдельное приложение `api`).

31.2.1. Создание и настройка проекта

Запустим командную строку, перейдем в папку, в которой будет находиться папка нового проекта, и подадим команду на создание проекта `bboard`;

```
django-admin startproject bboard
```

Когда проект будет создан, откроем модуль настроек проекта settings.py из пакета конфигурации и внесем в него следующие правки:

- изменим имя файла, в котором будет храниться база данных сайта, — на bboard.data;
- изменим код языка для вывода системных сообщений и страниц административного сайта — на `"ru"`.

Исправленные фрагменты кода модуля settings.py должны выглядеть так:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'bboard.data'),
    }
}
. . .
LANGUAGE_CODE = 'ru'
```

Пока этого достаточно. Остальные необходимые правки в настройки проекта мы внесем позднее, по ходу работы.

31.2.2. Создание и настройка приложения *main*

В командной строке перейдем в папку проекта и подадим команду на создание приложения `main`:

```
manage.py startapp main
```

В пакете приложения найдем модуль настроек приложения app.py и откроем его. Добавим в объявление конфигурационного класса `MainConfig` атрибут `verbose_name` с названием приложения:

```
class MainConfig(AppConfig):
    name = 'main'
    verbose_name = 'Доска объявлений'
```

Вернемся к модулю настроек проекта settings.py и добавим только что созданное приложение в список зарегистрированных в проекте:

```
INSTALLED_APPS = [
    . . .
    'main.apps.MainConfig',
]
```

31.3. Базовый шаблон

Создадим в пакете приложения `main` папки templates\layout и static\main. В первой мы сохраним базовый шаблон для страниц сайта, во второй — нашу таблицу стилей.

Для оформления страниц используем популярный CSS-фреймворк Bootstrap 4. Сразу же установим дополнительную библиотеку django-bootstrap4 (если не сделали этого ранее), задав в командной строке команду:

```
pip install django-bootstrap4
```

Добавим в список зарегистрированных в проекте приложение `bootstrap4` — программное ядро библиотеки django-bootstrap4:

```
INSTALLED_APPS = [
    . . .
    'bootstrap4',
]
```

В листинге 31.1 приведен код базового шаблона. Сохраним его в файле templates\layout\basic.html.

Листинг 31.1. Код базового шаблона templates\layout\basic.html

```
{% load bootstrap4 %}
{% load static %}
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <meta name="viewport"
        content="width=device-width, initial-scale=1, shrink-to-fit=no">
        <title>{% block title %}Главная{% endblock %} - Доска
        объявлений</title>
        {% bootstrap_css %}
        <link rel="stylesheet" type="text/css"
        href="{% static 'main/style.css' %}">
        {% bootstrap_javascript jquery='slim' %}
    </head>
    <body class="container-fluid">
        <header class="mb-4">
            <h1 class="display-1 text-center">Объявления</h1>
        </header>
        <div class="row">
            <ul class="col nav justify-content-end border">
                <li class="nav-item"><a class="nav-link"
                href="#">Регистрация</a></li>
                <li class="nav-item dropdown">
                    <a class="nav-link dropdown-toggle"
                    data-toggle="dropdown"
                    href="#" role="button" aria-haspopup="true"
                    aria-expanded="false">Профиль</a>
                    <div class="dropdown-menu">
                        <a class="dropdown-item" href="#">Мои
                        объявления</a>
                        <a class="dropdown-item" href="#">Изменить личные
                        данные</a>
                        <a class="dropdown-item" href="#">Изменить
                        пароль</a>
                        <div class="dropdown-divider"></div>
                        <a class="dropdown-item" href="#">Выйти</a>
```

```
                        <div class="dropdown-divider"></div>
                        <a class="dropdown-item" href="#">Удалить</a>
                    </div>
                </li>
                <li class="nav-item"><a class="nav-link"
                href="#">Вход</a></li>
            </ul>
        </div>
        <div class="row">
            <nav class="col-md-auto nav flex-column border">
                <a class="nav-link root"
                href="{% url 'main:index' %}">Главная</a>
                <span class="nav-link root font-weight-bold">
                Недвижимость</span>
                <a class="nav-link" href="#">Жилье</a>
                <a class="nav-link" href="#">Склады</a>
                <a class="nav-link" href="#">Гаражи</a>
                <span class="nav-link root font-weight-bold" href="#">
                Транспорт</span>
                <a class="nav-link" href="#">Легковой</a>
                <a class="nav-link" href="#">Грузовой</a>
            </nav>
            <section class="col border py-2">
                {% bootstrap_messages %}
                {% block content %}
                {% endblock %}
            </section>
        </div>
        <footer class="mt-3">
            <p class="text-right font-italic">&copy; читатели.</p>
        </footer>
    </body>
</html>
```

Код базового шаблона очень велик, сложен и включает много тегов и стилевых классов, формирующих разметку и оформление в стиле Bootstrap. Рассмотрим наиболее значимые фрагменты этого кода и выясним, что они делают:

- `<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">`

  Этот метатег, помещенный в секцию заголовка страницы (в парный тег `<head>`), необходим для того, чтобы Bootstrap правильно обработал страницу;

- `<title>{% block title %}Главная{% endblock %} - Доска объявлений</title>`

  В теге `<title>` мы создаем первый блок, назвав его `title`. С его помощью будет выводиться название страницы;

- `{% bootstrap_css %}`

  Привязываем к странице таблицы стилей Bootstrap;

- ```
  <link rel="stylesheet" type="text/css"
  href="{% static 'main/style.css' %}">
  ```

  Привязываем таблицу стилей static\main\style.css, которую создадим чуть позже. В ней мы запишем некоторые специфические для нашего сайта стили;

- `{% bootstrap_javascript jquery='slim' %}`

  Привязываем файлы веб-сценариев с программным кодом Bootstrap вместе с сокращенной редакцией библиотеки jQuery, без которой не заработает созданное нами раскрывающееся меню (AJAX и анимацию мы использовать не собираемся, так что полная редакция jQuery нам не нужна);

- ```
  <body class="container-fluid">
      . . .
  </body>
  ```

  К телу страницы (тегу `<body>`) привязываем стилевой класс `container-fluid`, как того требует Bootstrap;

- ```
  <header class="mb-4">
      <h1 class="display-1 text-center">Объявления</h1>
  </header>
  ```

  Стилевой класс `mb-4`, привязанный к элементу страницы, установит у него достаточно большой внешний отступ снизу. А стилевые классы `display-1` и `text-center`, привязанные к заголовку, предпишут веб-обозревателю вывести текст увеличенным шрифтом и выровнять его посередине;

- ```
  <div class="row">
      <ul class="col . . .">
          . . .
      </ul>
  </div>
  ```

  Bootstrap позволяет выполнять табличную верстку без участия собственно таблиц. Стилевой класс `row`, привязанный к элементу страницы, вынуждает его вести себя как строка таблицы, а стилевой класс `col` — как ячейка в этой строке.

  В нашем случае мы создаем строку (сформированную блоком — тегом `<div>`) с единственной ячейкой — маркированным списком. Мы сделали так для того, чтобы убрать у создаваемого элемента страницы просветы слева и справа, которые выглядят очень некрасиво;

- ```
  <ul class=". . . nav justify-content-end border">
      <li class="nav-item"><a class="nav-link"
      href="#">Регистрация</a></li>
      . . .
  </ul>
  ```
  ```

Упомянутый ранее маркированный список мы используем для создания горизонтальной полосы навигации. Для этого привяжем к нему стилевой класс `nav`. Стилевой класс `justify-content-end` укажет выровнять пункты полосы навигации по правому краю, а стилевой класс `border` создаст рамку вокруг нее.

Пункты полосы навигации создаются так же, как и пункты списков, — тегами `<li>` — с привязанным стилевым классом `nav-item`. Внутри этих тегов должны находиться гиперссылки с привязанными стилевыми классами `nav-link`;

```
<li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle" data-toggle="dropdown"
    href="#" role="button" aria-haspopup="true"
    aria-expanded="false">Профиль</a>
    <div class="dropdown-menu">
        <a class="dropdown-item" href="#">Мои объявления</a>
        . . .
        <div class="dropdown-divider"></div>
        <a class="dropdown-item" href="#">Удалить</a>
    </div>
</li>
```

А данный код создает в полосе навигации пункт с раскрывающимся меню. Для этого внутрь тега `<li>` помещается, помимо гиперссылки, еще и блок, который создаст само меню. Обратим внимание на стилевые классы, привязываемые к различным элементам, и дополнительные атрибуты, которые обязательно должны присутствовать в тегах.

Пункты раскрывающегося меню формируются с помощью обычных гиперссылок со стилевым классом `dropdown-item`. А "пустой" блок со стилевым классом `dropdown-divider` создаст разделитель между пунктами;

```
<div class="row">
    <nav class="col-md-auto . . .">
        . . .
    </nav>
    <section class="col . . .">
        . . .
    </section>
</div>
```

Здесь мы снова применили табличную верстку, но на этот раз из двух "ячеек": семантической панели навигации (тега `<nav>`) и семантической секции страницы (тега `<section>`).

Знакомый нам стилевой класс `col` при привязке к элементам-"ячейкам" создаст "ячейки" одинаковой ширины. Если нужно сделать так, чтобы ширина какой-либо ячейки соответствовала ширине ее содержимого, то нужно привязать к создающему эту ячейку элементу страницы стилевой класс `col-md-auto`. Мы привязали его к панели навигации;

```
<nav class=". . . nav flex-column border">
    <a class="nav-link root" href="{% url 'main:index' %}">Главная</a>
    <span class="nav-link root font-weight-bold">Недвижимость</span>
    <a class="nav-link" href="#">Жилье</a>
    . . .
</nav>
```

Мы привязали к семантической панели навигации стилевые классы `nav` и `flex-column` (чтобы превратить ее в вертикальную панель навигации в стиле Bootstrap), а также создающий рамку стилевой класс `border`.

Разные пункты панели навигации мы формируем тремя разными способами:

- пункты, ведущие на служебные страницы, — гиперссылками с двумя привязанными стилевыми классами: `nav-link`, который нам уже знаком, и `root`, который мы запишем в таблице стилей static\main\style.css (он задаст увеличенный размер шрифта);
- пункты, обозначающие рубрики верхнего уровня (надрубрики), — тегами `<span>` со стилевыми классами `nav-link`, `root` и `font-weight-bold`. Последний стилевой класс задаст полужирное начертание шрифта — так мы визуально выделим пункты этого типа. Обратим внимание, что такие пункты не являются гиперссылками;
- пункты, ведущие на рубрики нижнего уровня (подрубрики), — гиперссылками со стилевым классом `nav-link`;

```
<section class=". . . border py-2">
    {% bootstrap_messages %}
    {% block content %}
    {% endblock %}
</section>
```

К семантической секции мы также привязали стилевые классы `border` и `py-2`. Первый нам уже знаком, а второй установит для тега небольшие внутренние отступы сверху и внизу, чтобы содержимое тега не примыкало к рамке вплотную.

В семантическую секцию мы поместили код, выводящий всплывающие сообщения, и блок `content`, в котором будет выводиться основное содержимое страниц;

```
<footer class="mt-3">
    <p class="text-right font-italic">&copy; читатели.</p>
</footer>
```

"Поддон" сайта мы создаем специализированным тегом `<footer>`. Стилевой класс `mt-3` укажет для него внешний отступ сверху средних размеров, чтобы отделить его от вышерасположенных элементов. Чтобы выровнять текст абзаца по правому краю и вывести его курсивом, мы привязали к тегу `<p>` стилевые классы `text-right` и `font-italic`.

В листинге 31.2 приведен код таблицы стилей static\main\style.css. Создадим ее.

Листинг 31.2. Таблица стилей static\main\style.css

```
header h1 {
    background: url("bg.jpg") left / auto 100% no-repeat, url("bg.jpg") ↪
    right / auto 100% no-repeat;
}
.root {
    font-size: larger;
}
```

Первый стиль создаст у заголовка сайта фон в виде двух изображений доски объявлений, выведенных слева и справа. Подходящее изображение найдем в Интернете и также сохраним в папке static\main пакета приложения под именем bg.jpg.

31.4. Главная веб-страница

Главную страницу сделаем совсем минималистичной — только чтобы удостовериться, работает ли сайт.

В листинге 31.3 приведен код контроллера `index()`, выводящего главную страницу, который запишем в модуль views.py пакета приложения `main`. Этот контроллер мы реализовали в виде функции — так проще.

Листинг 31.3. Код контроллера-функции `index()`

```
from django.shortcuts import render

def index(request):
    return render(request, 'main/index.html')
```

В пакете приложения main создадим папку templates\main, в которой будем сохранять шаблоны страниц.

Создадим шаблон главной страницы templates\main\index.html, записав в него код из листинга 31.4.

Листинг 31.4. Код шаблона templates\main\index.html

```
{% extends "layout/basic.html" %}

{% block content %}
<h2>Последние 10 объявлений</h2>
{% endblock %}
```

Теперь нужно написать маршруты. Заодно мы сделаем так, чтобы статические файлы сайта не кэшировались веб-обозревателем — ведь работа над таблицей стилей еще не закончена.

Начнем со списка маршрутов уровня проекта, хранящегося в модуле urls.py пакета конфигурации. Откроем и исправим его код согласно листингу 31.5.

Листинг 31.5. Код модуля urls.py пакета конфигурации

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.contrib.staticfiles.views import serve
from django.views.decorators.cache import never_cache

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('main.urls')),
]

if settings.DEBUG:
    urlpatterns.append(path('static/<path:path>', never_cache(serve)))
```

Приложение `main` мы установили в качестве корневого.

Приступим к созданию списка маршрутов уровня приложения. Создадим в пакете приложения модуль urls.py и запишем в него код из листинга 31.6.

Листинг 31.6. Код модуля urls.py пакета приложения

```
from django.urls import path

from .views import index

app_name = 'main'
urlpatterns = [
    path('', index, name='index'),
]
```

Сохраним все вновь созданные и исправленные файлы и запустим отладочный веб-сервер Django с отключенной обработкой статических файлов:

```
manage.py runserver --nostatic
```

Откроем веб-обозреватель, выполним переход по интернет-адресу **http://localhost:8000/** и попадем на главную страницу нашего сайта (рис. 31.1).

Здесь мы видим заголовок сайта, украшенный изображениями стилизованной доски объявлений, горизонтальную полосу навигации с тремя пунктами, причем третий имеет раскрывающееся меню (на рис. 31.1 оно как раз открыто), вертикальную панель навигации слева и "поддон".

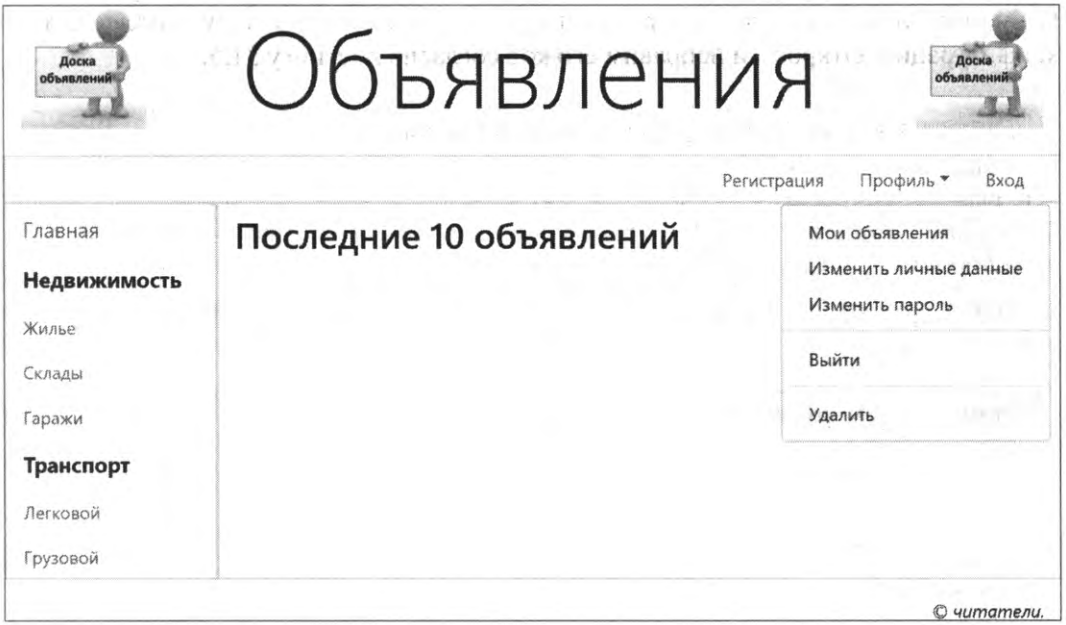


Рис. 31.1. Главная веб-страница сайта доски объявлений

31.5. Вспомогательные веб-страницы

Сразу же создадим первую из вспомогательных веб-страниц — со сведениями о сайте и правах его разработчиков. Сделать это можно двумя способами.

Первый способ, самый очевидный, заключается в том, что для каждой страницы пишется отдельный контроллер и отдельный маршрут. Этот способ весьма трудоемок, поскольку придется писать несколько контроллеров с практически одинаковым кодом, и подходит лишь для тех случаев, когда страницы должны выводить какие-либо данные, извлекаемые из базы или формируемые программно.

Второй способ — вывод всех страниц с применением одного контроллера и, соответственно, одного маршрута. Какой-либо идентификатор страницы, предназначенной к выводу на экран, передается контроллеру с URL-параметром. Трудоемкость работы в таком случае существенно снижается, поскольку нужно написать всего один контроллер.

Реализуем вывод вспомогательных страниц вторым способом. В качестве идентификатора страницы используем имя формирующего ее шаблона без пути и без расширения — так проще.

В список маршрутов уровня приложения, что хранится в модуле urls.py пакета приложения, добавим такой код:

```
from .views import other_page
. . .
```

```
urlpatterns = [
    path('<str:page>/', other_page, name='other'),
    path('', index, name='index'),
]
```

Имя шаблона выводимой страницы передаем через URL-параметр `page`. А контроллер, выводящий вспомогательные страницы, назовем `other_page` и реализуем в виде функции. Вообще, контроллеры-функции — идеальный инструмент для написания чего-либо нестандартного, специфического.

В модуль views.py пакета приложения, где хранится код контроллеров, добавим код контроллера-функции `other_page()`, приведенный в листинге 31.7.

Листинг 31.7. Код контроллера-функции other_page()

```
from django.http import HttpResponse, Http404
from django.template import TemplateDoesNotExist
from django.template.loader import get_template

def other_page(request, page):
    try:
        template = get_template('main/' + page + '.html')
    except TemplateDoesNotExist:
        raise Http404
    return HttpResponse(template.render(request=request))
```

Имя выводимой страницы получаем из параметра `page`, добавляем к нему путь и расширение, получив тем самым полный путь к нужному шаблону, и пытаемся загрузить его вызовом функции `get_template()`. Если загрузка прошла успешно, то формируем на основе этого шаблона страницу.

Если же шаблон загрузить не удалось, то функция `get_template()` возбудит исключение `TemplateDoesNotExist`. Мы перехватываем это исключение и возбуждаем другое исключение — `Http404`, которое приведет к отправке страницы с сообщением об ошибке 404 (запрошенная страница не существует).

Странице со сведениями о сайте и правах его разработчиков дадим имя `about`. Код формирующего ее шаблона templates\main\about.html приведен в листинге 31.8.

Листинг 31.8. Код шаблона templates\main\about.html

```
{% extends "layout/basic.html" %}

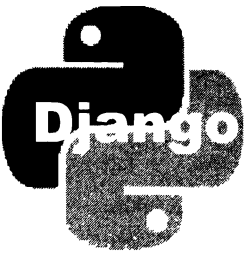
{% block title %}О сайте{% endblock %}

{% block content %}
<h2>О сайте</h2>
<p>Сайт для публикации объявлений о продаже, разбитых на рубрики.</p>
<p>Все права принадлежат читателям книги &laquo;Django 3.0&raquo;.</p>
{% endblock %}
```

Откроем базовый шаблон templates\layout\basic.html и добавим в левую панель навигации такой код:

```
<nav class="col-md-auto nav flex-column border">
    . . .
    <a class="nav-link root" href="{% url 'main:other' page='about' %}">
    О сайте</a>
</nav>
```

Сохраним все исправленные и вновь созданные файлы, обновим открытую в веб-обозревателе главную страницу, перейдем по гиперссылке **О сайте** — и увидим только что созданную страницу.



ГЛАВА 32

Работа с пользователями и разграничение доступа

Теперь займемся инструментами для работы с пользователями и разграничения доступа. Мы создадим страницы для входа и выхода, регистрации, активации, страницы пользовательского профиля, для смены данных о пользователе, смены его пароля и удаления профиля.

32.1. Модель пользователя

Стандартная модель пользователя `User`, предлагаемая стандартным же приложением `django.contrib.auth`, не подходит, поскольку нам нужно хранить дополнительные данные о пользователе. Поэтому создадим свою собственную модель, сделав ее производной от стандартной абстрактной модели `AbstractUser`, объявленной в модуле `django.contrib.auth`.

Наша модель будет носить название `AdvUser`. Список полей, которые мы объявим в ней, приведен в табл. 32.1.

***Таблица 32.1.** Структура модели `AdvUser`*

Имя	Тип	Дополнительные параметры	Описание
`is_activated`	`BooleanField`	Значение по умолчанию — `True`, индексированное	Признак, прошел ли пользователь процедуру активации
`send_messages`	`BooleanField`	Значение по умолчанию — `True`	Признак, желает ли пользователь получать уведомления о новых комментариях

Код, объявляющий эту модель, приведен в листинге 32.1. Запишем его в модуль models.py пакета приложения.

Листинг 32.1. Код модели AdvUser

```
from django.db import models
from django.contrib.auth.models import AbstractUser

class AdvUser(AbstractUser):
    is_activated = models.BooleanField(default=True, db_index=True,
                                       verbose_name='Прошел активацию?')
    send_messages = models.BooleanField(default=True,
                    verbose_name='Слать оповещения о новых комментариях?')

    class Meta(AbstractUser.Meta):
        pass
```

Сразу же укажем ее как модель пользователя, используемую подсистемой разграничения доступа Django. Для этого откроем модуль settings.py пакета конфигурации и добавим в него строку:

```
AUTH_USER_MODEL = 'main.AdvUser'
```

Остановим отладочный веб-сервер Django и зададим в командной строке команду сначала на создание миграций:

```
manage.py makemigrations
```

а потом — на их выполнение:

```
manage.py migrate
```

Как только миграции будут выполнены, создадим суперпользователя, подав команду:

```
manage.py createsuperuser
```

Введем выбранные нами имя, адрес электронной почты и пароль создаваемого пользователя.

Напоследок откроем модуль admin.py пакета приложения, в котором объявляются классы-редакторы и регистрируются модели в административном сайте. Зарегистрируем нашу модель пользователя, добавив в этот модуль код:

```
from .models import AdvUser

admin.site.register(AdvUser)
```

Запустим отладочный веб-сервер, опять же отключив обработку статических файлов. Откроем административный веб-сайт, набрав интернет-адрес **http://localhost:8000/admin/**, и попробуем выполнить вход от имени только что созданного суперпользователя.

32.2. Основные веб-страницы: входа, профиля и выхода

Далее ради простоты мы будем, по возможности, следовать установленным Django соглашениям (вообще, на взгляд автора, это лучшая политика в Django-программировании, да и в целом в разработке сайтов).

32.2.1. Веб-страница входа

Для реализации входа мы создадим подкласс контроллера-класса `LoginView`, в котором запишем все необходимые для работы контроллера параметры.

Код контроллера-класса, выполняющего вход и носящего имя `BBLoginView`, приведен в листинге 32.2. Добавим его в модуль views.py пакета приложения.

Листинг 32.2. Код контроллера-класса BBLoginView

```
from django.contrib.auth.views import LoginView

class BBLoginView(LoginView):
    template_name = 'main/login.html'
```

В классе мы указали лишь путь к файлу шаблона, занеся его в атрибут `template_name`. Остальные параметры сохранят значения по умолчанию, т. к. мы собираемся следовать принятым в Django соглашениям.

Шаблон страницы входа login.html мы поместили в папку templates\main — туда же, где находятся все остальные шаблоны. Поскольку наш сайт включает относительно немного страниц, будем хранить их в одной папке, чтобы упростить сопровождение сайта.

Запишем новый маршрут, указывающий на контроллер `BBLoginView`, в списке уровня приложения. Откроем модуль urls.py пакета приложения и добавим в него код:

```
from .views import BBLoginView
. . .
urlpatterns = [
    path('accounts/login/', BBLoginView.as_view(), name='login'),
    . . .
]
```

В маршруте мы указали шаблонный путь **accounts/login/**. По умолчанию именно по нему Django выполняет перенаправление при попытке гостя получить доступ к закрытой от него странице.

> ***На заметку***
>
> Мы могли бы записать маршрут и таким образом:
>
> ```
> from django.contrib.auth.views import LoginView
> . . .
> ```

```
urlpatterns = [
    path('accounts/login/',
         LoginView.as_view(template_name='main/login.html'), name='login'),
    . . .
]
```

указав в маршруте непосредственно класс `LoginView` и записав все необходимые параметры контроллера в вызове его метода `as_view()`. Но в таком случае код контроллеров окажется записан в двух модулях: *views.py* и *urls.py*, и в дальнейшем, при сопровождении сайта, в поисках нужного фрагмента кода нам придется просматривать оба этих модуля, что неудобно.

Поэтому давайте держать код контроллеров в модуле *views.py*, а код списка маршрутов — в модуле *urls.py*. Так нам будет проще.

Напишем шаблон страницы входа templates\main\login.html. Его код приведен в листинге 32.3.

Листинг 32.3. Код шаблона templates\main\login.html

```
{% extends "layout/basic.html" %}

{% load bootstrap4 %}

{% block title %}Вход{% endblock %}

{% block content %}
<h2>Вход</h2>
{% if user.is_authenticated %}
<p>Вы уже выполнили вход.</p>
{% else %}
<form method="post">
    {% csrf_token %}
    {% bootstrap_form form layout='horizontal' %}
    <input type="hidden" name="next" value="{{ next }}">
    {% buttons submit='Войти' %}{% endbuttons %}
</form>
{% endif %}
{% endblock %}
```

Поля ввода имени и пароля невелики, так что мы вывели форму в "горизонтальной" разметке Bootstrap, когда надпись и относящийся к ней элемент управления располагаются по горизонтали.

Напоследок внесем правки в базовый шаблон templates\layout\basic.html. Найдем в нем фрагмент кода, создающего пункт **Вход** и пункт с раскрывающимся меню **Профиль** горизонтальной полосы навигации, и исправим его следующим образом:

```
<ul class="col nav justify-content-end border">
    <li class="nav-item"><a .. . .>Регистрация</a></li>
    {% if user.is_authenticated %}
```

```
    <li class="nav-item dropdown">
        . . .
    </li>
    {% else %}
    <li class="nav-item"><a . . .>Вход</a></li>
    {% endif %}
</ul>
```

В результате пункт **Вход** будет выводиться только гостям, а пункт **Профиль** — только пользователям, выполнившим вход.

Запишем в тег `<a>`, создающий гиперссылку **Вход**, интернет-адрес страницы входа:

```
<a . . . href="{% url 'main:login' %}">Вход</a>
```

Сохраним все новые и исправленные файлы, обновим открытую в веб-обозревателе главную страницу и щелкнем на гиперссылке **Вход**. Если мы все сделали без ошибок, то сразу же попадем на страницу входа.

Но пока не будем выполнять вход, иначе возникнет ошибка. Сначала сделаем страницы профиля и выхода.

32.2.2. Веб-страница пользовательского профиля

Контроллер, выводящий страницу пользовательского профиля, реализуем в виде функции и назовем `profile()`. Его код чрезвычайно прост — см. листинг 32.4. Не забываем, что все контроллеры объявляются в модуле views.py пакета приложения.

Листинг 32.4. Код контроллера-функции `profile()`

```
from django.contrib.auth.decorators import login_required

@login_required
def profile(request):
    return render(request, 'main/profile.html')
```

Поскольку страница пользовательского профиля должна быть доступна только зарегистрированным пользователям, выполнившим вход на сайт, мы пометили контроллер-функцию `profile()` декоратором `login_required()`.

Добавим в список маршрутов уровня приложения (не забываем, что он хранится в модуле urls.py пакета приложения) маршрут на контроллер `profile()`:

```
from .views import profile
. . .
urlpatterns = [
    path('accounts/profile/', profile, name='profile'),
    . . .
]
```

Мы указали в этом маршруте шаблонный путь **accounts/profile/** — по нему Django по умолчанию выполняет перенаправление после успешного входа.

Код шаблона templates\main\profile.html, формирующего страницу профиля, приведен в листинге 32.5.

Листинг 32.5. Код шаблона templates\main\profile.html

```
{% extends "layout/basic.html" %}

{% block title %}Профиль пользователя{% endblock %}

{% block content %}
<h2>Профиль пользователя {{ user.username }}</h2>
{% if user.first_name and user.last_name %}
<p>Здравствуйте, {{ user.first_name }} {{ user.last_name }}!</p>
{% else %}
<p>Здравствуйте!</p>
{% endif %}
<h3>Ваши объявления</h3>
{% endblock %}
```

Если пользователь при регистрации написал свои имя и фамилию, то на странице будет выведено персонализированное приветствие. Если же пользователь не ввел эти данные, появится простое приветствие.

В дальнейшем на странице пользовательского профиля будет выводиться список объявлений, оставленных текущим пользователем. Но объявлений у нас пока что нет (более того, сама функциональность по их написанию еще не создавалась), так что более на этой странице ничего не выводится.

В шаблоне templates/layout/basic.html найдем тег <a>, выводящий гиперссылку **Мои объявления**, и вставим в него интернет-адрес страницы профиля:

```
<a . . . href="{% url 'main:profile' %}">Мои объявления</a>
```

32.2.3. Веб-страница выхода

Контроллер выхода реализуем в виде класса BBLogoutView, производного от класса LogoutView. Его код приведен в листинге 32.6.

Листинг 32.6. Код контроллера-класса BBLogoutView

```
from django.contrib.auth.views import LogoutView
from django.contrib.auth.mixins import LoginRequiredMixin

class BBLogoutView(LoginRequiredMixin, LogoutView):
    template_name = 'main/logout.html'
```

Страница выхода должна быть доступна только зарегистрированным пользователям, выполнившим вход. Поэтому мы добавили в число суперклассов контроллера-класса `BBLogoutView` примесь `LoginRequiredMixin`.

В списке маршрутов уровня приложения запишем маршрут, ведущий на этот контроллер:

```
from .views import BBLogoutView
. . .
urlpatterns = [
    path('accounts/logout/', BBLogoutView.as_view(), name='logout'),
    . . .
]
```

Напишем шаблон страницы выхода templates\main\logout.html, очень простой код которого приведен в листинге 32.7.

Листинг 32.7. Код шаблона templates\main\logout.html

```
{% extends "layout/basic.html" %}

{% block title %}Выход{% endblock %}

{% block content %}
<h2>Выход</h2>
<p>Вы успешно вышли с сайта.</p>
{% endblock %}
```

В шаблоне templates/layout/basic.html найдем тег `<a>`, выводящий гиперссылку **Выход**, и поместим в него интернет-адрес страницы выхода:

```
<a . . . href="{% url 'main:logout' %}">Выйти</a>
```

Сохраним все файлы, подождем, пока отладочный веб-сервер не перезапустится, и обновим страницу входа, открытую в веб-обозревателе. Занесем в форму имя и пароль созданного ранее суперпользователя и выполним вход. Посмотрим на страницу профиля и выйдем с сайта.

32.3. Веб-страницы правки личных данных пользователя

32.3.1. Веб-страница правки основных сведений

На этой странице пользователь сможет исправить свои имя (логин), адрес электронной почты, реальные имя, фамилию и признак, хочет ли он получать по электронной почте оповещения о появлении новых комментариев к его объявлениям. Адрес электронной почты будет обязательным к заполнению.

Сначала объявим форму ChangeUserInfoForm, связанную с моделью AdvUser и предназначенную для ввода основных данных. Код формы приведен в листинге 32.8. Его мы запишем во вновь созданный модуль forms.py пакета приложения.

Листинг 32.8. Код формы ChangeUserInfoForm

```
from django import forms

from .models import AdvUser

class ChangeUserInfoForm(forms.ModelForm):
    email = forms.EmailField(required=True,
                             label='Адрес электронной почты')

    class Meta:
        model = AdvUser
        fields = ('username', 'email', 'first_name', 'last_name',
                  'send_messages')
```

Так как мы хотим сделать поле email модели AdvUser обязательным для заполнения, то выполним полное объявление поля email формы. А поскольку параметры остальных полей формы: username, first_name, last_name и send_messages — у нас не меняются, в их отношении мы применим быстрое объявление.

Контроллер страницы основных данных должен выполнять правку записи модели, так что мы можем написать его на базе высокоуровневого класса UpdateView. Готовый код контроллера-класса ChangeUserInfoView приведен в листинге 32.9.

Листинг 32.9. Код контроллера-класса ChangeUserInfoView

```
from django.views.generic.edit import UpdateView
from django.contrib.messages.views import SuccessMessageMixin
from django.urls import reverse_lazy
from django.shortcuts import get_object_or_404

from .models import AdvUser
from .forms import ChangeUserInfoForm

class ChangeUserInfoView(SuccessMessageMixin, LoginRequiredMixin, UpdateView):
    model = AdvUser
    template_name = 'main/change_user_info.html'
    form_class = ChangeUserInfoForm
    success_url = reverse_lazy('main:profile')
    success_message = 'Данные пользователя изменены'

    def setup(self, request, *args, **kwargs):
        self.user_id = request.user.pk
        return super().setup(request, *args, **kwargs)
```

```
    def get_object(self, queryset=None):
        if not queryset:
            queryset = self.get_queryset()
        return get_object_or_404(queryset, pk=self.user_id)
```

В процессе работы этот контроллер должен извлечь из модели `AdvUser` запись, представляющую текущего пользователя, для чего ему нужно предварительно получить ключ текущего пользователя. Получить его можно из объекта текущего пользователя, хранящегося в атрибуте `user` объекта запроса.

Вероятно, наилучшее место для получения ключа текущего пользователя — метод `setup()`, наследуемый всеми контроллерами-классами от их общего суперкласса `View`. Этот метод выполняется в самом начале исполнения контроллера-класса и получает объект запроса в качестве одного из параметров. В переопределенном методе `setup()` мы извлечем ключ пользователя и сохраним его в атрибуте `user_id`.

Извлечение исправляемой записи выполняем в методе `get_object()`, который контроллер-класс унаследовал от примеси `SingleObjectMixin`. В переопределенном методе сначала учитываем тот момент, что набор записей, из которого следует извлечь искомую запись, может быть передан методу с параметром `queryset`, а может быть и не передан — в этом случае набор записей следует получить вызовом метода `get_queryset()`. После этого непосредственно ищем запись, представляющую текущего пользователя.

В качестве одного из суперклассов этого контроллера-класса мы указали примесь `LoginRequiredMixin`, запрещающую доступ к контроллеру гостям, и примесь `SuccessMessageMixin`, которая применяется для вывода всплывающих сообщений об успешном выполнении операции. Зря мы, что ли, вставили в шаблон templates\layout\basic.html код, выводящий всплывающие сообщения...

В список маршрутов уровня приложения добавим соответствующий маршрут:

```
from .views import ChangeUserInfoView
. . .
urlpatterns = [
    . . .
    path('accounts/profile/change/', ChangeUserInfoView.as_view(),
                                     name='profile_change'),
    path('accounts/profile/', profile, name='profile'),
    . . .
]
```

Код шаблона templates\main\change_user_info.html, создающего страницу для правки основных данных, приведен в листинге 32.10. Ничего особо сложного в нем нет.

Листинг 32.10. Код шаблона templates\main\change_user_info.html

```
{% extends "layout/basic.html" %}

{% load bootstrap4 %}
```

```
{% block title %}Правка личных данных{% endblock %}

{% block content %}
<h2>Правка личных данных пользователя {{ user.username }}</h2>
<form method="post">
    {% csrf_token %}
    {% bootstrap_form form layout='horizontal' %}
    {% buttons submit='Сохранить' %}{% endbuttons %}
</form>
{% endblock %}
```

В шаблоне templates\layout\basic.html отыщем тег <a>, выводящий гиперссылку **Изменить личные данные**, и вставим в него интернет-адрес страницы правки основных данных:

```
<a . . . href="{% url 'main:profile_change' %}">Изменить личные данные</a>
```

Сохраним файлы, перейдем на страницу пользовательского профиля и проверим только что созданную страницу правки основных данных в работе.

32.3.2. Веб-страница правки пароля

Контроллер-класс BBPasswordChangeView, выводящий эту страницу, сделаем производным от класса PasswordChangeView, который реализует смену пароля. Код нашего контроллера-класса приведен в листинге 32.11.

Листинг 32.11. Код контроллера-класса BBPasswordChangeView

```
from django.contrib.auth.views import PasswordChangeView

class BBPasswordChangeView(SuccessMessageMixin, LoginRequiredMixin,
                           PasswordChangeView):
    template_name = 'main/password_change.html'
    success_url = reverse_lazy('main:profile')
    success_message = 'Пароль пользователя изменен'
```

После успешной смены пароля выполняем перенаправление на страницу профиля пользователя с выводом соответствующего всплывающего сообщения.

Добавим в список маршрутов уровня приложения маршрут, который укажет на новый контроллер:

```
from .views import BBPasswordChangeView
. . .
urlpatterns = [
    path('accounts/logout/', BBLogoutView.as_view(), name='logout'),
    path('accounts/password/change/', BBPasswordChangeView.as_view(),
                                      name='password_change'),
    . . .
]
```

В листинге 32.12 приведен код шаблона страницы для смены пароля templates\main\password_change.html.

Листинг 32.12. Код шаблона templates\main\password_change.html

```
{% extends "layout/basic.html" %}

{% load bootstrap4 %}

{% block title %}Смена пароля{% endblock %}

{% block content %}
<h2>Смена пароля пользователя {{ user.username }}</h2>
<form method="post">
    {% csrf_token %}
    {% bootstrap_form form layout='horizontal' %}
    {% buttons submit='Сменить пароль' %}{% endbuttons %}
</form>
{% endblock %}
```

Осталось в коде шаблона templates\layout\basic.html найти код, создающий гиперссылку **Изменить пароль**, и поместить в нее правильный интернет-адрес:

```
<a . . . href="{% url 'main:password_change' %}">Изменить пароль</a>
```

Теперь можно сохранить все файлы и проверить, работает ли страница смены пароля.

32.4. Веб-страницы регистрации и активации пользователей

32.4.1. Веб-страницы регистрации нового пользователя

Мы напишем форму для ввода сведений о новом пользователе, контроллеры и шаблоны для страниц непосредственно регистрации и уведомления об успешной регистрации.

Для отправки письма о необходимости активации мы объявим свой сигнал. Называться он будет `user_registered` и получит в качестве единственного параметра `instance` объект вновь созданного пользователя.

Сигнал мы объявим в модуле apps.py пакета приложения. Этот модуль выполняется непосредственно при инициализации приложения и, таким образом, является идеальным местом для записи кода, объявляющего сигналы.

32.4.1.1. Форма для занесения сведений о новом пользователе

Код класса формы `RegisterUserForm`, приведенный в листинге 32.13, запишем в модуль forms.py пакета приложения.

Листинг 32.13. Код формы RegisterUserForm

```
from django.contrib.auth import password_validation
from django.core.exceptions import ValidationError

from .apps import user_registered

class RegisterUserForm(forms.ModelForm):
    email = forms.EmailField(required=True,
                             label='Адрес электронной почты')
    password1 = forms.CharField(label='Пароль',
      widget=forms.PasswordInput,
      help_text=password_validation.password_validators_help_text_html())
    password2 = forms.CharField(label='Пароль (повторно)',
      widget=forms.PasswordInput,
      help_text='Введите тот же самый пароль еще раз для проверки')

    def clean_password1(self):
        password1 = self.cleaned_data['password1']
        if password1:
            password_validation.validate_password(password1)
        return password1

    def clean(self):
        super().clean()
        password1 = self.cleaned_data['password1']
        password2 = self.cleaned_data['password2']
        if password1 and password2 and password1 != password2:
            errors = {'password2': ValidationError(
              'Введенные пароли не совпадают', code='password_mismatch')}
            raise ValidationError(errors)

    def save(self, commit=True):
        user = super().save(commit=False)
        user.set_password(self.cleaned_data['password1'])
        user.is_active = False
        user.is_activated = False
        if commit:
            user.save()
        user_registered.send(RegisterUserForm, instance=user)
        return user

    class Meta:
        model = AdvUser
        fields = ('username', 'email', 'password1', 'password2',
                  'first_name', 'last_name', 'send_messages')
```

Здесь мы также комбинируем быстрое и полное объявление полей. Полное объявление используем для создания полей электронной почты (поскольку хотим сделать его обязательным для заполнения) и обоих полей для занесения пароля. Согласно общепринятой практике, отведем для занесения пароля два поля, в которые нужно ввести один и тот же пароль.

В качестве дополнительного поясняющего текста у первого поля пароля указываем объединенный текст с требованиями к вводимому паролю, предоставленный всеми доступными в системе валидаторами, — там новый пользователь сразу поймет, какие требования предъявляются к паролю.

В методе `clean_password1()` выполняем валидацию пароля, введенного в первое поле, с применением доступных в системе валидаторов пароля. Проверять таким же образом пароль из второго поля нет нужды — если пароль из первого поля некорректен, не имеет значения, является ли корректным пароль из второго поля.

В переопределенном методе `clean()` проверяем, совпадают ли оба введенных пароля. Эта проверка будет проведена после валидации пароля из первого поля.

В переопределенном методе `save()` при сохранении нового пользователя заносим значения `False` в поля `is_active` (признак, является ли пользователь активным) и `is_activated` (признак, выполнил ли пользователь процедуру активации), тем самым сообщая фреймворку, что этот пользователь еще не может выполнять вход на сайт. Далее сохраняем в записи закодированный пароль и отправляем сигнал `user_registered`, чтобы отослать пользователю письмо с требованием активации.

32.4.1.2. Средства для регистрации пользователя

Эти средства будут включать две страницы: одна выведет веб-форму для ввода данных о регистрирующемся пользователе, вторая сообщит об успешной регистрации и отправке письма с требованием активации.

Контроллер-класс, регистрирующий пользователя, мы назовем `RegisterUserView` и сделаем производным от класса `CreateView`. Его код приведен в листинге 32.14.

Листинг 32.14. Код контроллера-класса `RegisterUserView`

```
from django.views.generic.edit import CreateView

from .forms import RegisterUserForm

class RegisterUserView(CreateView):
    model = AdvUser
    template_name = 'main/register_user.html'
    form_class = RegisterUserForm
    success_url = reverse_lazy('main:register_done')
```

Контроллер, который выведет сообщение об успешной регистрации, будет называться `RegisterDoneView` и, в силу его исключительной простоты, станет производным от класса `TemplateView`. Его код можно увидеть в листинге 32.15.

Листинг 32.15. Код контроллера-класса RegisterDoneView

```
from django.views.generic.base import TemplateView

class RegisterDoneView(TemplateView):
    template_name = 'main/register_done.html'
```

В список маршрутов уровня приложения добавим два маршрута, ведущие на только что написанные нами контроллеры:

```
from .views import RegisterUserView, RegisterDoneView
. . .
urlpatterns = [
    path('accounts/register/done/', RegisterDoneView.as_view(),
                                    name='register_done'),
    path('accounts/register/', RegisterUserView.as_view(),
                               name='register'),
    path('accounts/login/', BBLoginView.as_view(), name='login'),
    . . .
]
```

Код шаблонов templates\main\register_user.html и templates\main\register_done.html, которые формируют страницы регистрации и уведомления о ее успешном завершении, приведен в листингах 32.16 и 32.17 соответственно.

Листинг 32.16. Код шаблона templates\main\register_user.html

```
{% extends "layout/basic.html" %}

{% load bootstrap4 %}

{% block title %}Регистрация{% endblock %}

{% block content %}
<h2>Регистрация нового пользователя</h2>
<form method="post">
    {% csrf_token %}
    {% bootstrap_form form layout='horizontal' %}
    {% buttons submit='Зарегистрироваться' %}{% endbuttons %}
</form>
{% endblock %}
```

Листинг 32.17. Код шаблона templates\main\register_done.html

```
{% extends "layout/basic.html" %}

{% block title %}Регистрация завершена{% endblock %}
```

```
{% block content %}
<h2>Регистрация</h2>
<p>Регистрация пользователя завершена.</p>
<p>На адрес электронной почты, указанный пользователем, выслано письмо
для активации.</p>
{% endblock %}
```

В шаблоне templates\layout\basic.html найдем фрагмент, создающий гиперссылку **Регистрация**, и вставим в нее интернет-адрес страницы регистрации:

```
<a . . . href="{% url 'main:register' %}">Регистрация</a>
```

32.4.1.3. Средства для отправки писем с требованиями активации

Непосредственную рассылку электронных писем будет выполнять функция `send_activation_notification()`, которую мы объявим чуть позже, во вновь созданном модуле utilities.py. Эта функция еще пригодится нам, когда мы будем писать редактор для модели `AdvUser`.

Откроем модуль apps.py пакета приложения и запишем в него код, который объявит сигнал `user_registered` и привяжет к нему обработчик:

```
from django.dispatch import Signal

from .utilities import send_activation_notification

user_registered = Signal(providing_args=['instance'])

def user_registered_dispatcher(sender, **kwargs):
    send_activation_notification(kwargs['instance'])

user_registered.connect(user_registered_dispatcher)
```

Создадим в пакете приложения модуль utilities.py. Занесем в него объявление функции `send_activation_notification()` из листинга 32.18.

Листинг 32.18. Код, реализующий отправку писем с оповещениями об активации

```
from django.template.loader import render_to_string
from django.core.signing import Signer

from bboard.settings import ALLOWED_HOSTS

signer = Signer()

def send_activation_notification(user):
    if ALLOWED_HOSTS:
        host = 'http://' + ALLOWED_HOSTS[0]
```

```
    else:
        host = 'http://localhost:8000'
    context = {'user': user, 'host': host,
                             'sign': signer.sign(user.username)}
    subject = render_to_string('email/activation_letter_subject.txt',
                               context)
    body_text = render_to_string('email/activation_letter_body.txt',
                                 context)
    user.email_user(subject, body_text)
```

Чтобы сформировать интернет-адрес, ведущий на страницу подтверждения активации, понадобится, во-первых, домен, на котором находится наш сайт, а во-вторых, некоторое значение, уникально идентифицирующее только что зарегистрированного пользователя и при этом устойчивое к попыткам его подделать.

Домен мы можем извлечь из списка разрешенных доменов, который записан в параметре `ALLOWED_HOSTS` настроек проекта. Выберем самый первый домен, присутствующий в списке. Если же список доменов пуст, мы задействуем интернет-адрес, используемый отладочным веб-сервером Django.

В качестве уникального и стойкого к подделке идентификатора пользователя применяем его имя, защищенное цифровой подписью. Создание цифровой подписи выполняем посредством класса `Signer`.

Текст темы и тела письма формируем с применением шаблонов templates\email\activation_letter_subject.txt и templates\email\activation_letter_body.txt соответственно. Их код приведен в листингах 32.19 и 32.20.

Листинг 32.19. Код шаблона templates\email\activation_letter_subject.txt

```
Активация пользователя {{ user.username }}
```

Листинг 32.20. Код шаблона templates\email\activation_letter_body.txt

```
Уважаемый пользователь {{ user.username }}!

Вы зарегистрировались на сайте "Доска объявлений".
Вам необходимо выполнить активацию, чтобы подтвердить свою личность.
Для этого пройдите, пожалуйста, по ссылке

{{ host }}{% url 'main:register_activate' sign=sign %}

До свидания!

С уважением, администрация сайта "Доска объявлений".
```

32.4.2. Веб-страницы активации пользователя

Чтобы реализовать активацию нового пользователя, мы напишем один контроллер и целых три шаблона. Они создадут страницы с сообщением об успешной активации, о том, что активация была выполнена ранее, и о том, что цифровая подпись у идентификатора пользователя, полученного в составе интернет-адреса, скомпрометирована.

Контроллер мы реализуем в виде функции `user_activate()`. Ее код приведен в листинге 32.21.

Листинг 32.21. Код контроллера-функции `user_activate()`

```
from django.core.signing import BadSignature

from .utilities import signer

def user_activate(request, sign):
    try:
        username = signer.unsign(sign)
    except BadSignature:
        return render(request, 'main/bad_signature.html')
    user = get_object_or_404(AdvUser, username=username)
    if user.is_activated:
        template = 'main/user_is_activated.html'
    else:
        template = 'main/activation_done.html'
        user.is_active = True
        user.is_activated = True
        user.save()
    return render(request, template)
```

Подписанный идентификатор пользователя, передаваемый в составе интернет-адреса, получаем с параметром `sign`. Далее извлекаем из него имя пользователя, ищем пользователя с этим именем, делаем его активным, присвоив значения `True` полям `is_active` и `is_activated` модели, и выводим страницу с сообщением об успешной активации. Если цифровая подпись оказалась скомпрометированной, выводим страницу с сообщением о неуспехе активации, а если пользователь был активирован ранее (поле `is_activated` уже хранит значение `True`) — страницу с сообщением, что активация уже произошла.

Для обработки подписанного значения используем экземпляр класса `Signer`, созданный в модуле utilities.py и хранящийся в переменной `signer`. Так мы сэкономим оперативную память.

В списке маршрутов уровня приложения запишем маршрут, ведущий на контроллер `user_activate()`:

```
from .views import user_activate
. . .
urlpatterns = [
    path('accounts/register/activate/<str:sign>/', user_activate,
                                        name='register_activate'),
    path('accounts/register/done/', RegisterDoneView.as_view(),
                                    name='register_done'),
    . . .
]
```

Код шаблонов templates\main\activation_done.html, templates\main\bad_signature.html и templates\main\user_is_activated.html страниц с сообщениями соответственно об успешной, неуспешной и уже выполненной активации приведен в листингах 32.22—32.24.

Листинг 32.22. Код шаблона templates\main\activation_done.html

```
{% extends "layout/basic.html" %}

{% block title %}Активация выполнена{% endblock %}

{% block content %}
<h2>Активация</h2>
<p>Пользователь с таким именем успешно активирован.</p>
<p><a href="{% url 'main:login' %}">Войти на сайт</a></p>
{% endblock %}
```

Листинг 32.23. Код шаблона templates\main\bad_signature.html

```
{% extends "layout/basic.html" %}

{% block title %}Ошибка при активации{% endblock %}

{% block content %}
<h2>Активация</h2>
<p>Активация пользователя с таким именем прошла неудачно.</p>
<p><a href="{% url 'main:register' %}">Зарегистрироваться
повторно</a></p>
{% endblock %}
```

Листинг 32.24. Код шаблона templates\main\user_is_activated.html

```
{% extends "layout/basic.html" %}

{% block title %}Пользователь уже активирован{% endblock %}

{% block content %}
<h2>Активация</h2>
```

```
<p>Пользователь с таким именем был активирован ранее.</p>
<p><a href="{% url 'main:login' %}">Войти на сайт</a></p>
{% endblock %}
```

Чтобы протестировать отправку электронных писем, воспользуемся отладочным SMTP-сервером (см. *разд. 25.4*). Добавим в модуль settings.py пакета конфигурации выражение, задающее TCP-порт № 1025, который используется этим сервером по умолчанию:

```
EMAIL_PORT = 1025
```

Запустим еще один экземпляр командной строки и отдадим команду, запускающую отладочный SMTP-сервер с использованием TCP-порта № 1025:

```
python -m smtpd -n -c DebuggingServer localhost:1025
```

Сохраним файлы с исходным кодом, перейдем на страницу регистрации, введем сведения о новом пользователе и дождемся письма с требованием активации. Перейдем по находящемуся в этом письме интернет-адресу, удостоверимся, что активация прошла успешно, и попытаемся выполнить вход от имени только что созданного пользователя.

Добавим таким же образом еще двух или трех пользователей — они пригодятся нам для реализации удаления пользователей.

32.5. Веб-страница удаления пользователя

Контроллер-класс `DeleteUserView`, удаляющий текущего пользователя, сделаем производным от класса `DeleteView`. Его код приведен в листинге 32.25.

Листинг 32.25. Код контроллера-класса `DeleteUserView`

```
from django.views.generic.edit import DeleteView
from django.contrib.auth import logout
from django.contrib import messages

class DeleteUserView(LoginRequiredMixin, DeleteView):
    model = AdvUser
    template_name = 'main/delete_user.html'
    success_url = reverse_lazy('main:index')

    def setup(self, request, *args, **kwargs):
        self.user_id = request.user.pk
        return super().setup(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        logout(request)
        messages.add_message(request, messages.SUCCESS,
                                      'Пользователь удален')
        return super().post(request, *args, **kwargs)
```

```
    def get_object(self, queryset=None):
        if not queryset:
            queryset = self.get_queryset()
        return get_object_or_404(queryset, pk=self.user_id)
```

Здесь мы использовали те же программные приемы, что и в контроллере `ChangeUserInfoView` (см. листинг 32.9). В переопределенном методе `setup()` сохранили ключ текущего пользователя, а в переопределенном методе `get_object()` отыскали по этому ключу пользователя, подлежащего удалению.

Перед удалением текущего пользователя необходимо выполнить выход, что мы и сделали в переопределенном методе `post()`. В том же методе `post()` мы создали всплывающее сообщение об успешном удалении пользователя.

В списке маршрутов уровня приложения запишем код, который добавит новый маршрут:

```
from .views import DeleteUserView
. . .
urlpatterns = [
    . . .
    path('accounts/profile/delete/', DeleteUserView.as_view(),
                                     name='profile_delete'),
    path('accounts/profile/change/', ChangeUserInfoView.as_view(),
                                     name='profile_change'),
    . . .
]
```

Напишем шаблон templates\main\delete_user.html страницы для удаления пользователя. Его код приведен в листинге 32.26.

Листинг 32.26. Код шаблона templates\main\delete_user.html

```
{% extends "layout/basic.html" %}

{% load bootstrap4 %}

{% block title %}Удаление пользователя{% endblock %}

{% block content %}
<h2>Удаление пользователя {{ object.username }}</h2>
<form method="post">
    {% csrf_token %}
    {% buttons submit='Удалить' %}{% endbuttons %}
</form>
{% endblock %}
```

Осталось записать в шаблоне templates\layout\basic.html интернет-адрес, ведущий на страницу удаления пользователя:

```
<a . . . href="{% url 'main:profile_delete' %}">Удалить</a>
```

Для проверки попробуем войти на сайт от имени одного из ранее созданных пользователей (только не суперпользователя — иначе придется создавать его заново) и выполнить его удаление.

32.6. Инструменты для администрирования пользователей

Напоследок напишем редактор, посредством которого администрация сайта будет работать с зарегистрированными пользователями. В него добавим возможность фильтрации пользователей по именам, адресам электронной почты, настоящим именам и фамилиям. Также реализуем вывод пользователей, уже выполнивших активацию, не выполнивших ее в течение трех дней и недели, и действие по отправке выбранным пользователям писем с требованиями пройти активацию.

Полный код класса редактора `AdvUserAdmin`, вспомогательного класса и функции приведен в листинге 32.27. Этот код следует занести в модуль admin.py пакета приложения, заменив им имеющийся там код.

Листинг 32.27. Код редактора `AdvUserAdmin`

```
from django.contrib import admin
import datetime

from .models import AdvUser
from .utilities import send_activation_notification

def send_activation_notifications(modeladmin, request, queryset):
    for rec in queryset:
        if not rec.is_activated:
            send_activation_notification(rec)
    modeladmin.message_user(request, 'Письма с требованиями отправлены')
send_activation_notifications.short_description = \
'Отправка писем с требованиями активации'

class NonactivatedFilter(admin.SimpleListFilter):
    title = 'Прошли активацию?'
    parameter_name = 'actstate'

    def lookups(self, request, model_admin):
        return (
                    ('activated', 'Прошли'),
                    ('threedays', 'Не прошли более 3 дней'),
                    ('week', 'Не прошли более недели'),
                )
```

```
    def queryset(self, request, queryset):
        val = self.value()
        if val == 'activated':
            return queryset.filter(is_active=True, is_activated=True)
        elif val == 'threedays':
            d = datetime.date.today() - datetime.timedelta(days=3)
            return queryset.filter(is_active=False, is_activated=False,
                                   date_joined__date__lt=d)
        elif val == 'week':
            d = datetime.date.today() - datetime.timedelta(weeks=1)
            return queryset.filter(is_active=False, is_activated=False,
                                   date_joined__date__lt=d)

class AdvUserAdmin(admin.ModelAdmin):
    list_display = ('__str__', 'is_activated', 'date_joined')
    search_fields = ('username', 'email', 'first_name', 'last_name')
    list_filter = (NonactivatedFilter,)
    fields = (('username', 'email'), ('first_name', 'last_name'),
              ('send_messages', 'is_active', 'is_activated'),
              ('is_staff', 'is_superuser'),
              'groups', 'user_permissions',
              ('last_login', 'date_joined'))
    readonly_fields = ('last_login', 'date_joined')
    actions = (send_activation_notifications,)

admin.site.register(AdvUser, AdvUserAdmin)
```

В списке записей указываем выводить строковое представление записи (имя пользователя — как реализовано в модели `AbstractUser`, от которой наследует наша модель), поле признака, выполнил ли пользователь активацию, временну́ю отметку его регистрации. Также разрешаем выполнять фильтрацию по полям имени, адреса электронной почты, настоящих имени и фамилии.

Для выполнения фильтрации пользователей, выполнивших активацию, не выполнивших ее в течение трех дней и недели, используем класс `NonactivatedFilter`. Обратим внимание на код, непосредственно фильтрующий пользователей по значению даты их регистрации.

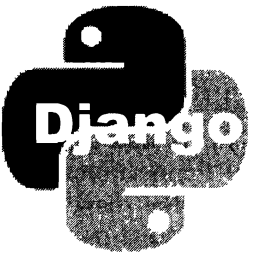
Мы явно указываем список полей, которые должны выводиться в формах для правки пользователей, чтобы выстроить их в удобном для работы порядке. Поля даты регистрации пользователя и последнего его входа на сайт делаем доступными только для чтения.

Наконец, регистрируем действие, которое разошлет пользователям письма с предписаниями выполнить активацию. Это действие реализовано функцией `send_activation_notifications()`. В ней мы перебираем всех выбранных пользователей

и для каждого, кто не выполнил активацию, вызываем функцию `send_activation_notification()`, объявленную ранее в модуле utilities.py и непосредственно производящую отправку писем.

На этом все. Сохраним весь исправленный код и проверим написанный нами редактор в действии.

В качестве домашнего задания реализуйте сброс пароля. Все необходимые для этого сведения были даны в предыдущих главах данной книги, так что вы, уважаемые читатели, справитесь с этим без труда.



ГЛАВА 33

Рубрики

Как условились в *главе 31*, мы реализуем двухуровневую структуру рубрик: более общие рубрики верхнего уровня (*надрубрики*) и вложенные в них рубрики нижнего уровня (*подрубрики*). Список рубрик будет выводиться в вертикальной панели навигации на каждой странице сайта.

33.1. Модели рубрик

Напишем базовую модель, в которой будут храниться и надрубрики, и подрубрики, и две производные от нее прокси-модели: для надрубрик и подрубрик.

33.1.1. Базовая модель рубрик

Базовой модели, в которой будут храниться и надрубрики, и подрубрики, мы дадим имя `Rubric`. Ее структура приведена в табл. 33.1.

***Таблица 33.1.** Структура модели `Rubric`*

Имя	Тип	Дополнительные параметры	Описание
`name`	`CharField`	Длина — 20 символов, индексированное	Название
`order`	`IntegerField`	Значение по умолчанию — 0, индексированное	Порядок
`super_rubric`	`ForeignKey`	Необязательное, запрет каскадного удаления	Надрубрика

Поле `order` будет хранить целое число, обозначающее порядок следования рубрик друг за другом: при выводе рубрики сначала будут сортироваться по возрастанию значения порядка, а уже потом — по их названиям.

Поле `super_rubric` будет хранить надрубрику, к которой относится текущая подрубрика. Оно будет иметь следующие важные особенности:

- связь, создаваемая этим полем, должна устанавливаться с моделью надрубрик, которую мы объявим чуть позже. Условимся называть эту модель `SuperRubric`;
- это поле будет заполняться только в том случае, если запись хранит подрубрику. Если запись хранит надрубрику, то поле заполнять не нужно (собственно, отсутствие значения в этом поле является признаком надрубрики — ведь надрубрика в принципе не может ссылаться на надрубрику). Поэтому данное поле сделано необязательным для заполнения;
- нужно обязательно запретить каскадное удаление записей, чтобы пользователь по ошибке не удалил надрубрику вместе со всеми подрубриками.

Код класса модели `Rubric` приведен в листинге 33.1. Не забываем, что код всех моделей заносится в модуль **models.py** пакета приложения.

Листинг 33.1. Код модели `Rubric`

```
class Rubric(models.Model):
    name = models.CharField(max_length=20, db_index=True, unique=True,
                            verbose_name='Название')
    order = models.SmallIntegerField(default=0, db_index=True,
                                     verbose_name='Порядок')
    super_rubric = models.ForeignKey('SuperRubric',
                   on_delete=models.PROTECT, null=True, blank=True,
                   verbose_name='Надрубрика')
```

Мы не задаем никаких параметров самой модели — поскольку пользователи не будут работать с ней непосредственно, здесь это совершенно излишне.

33.1.2. Модель надрубрик

Для работы с надрубриками объявим прокси-модель `SuperRubric`, производную от `Rubric` (прокси-модель позволяет менять лишь функциональность модели, но не набор объявленных в ней полей, однако нам и надо изменить лишь функциональность модели — см. *разд. 16.4.3*). Она будет обрабатывать только надрубрики.

Чтобы изменить состав обрабатываемых моделью записей, нужно задать для нее свой диспетчер записей, который и укажет необходимые условия фильтрации.

Код обоих классов: и модели `SuperRubric`, и диспетчера записей `SuperRubricManager` — приведен в листинге 33.2.

Листинг 33.2. Код модели `SuperRubric` и диспетчера записей `SuperRubricManager`

```
class SuperRubricManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(super_rubric__isnull=True)

class SuperRubric(Rubric):
    objects = SuperRubricManager()
```

```
    def __str__(self):
        return self.name

    class Meta:
        proxy = True
        ordering = ('order', 'name')
        verbose_name = 'Надрубрика'
        verbose_name_plural = 'Надрубрики'
```

Условия фильтрации записей указываем в переопределенном методе `get_queryset()` класса диспетчера записей `SuperRubricManager`. Он станет выбирать только записи с пустым полем `super_rubric`, т. е. надрубрики.

В самом классе модели `SuperRubric` задаем диспетчер записей `SuperRubricManager` в качестве основного. И не забываем объявить метод `__str__()`, который станет генерировать строковое представление надрубрики — ее название.

Как и условились ранее, указываем сортировку записей сначала по возрастанию значения порядка, а потом — по названию.

33.1.3. Модель подрубрик

Модель подрубрик `SubRubric` мы создадим таким же образом, как и модель надрубрик. Только теперь диспетчер записей, который мы создадим для нее, будет выбирать лишь подрубрики.

Код классов модели `SubRubric` и диспетчера записей `SubRubricManager` приведен в листинге 33.3.

Листинг 33.3. Код модели `SubRubric` и диспетчера записей `SubRubricManager`

```
class SubRubricManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(super_rubric__isnull=False)

class SubRubric(Rubric):
    objects = SubRubricManager()

    def __str__(self):
        return '%s - %s' % (self.super_rubric.name, self.name)

    class Meta:
        proxy = True
        ordering = ('super_rubric__order', 'super_rubric__name', 'order',
                    'name')
        verbose_name = 'Подрубрика'
        verbose_name_plural = 'Подрубрики'
```

Диспетчер записей SubRubricManager будет отбирать лишь записи с непустым полем super_rubric (т. е. подрубрики). Строковое представление, создаваемое моделью, будет выполнено в формате *<название надрубрики> - <название подрубрики>*. А сортировку записей укажем по порядку надрубрики, названию надрубрики, порядку подрубрики и названию подрубрики.

Объявив все необходимые классы, остановим отладочный веб-сервер (если он все еще работает), создадим и выполним миграции:

```
manage.py makemigrations
manage.py migrate
```

33.2. Инструменты для администрирования рубрик

Вся работа с надрубриками и подрубриками будет проводиться средствами административного сайта.

Для надрубрик мы создадим встроенный редактор, чтобы пользователь, добавив новую надрубрику, смог сразу же заполнить ее подрубриками. Из формы для ввода и правки надрубрик мы исключим поле надрубрики (super_rubric), поскольку оно там совершенно не нужно и, более того, собьет пользователя с толку.

В листинге 33.4 приведен код классов редактора SuperRubricAdmin и встроенного редактора SubRubricInline. Не забываем, что код редакторов, равно как и код, регистрирующий модели и редакторы в подсистеме административного сайта, должен записываться в модуль admin.py пакета приложения.

Листинг 33.4. Код редактора SuperRubricAdmin и встроенного редактора SubRubricInline

```
from .models import SuperRubric, SubRubric

class SubRubricInline(admin.TabularInline):
    model = SubRubric

class SuperRubricAdmin(admin.ModelAdmin):
    exclude = ('super_rubric',)
    inlines = (SubRubricInline,)

admin.site.register(SuperRubric, SuperRubricAdmin)
```

У подрубрик сделаем поле надрубрики (super_rubric) обязательным для заполнения. Для этого мы объявим форму SubRubricForm, записав ее код, приведенный в листинге 33.5, в модуле forms.py пакета приложения.

Листинг 33.5. Код формы SubRubricForm

```
from .models import SuperRubric, SubRubric

class SubRubricForm(forms.ModelForm):
    super_rubric = forms.ModelChoiceField(
                   queryset=SuperRubric.objects.all(),empty_label=None,
                   label='Надрубрика', required=True)

    class Meta:
        model = SubRubric
        fields = '__all__'
```

Мы убрали у раскрывающегося списка, с помощью которого пользователь будет выбирать подрубрику, "пустой" пункт, присвоив параметру `empty_label` конструктора класса поля `ModelChoiceField` значение `None`. Так мы дополнительно дадим понять, что в это поле обязательно должно быть занесено значение.

В листинге 33.6 приведен код, объявляющий класс редактора `SubRubricAdmin`.

Листинг 33.6. Код редактора SubRubricAdmin

```
from .forms import SubRubricForm

class SubRubricAdmin(admin.ModelAdmin):
    form = SubRubricForm

admin.site.register(SubRubric, SubRubricAdmin)
```

Сохраним весь исправленный код, запустим отладочный веб-сервер (не забыв при этом отключить обработку статических файлов), войдем на административный сайт и добавим несколько надрубрик и подрубрик.

33.3. Вывод списка рубрик в вертикальной панели навигации

Сначала необходимо поместить в состав контекста каждого шаблона переменную, в которой хранится список подрубрик (именно на его основе мы будем формировать пункты панели навигации). Можно создавать такую переменную в каждом контроллере, но это очень трудоемко. Поэтому объявим и зарегистрируем в проекте обработчик контекста, в котором и будет формироваться список подрубрик.

Условимся, что список подрубрик будет помещаться в переменную `rubrics` контекста шаблона.

Создадим в пакете приложения модуль middlewares.py и запишем в него код обработчика контекста `bboard_context_processor()`, приведенный в листинге 33.7.

Листинг 33.7. Код обработчика контекста bboard_context_processor()

```
from .models import SubRubric

def bboard_context_processor(request):
    context = {}
    context['rubrics'] = SubRubric.objects.all()
    return context
```

Откроем модуль settings.py пакета конфигурации и зарегистрируем только что написанный обработчик контекста. Для этого добавим имя этого обработчика в список context_processors из параметра OPTIONS:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        . . .
        'OPTIONS': {
            'context_processors': [
                . . .
                'main.middlewares.bboard_context_processor',
            ],
        },
    },
]
```

Выполним еще пару подготовительных действий. Во-первых, в модуле views.py пакета приложения объявим "пустой" контроллер-функцию by_rubric():

```
def by_rubric(request, pk):
    pass
```

Во-вторых, добавим в список маршрутов уровня приложения маршрут, ведущий на этот контроллер:

```
from .views import by_rubric
. . .
urlpatterns = [
    . . .
    path('<int:pk>/', by_rubric, name='by_rubric'),
    path('<str:page>/', other_page, name='other'),
    . . .
]
```

Этот маршрут мы поместим перед маршрутом, ведущим на контроллер other_page(), который выводит вспомогательные страницы. Если же мы поместим его после упомянутого ранее маршрута, то при просмотре списка маршрутов Django примет присутствующий в интернет-адресе ключ рубрики за имя шаблона страницы и запустит контроллер other_page(), что приведет к ошибке 404.

Зачем мы объявляли эти контроллер и маршрут? Чтобы прямо сейчас сформировать в панели навигации гиперссылки с правильными интернет-адресами. В *главе 34* мы заменим контроллер-"заглушку" другим, выполняющим полезную работу — вывод объявлений из выбранной посетителем рубрики.

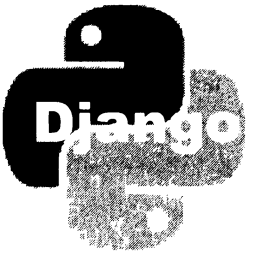
Откроем шаблон layout\basic.html (давайте ради краткости не указывать папку templates в путях к шаблонам — мы уже давно знаем, что шаблоны хранятся в папке templates) и исправим код вертикальной панели навигации следующим образом:

```
<a class="nav-link root" href="{% url 'main:index' %}">Главная</a>
<span class="nav-link root font-weight-bold">Недвижимость</span>
. . .
<a class="nav-link" href="#">Грузовой</a>
{% for rubric in rubrics %}
{% ifchanged rubric.super_rubric.pk %}
<span class="nav-link root font-weight-bold">
{{ rubric.super_rubric.name }}</span>
{% endifchanged %}
<a class="nav-link" href="{% url 'main:by_rubric' pk=rubric.pk %}">
{{ rubric.name }}</a>
{% endfor %}
<a class="nav-link root" href="{% url 'main:other' page='about' %}">О
сайте</a>
```

Мы перебираем список подрубрик, хранящийся в переменной `rubrics` контекста шаблона (эту переменную создал наш обработчик контекста `bboard_context_processor()`), и для каждой подрубрики выводим:

- если ключ связанной надрубрики изменился (т. е. если начали выводиться подрубрики из другой надрубрики) — пункт с именем надрубрики;
- пункт-гиперссылку с именем подрубрики.

Сохраним код, на всякий случай перезапустим отладочный веб-сервер, обновим открытую в веб-обозревателе страницу и полюбуемся на список рубрик. Только не будем пока щелкать на них.



ГЛАВА 34

Объявления

Теперь можно приступать к работе над объявлениями. Мы создадим страницу для просмотра объявлений, относящихся к выбранной рубрике, с поддержкой пагинации и поиска, страницу сведений о выбранном объявлении, страницы для добавления, правки и удаления объявлений. И не забудем вывести на странице профиля объявления, оставленные текущим пользователем.

34.1. Подготовка к обработке выгруженных файлов

В составе каждого объявления будет присутствовать графическое изображение с основной иллюстрацией к продаваемому товару. Помимо этого, пользователь может создать в объявлении произвольное количество дополнительных иллюстраций.

Чтобы Django смог обработать выгруженные посетителями файлы, необходимо установить три дополнительных библиотеки: Easy Thumbnails (создает миниатюры), django-cleanup (удаляет выгруженные файлы после удаления хранящих их записей моделей) и Pillow (обеспечивает поддержку графики, будет автоматически установлена при установке Easy Thumbnails). Установим их, набрав команды:

```
pip install easy-thumbnails
pip install django-cleanup
```

Добавим программные ядра двух последних библиотек: приложения `easy_thumbnails` и `django_cleanup` — в список зарегистрированных в проекте. Для этого откроем модель settings.py пакета конфигурации и добавим в список параметра `INSTALLED_APPS` псевдонимы этих приложений:

```
INSTALLED_APPS = [
    . . .
    'django_cleanup',
    'easy_thumbnails',
]
```

Для хранения самих выгруженных файлов отведем папку media, которую создадим в папке проекта. Для хранения миниатюр создадим в ней папку thumbnails.

В модуле settings.py укажем путь к папке media и префикс для интернет-адресов выгруженных файлов:

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

И сразу же добавим туда настройки приложения easy_thumbnails:

```
THUMBNAIL_ALIASES = {
    '': {
        'default': {
            'size': (96, 96),
            'crop': 'scale',
        },
    },
}
THUMBNAIL_BASEDIR = 'thumbnails'
```

Мы задали для миниатюр один-единственный пресет, указывающий выполнять простое масштабирование до размеров 96×96 пикселов. Также мы задали имя вложенной папки, в которой будут храниться миниатюры, — thumbnails.

И наконец, добавим в список маршрутов уровня проекта маршрут для обработки выгруженных файлов:

```
from django.conf.urls.static import static

urlpatterns = [
    . . .
]

if settings.DEBUG:
    urlpatterns.append(path('static/<path:path>', never_cache(serve)))
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

34.2. Модели объявлений и дополнительных иллюстраций

Мы создадим две модели: одну — для объявлений и вторую — для дополнительных иллюстраций.

34.2.1. Модель самих объявлений

Модель, хранящая объявления, будет называться Bb. Ее структура приведена в табл. 34.1.

Таблица 34.1. *Структура модели* `Bb`

Имя	Тип	Дополнительные параметры	Описание
`rubric`	`ForeignKey`	Запрет каскадного удаления	Подрубрика
`title`	`CharField`	Длина — 40 символов	Название товара
`content`	`TextField`		Описание товара
`price`	`IntegerField`	Значение по умолчанию — 0	Цена товара
`contacts`	`TextField`		Контакты
`image`	`ImageField`	Необязательное	Основная иллюстрация к объявлению
`author`	`ForeignKey`		Пользователь, оставивший объявление
`is_active`	`BooleanField`	Значение по умолчанию — `True`, индексированное	Признак, показывать ли объявление в списке
`created_at`	`DateTimeField`	Подставляется текущее значение даты и времени, индексированное	Дата и время публикации объявления

В поле `rubric`, устанавливающем связь с моделью подрубрик `SubRubric`, мы указали запрет каскадного удаления, чтобы предотвратить случайное удаление подрубрики вместе со всеми относящимися к ней объявлениями.

Графические файлы, сохраняемые в поле `image` модели, будут иметь в качестве имен текущие временны́е отметки. Так мы приведем имена к единому типу и заодно устраним ситуацию, когда имя выгруженного файла настолько длинное, что оно не помещается в поле модели.

При разработке модели объявления нужно учесть еще один момент. Чуть позже мы напишем модель дополнительных иллюстраций, которую свяжем с моделью объявлений связью "один-со-многими". Если при объявлении этой связи мы разрешим каскадное удаление, то при удалении объявления будут уничтожены все относящиеся к нему дополнительные иллюстрации. Но это действие выполнит не Django, а СУБД, отчего приложение `django_cleanup` не получит сигнала об удалении записей и не сможет в ответ удалить хранящиеся в них графические файлы. В результате эти файлы останутся на диске бесполезным мусором.

Мы обязательно решим эту проблему позже. А сейчас условимся об имени модели дополнительных иллюстраций — `AdditionalImage`.

В *главе 32* мы создали в пакете приложения модуль utilities.py, в который записали объявление функции, выполняющей отправку писем. Этот модуль — отличное место для сохранения кода, не относящегося напрямую ни к моделям, ни к редакторам, ни к контроллерам. Поместим в него объявление функции `get_timestamp_path()`, генерирующей имена сохраняемых в модели выгруженных файлов (листинг 34.1).

Листинг 34.1. Код функции get_timestamp_path()

```
from datetime import datetime
from os.path import splitext

def get_timestamp_path(instance, filename):
    return '%s%s' % (datetime.now().timestamp(), splitext(filename)[1])
```

В листинге 34.2 приведен код, объявляющий сам класс модели Bb.

Листинг 34.2. Код модели Bb

```
from .utilities import get_timestamp_path

class Bb(models.Model):
    rubric = models.ForeignKey(SubRubric, on_delete=models.PROTECT,
                                          verbose_name='Рубрика')
    title = models.CharField(max_length=40, verbose_name='Товар')
    content = models.TextField(verbose_name='Описание')
    price = models.FloatField(default=0, verbose_name='Цена')
    contacts = models.TextField(verbose_name='Контакты')
    image = models.ImageField(blank=True, upload_to=get_timestamp_path,
                              verbose_name='Изображение')
    author = models.ForeignKey(AdvUser, on_delete=models.CASCADE,
                               verbose_name='Автор объявления')
    is_active = models.BooleanField(default=True, db_index=True,
                                    verbose_name='Выводить в списке?')
    created_at = models.DateTimeField(auto_now_add=True, db_index=True,
                                      verbose_name='Опубликовано')

    def delete(self, *args, **kwargs):
        for ai in self.additionalimage_set.all():
            ai.delete()
        super().delete(*args, **kwargs)

    class Meta:
        verbose_name_plural = 'Объявления'
        verbose_name = 'Объявление'
        ordering = ['-created_at']
```

В переопределенном методе delete() перед удалением текущей записи мы перебираем и вызовом метода delete() удаляем все связанные дополнительные иллюстрации. При вызове метода delete() возникает сигнал post_delete, обрабатываемый приложением django_cleanup, которое в ответ удалит все файлы, хранящиеся в удаленной записи.

34.2.2. Модель дополнительных иллюстраций

Модель дополнительных иллюстраций мы назовем, как условились в *разд. 34.2.1*, `AdditionalImage`. Ее структура приведена в табл. 34.2.

Таблица 34.2. *Структура модели* `AdditionalImage`

Имя	Тип	Описание
bb	ForeignKey	Объявление, к которому относится иллюстрация
image	ImageField	Собственно иллюстрация

Графические файлы, сохраняемые в поле `image`, также получат в качестве имен текущие временны́е отметки. Для формирования имен файлов применим объявленную ранее функцию `get_timestamp_path()` из модуля utilities.py.

Готовый код модели `AdditionalImage` приведен в листинге 34.3.

Листинг 34.3. Код модели `AdditionalImage`

```
class AdditionalImage(models.Model):
    bb = models.ForeignKey(Bb, on_delete=models.CASCADE,
                           verbose_name='Объявление')
    image = models.ImageField(upload_to=get_timestamp_path,
                              verbose_name='Изображение')

    class Meta:
        verbose_name_plural = 'Дополнительные иллюстрации'
        verbose_name = 'Дополнительная иллюстрация'
```

Сохраним код моделей, создадим и выполним миграции. И сделаем еще одно очень важное дело.

34.2.3. Реализация удаления объявлений в модели пользователя

В *разд. 34.2.1* мы сделали так, чтобы при удалении объявления явно удалялись все связанные с ним дополнительные иллюстрации. Это нужно для того, чтобы приложение `django_cleanup` удалило хранящие их файлы.

Теперь сделаем так, чтобы при удалении пользователя удалялись оставленные им объявления. Для этого добавим в код модели `AdvUser` следующий фрагмент:

```
class AdvUser(AbstractUser):
    . . .
    def delete(self, *args, **kwargs):
        for bb in self.bb_set.all():
            bb.delete()
        super().delete(*args, **kwargs)
```

```
    class Meta(AbstractUser.Meta):
        pass
```

34.3. Инструменты для администрирования объявлений

Чтобы с объявлениями можно было работать посредством административного сайта, объявим редактор объявлений `BbAdmin` и встроенный редактор дополнительных иллюстраций `AdditionalImageInline`. Их код приведен в листинге 34.4.

Листинг 34.4. Код редактора `BbAdmin` и встроенного редактора `AdditionalImageInline`

```
from .models import Bb, AdditionalImage

class AdditionalImageInline(admin.TabularInline):
    model = AdditionalImage

class BbAdmin(admin.ModelAdmin):
    list_display = ('rubric', 'title', 'content', 'author', 'created_at')
    fields = (('rubric', 'author'), 'title', 'content', 'price',
              'contacts', 'image', 'is_active')
    inlines = (AdditionalImageInline,)

admin.site.register(Bb, BbAdmin)
```

На страницах добавления и правки объявлений выведем раскрывающиеся списки подрубрики и пользователя в одну строку — ради компактности.

Сохраним код, запустим отладочный веб-сервер, войдем на административный сайт и добавим несколько объявлений, обязательно с дополнительными иллюстрациями. Попробуем исправить одно объявление и удалить другое, проверив, действительно ли при этом будут удалены все файлы, хранящиеся в самом объявлении, и связанные с ним иллюстрации.

34.4. Вывод объявлений

Мы создадим две страницы:

- страницу списка объявлений, относящихся к выбранной посетителем рубрике, с поддержкой пагинации и поиска объявлений по введенному слову;
- страницу сведений о выбранном объявлении, на которой будут выводиться также и дополнительные иллюстрации.

Кроме того, реализуем вывод десяти наиболее "свежих" объявлений на главной странице.

34.4.1. Вывод списка объявлений

Вывод списка объявлений с поддержкой поиска — достаточно сложная задача. Нам понадобится обычная, не связанная с моделью, форма для ввода искомого слова, контроллер и шаблон. А еще придется решить весьма серьезную проблему корректного возврата, о которой мы поговорим чуть позже.

34.4.1.1. Форма поиска и контроллер списка объявлений

Сразу условимся, что искомое слово, введенное посетителем, будем пересылать контроллеру методом GET в GET-параметре `keyword`. Поле в форме для ввода искомого слова назовем так же.

Код формы поиска `SearchForm` очень прост — убедимся в этом сами, взглянув на листинг 34.5.

Листинг 34.5. Код формы `SearchForm`

```
class SearchForm(forms.Form):
    keyword = forms.CharField(required=False, max_length=20, label='')
```

Поскольку посетитель может ввести в поле `keyword` искомое слово, а может и не ввести (чтобы отменить выполненный ранее поиск и вновь вывести все объявления из списка), мы пометили это поле как необязательное для заполнения. А еще убрали у этого поля надпись, присвоив параметру `label` пустую строку — все равно такого рода поля выводятся без надписей.

В *главе 33*, чтобы проверить написанный тогда код, мы создали ничего не делающий контроллер-функцию `by_rubric()`. Настала пора "наполнить" его полезным кодом, приведенным в листинге 34.6.

Листинг 34.6. Код контроллера-функции `by_rubric()`

```
from django.core.paginator import Paginator
from django.db.models import Q

from .models import SubRubric, Bb
from .forms import SearchForm

def by_rubric(request, pk):
    rubric = get_object_or_404(SubRubric, pk=pk)
    bbs = Bb.objects.filter(is_active=True, rubric=pk)
    if 'keyword' in request.GET:
        keyword = request.GET['keyword']
        q = Q(title__icontains=keyword) | Q(content__icontains=keyword)
        bbs = bbs.filter(q)
    else:
        keyword = ''
```

```
    form = SearchForm(initial={'keyword': keyword})
    paginator = Paginator(bbs, 2)
    if 'page' in request.GET:
        page_num = request.GET['page']
    else:
        page_num = 1
    page = paginator.get_page(page_num)
    context = {'rubric': rubric, 'page': page, 'bbs': page.object_list,
               'form': form}
    return render(request, 'main/by_rubric.html', context)
```

Извлекаем выбранную посетителем рубрику — нам понадобится вывести на странице ее название. Затем выбираем объявления, относящиеся к этой рубрике и помеченные для вывода (те, у которых поле `is_active` хранит значение `True`). После этого выполняем фильтрацию уже отобранных объявлений по введенному посетителем искомому слову, взятому из GET-параметра `keyword`.

Вот фрагмент кода, "отвечающий" за фильтрацию объявлений по введенному посетителем слову:

```
if 'keyword' in request.GET:
    keyword = request.GET['keyword']
    q = Q(title__icontains=keyword) | Q(content__icontains=keyword)
    bbs = bbs.filter(q)
else:
    keyword = ''
form = SearchForm(initial={'keyword': keyword})
```

Ради простоты получаем искомое слово непосредственно из GET-параметра `keyword`. Затем формируем на основе полученного слова условие фильтрации, применив объект `Q`, и выполняем фильтрацию объявлений.

Далее создаем экземпляр формы `SearchForm`, чтобы вывести ее на экран. Конструктору ее класса в параметре `initial` передаем полученное ранее искомое слово, чтобы оно присутствовало в выведенной на экран форме.

Не забываем создать пагинатор, указав у него количество записей в части равным `2`. Это позволит нам проверить, работает ли пагинация, имея в базе данных всего три-четыре объявления. Наконец, выводим страницу со списком объявлений, применив шаблон main\by_rubric.html. С написанием которого придется подождать...

34.4.1.2. Реализация корректного возврата

Предположим, что мы написали весь код, который будет выводить на экран и списки объявлений, разбитые на рубрики, и сведения о выбранном объявлении. И вот посетитель заходит на сайт, выбирает какую-либо рубрику, пролистывает несколько частей, сформированных пагинатором, находит нужное ему объявление и щелкает на гиперссылке, чтобы просмотреть это объявление полностью. Открывается страница со сведениями об объявлении, посетитель смотрит их, после чего щелкает

на гиперссылке возврата на список объявлений... И попадает на самую первую часть этого списка.

То же самое произойдет, если посетитель выполнит поиск, а уже потом отправится смотреть сведения о каком-либо объявлении. Когда он щелкнет на гиперссылке возврата, то вернется в изначальный список объявлений, в котором не был выполнен поиск.

Как избежать этой проблемы, в общем, понятно. Номер выводимой части и искомое слово у нас передаются посредством GET-параметров `page` и `keyword` соответственно. Тогда, чтобы вернуться на нужную часть списка уже отфильтрованных по заданному слову объявлений, следует передать эти параметры странице сведений об объявлении.

Конечно, готовый набор GET-параметров можно получить из элемента с ключом `QUERY_STRING` словаря, который хранится в атрибуте `META` объекта запроса. Но нет смысла передавать параметр `page`, если его значение равно 1, и параметр `keyword` с "пустой" строкой — это их значения по умолчанию.

Также можно формировать набор GET-параметров в контроллере. Но, по принятым в Django соглашениям, весь код, "ответственный" за формирование страниц, следует помещать в шаблон, посредник или — наш случай! — обработчик контекста.

Откроем модуль middlewares.py пакета приложения, найдем код обработчика контекста `bboard_context_processor()`, написанный в *главе 33*, и вставим в него следующий фрагмент:

```
def bboard_context_processor(request):
    context = {}
    context['rubrics'] = SubRubric.objects.all()
    context['keyword'] = ''
    context['all'] = ''
    if 'keyword' in request.GET:
        keyword = request.GET['keyword']
        if keyword:
            context['keyword'] = '?keyword=' + keyword
            context['all'] = context['keyword']
    if 'page' in request.GET:
        page = request.GET['page']
        if page != '1':
            if context['all']:
                context['all'] += '&page=' + page
            else:
                context['all'] = '?page=' + page
    return context
```

Добавленный код создаст в контексте шаблона две переменные:

- `keyword` — с GET-параметром `keyword`, который понадобится для генерирования интернет-адресов в гиперссылках пагинатора;

- `all` — с GET-параметрами `keyword` и `page`, которые мы добавим к интернет-адресам гиперссылок, указывающих на страницы сведений об объявлениях.

34.4.1.3. Шаблон веб-страницы списка объявлений

Код шаблона main\by_rubric.html, который сформирует страницу списка объявлений, приведен в листинге 34.7.

Листинг 34.7. Код шаблона main\by_rubric.html

```
{% extends "layout/basic.html" %}

{% load thumbnail %}
{% load static %}
{% load bootstrap4 %}

{% block title %}{{ rubric }}{% endblock %}

{% block content %}
<h2 class="mb-2">{{ rubric }}</h2>
<div class="container-fluid mb-2">
    <div class="row">
        <div class="col"> </div>
        <form class="col-md-auto form-inline">
            {% bootstrap_form form show_label=False %}
            {% bootstrap_button content='Искать' button_type='submit' %}
        </form>
    </div>
</div>
{% if bbs %}
<ul class="list-unstyled">
    {% for bb in bbs %}
    <li class="media my-5 p-3 border">
        {% url 'main:detail' rubric_pk=rubric.pk pk=bb.pk as url %}
        <a href="{{ url }}{{ all }}">
        {% if bb.image %}
        <img class="mr-3" src="{% thumbnail bb.image 'default' %}">
        {% else %}
        <img class="mr-3" src="{% static 'main/empty.jpg' %}">
        {% endif %}
        </a>
        <div class="media-body">
          <h3><a href="{{ url }}{{ all }}">
          {{ bb.title }}</a></h3>
          <div>{{ bb.content }}</div>
          <p class="text-right font-weight-bold">{{ bb.price }} руб.</p>
          <p class="text-right font-italic">{{ bb.created_at }}</p>
```

```
        </div>
    </li>
    {% endfor %}
</ul>
{% bootstrap_pagination page url=keyword %}
{% endif %}
{% endblock %}
```

Чтобы вывести форму поиска, прижав ее к правой части страницы, используем конструкцию следующего вида:

```
<div class="container-fluid mb-2">
    <div class="row">
        <div class="col"> </div>
        <form class="col-md-auto . . .">
            . . .
        </form>
    </div>
</div>
```

Знакомый нам по *главе 31* стилевой класс `container-fluid` заставляет элемент, к которому он привязан, вести себя как обычная таблица HTML (стилевой класс `mb-3` устанавливает средней величины внешний отступ снизу, отделяющий форму от собственно списка объявлений). Элемент с привязанным стилевым классом `row`, вложенный в этот элемент, ведет себя как строка таблицы. Вложенный в такую "строку" элемент со стилевым классом `col` ведет себя как ячейка таблицы, растягивающаяся на всю доступную ширину, а элемент со стилевым классом `col-md-auto` — как ячейка с шириной, равной ширине ее содержимого.

Первым элементом-"ячейкой" у нас является пустой блок, а вторым — форма поиска. В результате на экране мы увидим лишь форму, сдвинутую к правому краю страницы.

Посмотрим на код самой формы:

```
<form class=". . . form-inline">
    {% bootstrap_form form show_label=False %}
    {% bootstrap_button content='Искать' button_type='submit' %}
</form>
```

Стилевой класс `form-inline` укажет веб-обозревателю вывести все элементы управления формы в одну строку. В самой форме мы не помещаем электронный жетон защиты (который генерируется тегом шаблонизатора `csrf_token`), поскольку он там совершенно не нужен, а для вывода кнопки применяем тег `bootstrap_button` (использованный ранее тег `buttons` добавляет лишний блок, который в этом случае также не нужен).

Для вывода очередной части списка объявлений применяем особые перечни Bootstrap. Взглянем на код, который их формирует:

```
<ul class="list-unstyled">
    <li class="media my-5 p-3 border">
        <img class="mr-3" . . .>
        <div class="media-body">
            . . .
        </div>
    </li>
</ul>
```

Сам перечень создается маркированным списком HTML (тегом `<ul>`) со стилевым классом `list-unstyled`. Отдельная позиция перечня формируется пунктом списка (тегом `<li>`) со стилевым классом `media` (стилевой класс `my-5` задает большие внешние отступы сверху и снизу, стилевой класс `p-3` — внутренние отступы среднего размера со всех сторон, а стилевой класс `border` — рамку вокруг элемента). В пункте помещается тег `<img>` со стилевым классом `mr-3` и произвольное количество других элементов.

Гиперссылку на страницу сведений об объявлении создаем на базе основной иллюстрации и названия товара. Чтобы не генерировать интернет-адрес для этих гиперссылок дважды, сохраним его в переменной `url`:

```
{% url 'main:detail' rubric_pk=rubric.pk pk=bb.pk as url %}
```

Основная иллюстрация к объявлению у нас является необязательной к указанию. Поэтому нужно предусмотреть случай, когда пользователь оставит объявление без основной иллюстрации. Для этого мы написали такой код:

```
<a href="{{ url }}{{ all }}">
{% if bb.image %}
<img class="mr-3" src="{% thumbnail bb.image 'default' %}">
{% else %}
<img class="mr-3" src="{% static 'main/empty.jpg' %}">
{% endif %}
</a>
```

Если основная иллюстрация в объявлении указана, то будет выведена ее миниатюра. Если же автору объявления нечем его иллюстрировать, будет выведено изображение из статического файла main\empty.jpg.

Теперь посмотрим на самую первую строку приведенного ранее фрагмента. Там, в создающем гиперссылку теге `<a>`, мы сформировали интернет-адрес, объединив содержимое только что созданной в шаблоне переменной `url` и переменной `all` контекста шаблона, в которой хранятся GET-параметры `keyword` и `page`. В результате интернет-адрес гиперссылки включит искомое слово и номер части.

Еще не знакомый нам стилевой класс `font-italic` задает для элемента курсивное начертание шрифта.

Напоследок посмотрим на тег шаблонизатора, создающий пагинатор:

```
{% bootstrap_pagination page url=keyword %}
```

Базовый интернет-адрес берется из переменной `keyword` контекста шаблона, в которой хранится одноименный GET-параметр с искомым словом. В результате при переходе на другую часть пагинатора контроллер получит это слово и выведет отфильтрованный по нему список объявлений.

Осталось найти в Интернете какое-либо подходящее графическое изображение и сохранить его под именем empty.jpg в папке static\main папки проекта. Если же найденное изображение хранится в формате, отличном от JPEG, необходимо соответственно изменить расширение имени файла в коде шаблона.

34.4.2. Веб-страница сведений о выбранном объявлении

Эту страницу будет выводить контроллер-функция `detail()`, который мы напишем позже. А сейчас запишем ведущий на него маршрут, поместив его непосредственно перед маршрутом, ведущим на контроллер `by_rubric()`:

```
from .views import detail
. . .
urlpatterns = [
    . . .
    path('<int:rubric_pk>/<int:pk>/', detail, name='detail'),
    path('<int:pk>/', by_rubric, name='by_rubric'),
    . . .
]
```

Записанные в обоих маршрутах шаблонные пути показывают своего рода иерархию рубрик и отдельных объявлений:

- страницы со списками объявлений, принадлежащих определенной *рубрике*, имеют шаблонные пути формата ***<рубрика>/***;
- страницы с отдельными объявлениями, принадлежащими определенной рубрике, имеют шаблонные пути формата ***<рубрика>/<объявление>/***.

Код контроллера `detail()` приведен в листинге 34.8. Реализуем его в виде функции, поскольку в *главе 35* будем формировать в нем еще и список комментариев к объявлению, а в контроллере-функции это сделать проще.

Листинг 34.8. Код контроллера-функции `detail()`

```
def detail(request, rubric_pk, pk):
    bb = get_object_or_404(Bb, pk=pk)
    ais = bb.additionalimage_set.all()
    context = {'bb': bb, 'ais': ais}
    return render(request, 'main/detail.html', context)
```

Помимо самого объявления, которое мы помещаем в переменную `bb` контекста шаблона, также готовим перечень связанных с ним дополнительных иллюстраций, записав его в переменную `ais`.

Код шаблона main\detail.html, выводящего страницу сведений об объявлении, приведен в листинге 34.9.

Листинг 34.9. Код шаблона main\detail.html

```
{% extends "layout/basic.html" %}

{% block title %}{{ bb.title }} - {{ bb.rubric.name }}{% endblock %}

{% block content %}
<div class="container-fluid mt-3">
    <div class="row">
        {% if bb.image %}
        <div class="col-md-auto"><img src="{{ bb.image.url }}"
        class="main-image"></div>
        {% endif %}
        <div class="col">
            <h2>{{ bb.title }}</h2>
            <p>{{ bb.content }}</p>
            <p class="font-weight-bold">{{ bb.price }} руб.</p>
            <p>{{ bb.contacts }}</p>
            <p class="text-right font-italic">Добавлено
            {{ bb.created_at }}</p>
        </div>
    </div>
</div>
{% if ais %}
<div class="d-flex justify-content-between flex-wrap mt-5">
    {% for ai in ais %}
    <div>
        <img class="additional-image" src="{{ ai.image.url }}">
    </div>
    {% endfor %}
</div>
{% endif %}
<p><a href="{% url 'main:by_rubric' pk=bb.rubric.pk %}{{ all }}">
Назад</a></p>
{% endblock %}
```

Код, выводящий основные сведения об объявлении (название, описание и цену товара, контакты, временну́ю отметку добавления объявления, основную иллюстрацию, если таковая указана), располагается между вот этими тегами:

```
<div class="container-fluid mt-3">
    . . .
</div>
```

Аналогичный код мы рассматривали уже дважды (причем второй раз — непосредственно в текущей главе), так что он уже должен быть нам знаком.

А вот код, выводящий дополнительные иллюстрации, заслуживает более пристального рассмотрения. Вот он:

```
<div class="d-flex justify-content-between flex-wrap mt-5">
    {% for ai in ais %}
    <div>
        <img class="additional-image" src="{{ ai.image.url }}">
    </div>
    {% endfor %}
</div>
```

К блоку, в котором будут выводиться дополнительные иллюстрации, привязаны следующие стилевые классы:

- `d-flex` — устанавливает для элемента так называемую *гибкую* разметку, при которой дочерние элементы выстраиваются внутри родителя по горизонтали;
- `justify-content-between` — указывает, что дочерние элементы (собственно дополнительные иллюстрации) должны располагаться внутри родителя на равномерном расстоянии друг от друга;
- `flex-wrap` — если дочерним элементам не хватит места, чтобы выстроиться по горизонтали, то не помещающиеся элементы будут перенесены на следующую строку;
- `mt-5` — большой внешний отступ сверху, чтобы отделить дополнительные иллюстрации от основной информации.

Осталось установить ширину основной и дополнительных иллюстраций — соответственно 300 и 180 пикселов. Откроем таблицу стилей main\style.css и добавим в нее следующий фрагмент кода:

```
img.main-image {
    width: 300px;
}
img.additional-image {
    width: 180px;
}
```

Сохраним весь код, перезапустим отладочный веб-сервер и зайдем на страницу со списком объявлений, относящихся к какой-либо из категорий (рис. 34.1). После чего попробуем перейти на страницу со сведениями об объявлении (рис. 34.2).

34.4.3. Вывод последних 10 объявлений на главной веб-странице

Найдем код контроллера-функции `index()`, который выводит главную страницу, и добавим в него фрагмент, выбирающий из базы последние 10 объявлений:

```
def index(request):
    bbs = Bb.objects.filter(is_active=True)[:10]
    context = {'bbs': bbs}
    return render(request, 'main/index.html', context)
```

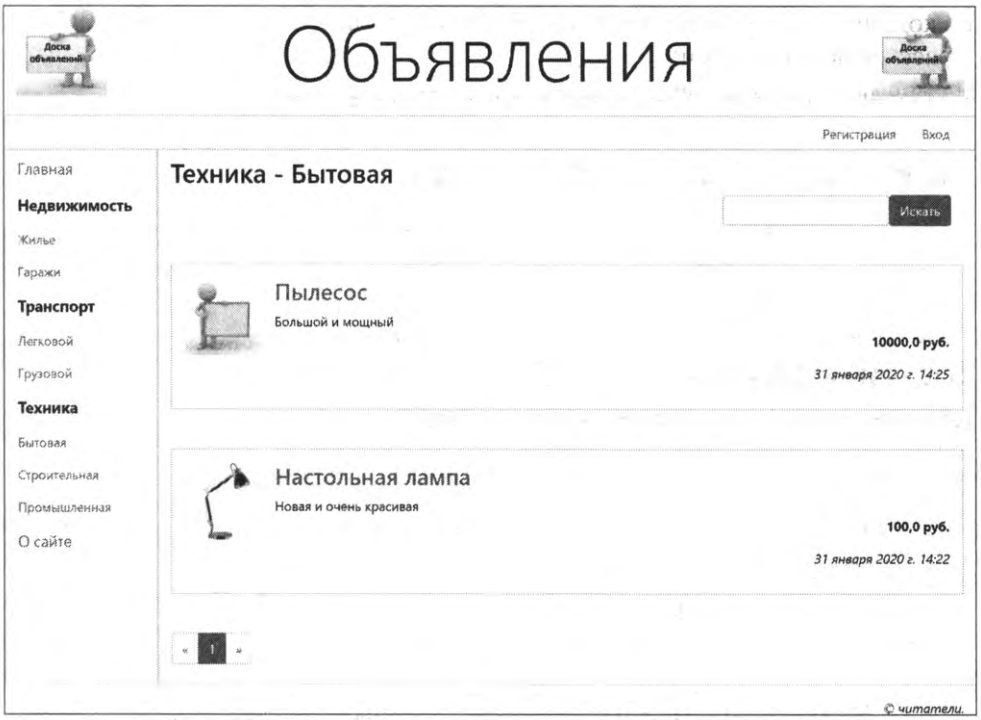


Рис. 34.1. Веб-страница списка объявлений

Рис. 34.2. Веб-страница сведений об объявлении

Осталось дописать в шаблоне main\index.html код, выводящий объявления. Сделайте это самостоятельно. За основу можете взять аналогичный код из шаблона main\by_rubric.html, приведенный в листинге 34.7.

34.5. Работа с объявлениями

Осталось подготовить инструменты, посредством которых зарегистрированные пользователи будут просматривать перечень своих объявлений, добавлять, править и удалять объявления.

34.5.1. Вывод объявлений, оставленных текущим пользователем

Сначала добавим в контроллер-функцию `profile()` следующий код:

```
@login_required
def profile(request):
    bbs = Bb.objects.filter(author=request.user.pk)
    context = {'bbs': bbs}
    return render(request, 'main/profile.html', context)
```

Он фильтрует объявления по значению поля `author` (ключ автора объявления, которым является зарегистрированный пользователь), сравнивая это значение с ключом текущего пользователя.

Для вывода списка объявлений мы вставим в шаблон main\profile.html, формирующий страницу пользовательского профиля, код, который мы практически без изменений можем взять из шаблона main\by_rubric.html (см. листинг 34.7).

Если пользователь зайдет на страницу своего профиля и щелкнет на объявлении, то он попадет на общедоступную страницу сведений об объявлении, написанную в *разд. 34.4.2*. Возможно, это окажется не очень удобно, так что давайте создадим другую, административную страницу сведений об объявлении, доступную лишь зарегистрированным пользователям.

Сначала объявим маршрут, который укажет на эту страницу, поместив его непосредственно перед маршрутом, ведущим на страницу профиля:

```
from .views import profile_bb_detail
. . .
urlpatterns = [
    . . .
    path('accounts/profile/<int:pk>/', profile_bb_detail,
                                       name='profile_bb_detail'),
    path('accounts/profile/', profile, name='profile'),
    . . .
]
```

Контроллер-функцию `profile_bb_detail()`, который выведет страницу сведений об объявлении, напишем самостоятельно, взяв за основу контроллер `detail()`

(см. листинг 34.8). Не забудем сделать его доступным только для зарегистрированных пользователей, выполнивших вход. Также самостоятельно напишем шаблон, который формирует эту страницу и за основу которого можно взять шаблон main\detail.html (см. листинг 34.9).

34.5.2. Добавление, правка и удаление объявлений

Объявим форму `BbForm`, связанную с моделью `Bb`, для ввода самого объявления и встроенный набор форм `AIFormSet`, связанный с моделью `AdditionalImage`, в которые будут заноситься дополнительные иллюстрации. Объявляющий их код приведен в листинге 34.10.

Листинг 34.10. Код формы `BbForm` и набора форм `AIFormSet`

```
from django.forms import inlineformset_factory

from .models import Bb, AdditionalImage

class BbForm(forms.ModelForm):
    class Meta:
        model = Bb
        fields = '__all__'
        widgets = {'author': forms.HiddenInput}

AIFormSet = inlineformset_factory(Bb, AdditionalImage, fields='__all__')
```

В форме будем выводить все поля модели `Bb`. Для поля автора объявления `author` зададим в качестве элемента управления `HiddenInput`, т. е. скрытое поле — все равно значение туда будет заноситься программно.

Контроллер, добавляющий объявление, реализуем в виде функции (в виде класса его реализовать будет сложнее) и назовем `profile_bb_add()`. Его код приведен в листинге 34.11.

Листинг 34.11. Код контроллера-функции `profile_bb_add()`

```
from django.shortcuts import redirect

from .forms import BbForm, AIFormSet

@login_required
def profile_bb_add(request):
    if request.method == 'POST':
        form = BbForm(request.POST, request.FILES)
        if form.is_valid():
            bb = form.save()
            formset = AIFormSet(request.POST, request.FILES, instance=bb)
```

```
            if formset.is_valid():
                formset.save()
                messages.add_message(request, messages.SUCCESS,
                                     'Объявление добавлено')
                return redirect('main:profile')
    else:
        form = BbForm(initial={'author': request.user.pk})
        formset = AIFormSet()
    context = {'form': form, 'formset': formset}
    return render(request, 'main/profile_bb_add.html', context)
```

Здесь нужно отметить три важных момента. Во-первых, при создании формы перед выводом страницы сохранения мы заносим в поле `author` формы ключ текущего пользователя, который станет автором объявления.

Во-вторых, во время сохранения введенного объявления, при создании объектов формы и набора форм, мы должны передать конструкторам их классов вторым позиционным параметром словарь со всеми полученными файлами (он хранится в атрибуте `FILES` объекта запроса). Если мы не сделаем этого, то отправленные пользователем иллюстрации окажутся потерянными.

В-третьих, при сохранении мы сначала выполняем валидацию и сохранение формы самого объявления. Метод `save()` в качестве результата возвращает сохраненную запись, и эту запись мы должны передать через параметр `instance` конструктору класса набора форм. Это нужно для того, чтобы все дополнительные иллюстрации после сохранения оказались связанными с объявлением.

Напишем маршрут, который укажет на страницу добавления, поместив его перед маршрутом, указывающим на страницу профиля:

```
from .views import profile_bb_add
. . .
urlpatterns = [
    . . .
    path('accounts/profile/add/', profile_bb_add, name='profile_bb_add'),
    path('accounts/profile/<int:pk>/', profile_bb_detail,
                                       name='profile_bb_detail'),
    . . .
]
```

Займемся шаблоном main\profile_bb_add.html, который создаст страницу добавления объявления. Его код приведен в листинге 34.12.

Листинг 34.12. Код шаблона main\profile_bb_add.html

```
{% extends "layout/basic.html" %}

{% load bootstrap4 %}
```

```
{% block title %}Добавление объявления - Профиль пользователя{% endblock %}

{% block content %}
<h2>Добавление объявления</h2>
<form method="post" enctype="multipart/form-data">
    {% csrf_token %}
    {% bootstrap_form form layout='horizontal' %}
    {% bootstrap_formset formset layout='horizontal' %}
    {% buttons submit='Добавить' %}{% endbuttons %}
</form>
{% endblock %}
```

Обязательно укажем у формы метод кодирования данных `multipart/form-data`. Если этого не сделать, то занесенные в форму файлы не будут отправлены. А набор форм выведем с помощью тега шаблонизатора `bootstrap_formset`.

Наконец, в шаблон страницы профиля main\profile.html добавим гиперссылку на страницу добавления объявления:

```
<p><a href="{% url 'main:profile_bb_add' %}">Добавить объявление</a></p>
```

После этого можно сохранить код и попробовать функциональность по добавлению новых объявлений в деле.

Код контроллеров `profile_bb_change()` и `profile_bb_delete()`, которые соответственно правят и удаляют объявление, приведен в листинге 34.13.

Листинг 34.13. Код контроллеров-функций `profile_bb_change()` и `profile_bb_delete()`

```
@login_required
def profile_bb_change(request, pk):
    bb = get_object_or_404(Bb, pk=pk)
    if request.method == 'POST':
        form = BbForm(request.POST, request.FILES, instance=bb)
        if form.is_valid():
            bb = form.save()
            formset = AIFormSet(request.POST, request.FILES, instance=bb)
            if formset.is_valid():
                formset.save()
                messages.add_message(request, messages.SUCCESS,
                                     'Объявление исправлено')
                return redirect('main:profile')
    else:
        form = BbForm(instance=bb)
        formset = AIFormSet(instance=bb)
    context = {'form': form, 'formset': formset}
    return render(request, 'main/profile_bb_change.html', context)
```

```
@login_required
def profile_bb_delete(request, pk):
    bb = get_object_or_404(Bb, pk=pk)
    if request.method == 'POST':
        bb.delete()
        messages.add_message(request, messages.SUCCESS,
                                      'Объявление удалено')
        return redirect('main:profile')
    else:
        context = {'bb': bb}
        return render(request, 'main/profile_bb_delete.html', context)
```

Для простоты на странице удаления объявления не будем выводить дополнительные иллюстрации — они там не особо нужны.

Объявим необходимые маршруты:

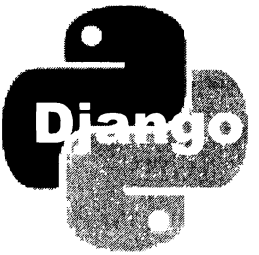
```
from .views import profile_bb_change, profile_bb_delete
. . .
urlpatterns = [
    . . .
    path('accounts/profile/change/<int:pk>/', profile_bb_change,
                                              name='profile_bb_change'),
    path('accounts/profile/delete/<int:pk>/', profile_bb_delete,
                                              name='profile_bb_delete'),
    path('accounts/profile/add/', profile_bb_add, name='profile_bb_add'),
    . . .
]
```

Шаблоны main\profile_bb_change.html и main\profile_bb_delete.html вы можете написать самостоятельно, использовав в качестве основы ранее написанные шаблоны.

В шаблон main\profile.html нужно добавить код, создающий гиперссылки для правки и удаления каждого из занесенных пользователем в базу объявлений:

```
<div class="media-body">
    <p>Рубрика: {{ bb.rubric }}</p>
    . . .
    <p class="text-right mt-2">
        <a href="{% url 'main:profile_bb_change' pk=bb.pk %}">
        Исправить</a>
        <a href="{% url 'main:profile_bb_delete' pk=bb.pk %}">Удалить</a>
    </p>
</div>
```

Напоследок следует проверить в действии реализованную функциональность по правке и удалению объявлений.



ГЛАВА 35

Комментарии

К каждому из опубликованных на нашем сайте объявлений посетители смогут оставить комментарии.

35.1. Подготовка к выводу CAPTCHA

Сделаем так, чтобы зарегистрированные пользователи могли оставлять комментарии беспрепятственно, а гости должны были дополнительно ввести CAPTCHA. Так мы хоть как-то обезопасим сайт от атаки служб рассылки спама.

Установим библиотеку Django Simple Captcha:

```
pip install django-simple-captcha
```

Добавим в список зарегистрированных в проекте приложение `captcha` — программное ядро этой библиотеки:

```
INSTALLED_APPS = [
    . . .
    'captcha',
]
```

И объявим в списке маршрутов уровня проекта (модуль urls.py пакета конфигурации) маршрут, указывающий на это приложение:

```
urlpatterns = [
    . . .
    path('captcha/', include('captcha.urls')),
    path('', include('main.urls')),
]
```

Миграции пока выполнять не будем — еще нужно объявить модель для хранения комментариев.

35.2. Модель комментария

Модель, хранящую комментарии, мы назовем `Comment`. Ее структура приведена в табл. 35.1.

Таблица 35.1. Структура модели `Comment`

Имя	Тип	Дополнительные параметры	Описание
bb	ForeignKey	Каскадное удаление разрешено	Объявление, к которому оставлен комментарий
author	CharField	Длина — 30 символов	Имя автора
content	TextField		Содержание
is_active	BooleanField	Значение по умолчанию — True, индексированное	Признак, показывать ли комментарий в списке
created_at	DateTimeField	Подставляется текущее значение даты и времени, индексированное	Дата и время публикации комментария

Код, объявляющий модель `Comment`, приведен в листинге 35.1.

Листинг 35.1. Код модели `Comment`

```
class Comment(models.Model):
    bb = models.ForeignKey(Bb, on_delete=models.CASCADE,
                           verbose_name='Объявление')
    author = models.CharField(max_length=30, verbose_name='Автор')
    content = models.TextField(verbose_name='Содержание')
    is_active = models.BooleanField(default=True, db_index=True,
                                    verbose_name='Выводить на экран?')
    created_at = models.DateTimeField(auto_now_add=True, db_index=True,
                                      verbose_name='Опубликован')

    class Meta:
        verbose_name_plural = 'Комментарии'
        verbose_name = 'Комментарий'
        ordering = ['created_at']
```

Для комментариев указываем сортировку по увеличению временно́й отметки их добавления. В результате более старые комментарии будут располагаться в начале списка, а более новые — в его конце.

Создадим миграции, после чего выполним их (при этом также будут выполнены миграции из библиотеки Django Simple Captcha).

35.3. Вывод и добавление комментариев

Список комментариев и форма для добавления комментария будут выводиться на общедоступной странице сведений об объявлении. Впоследствии мы сделаем так, чтобы комментарии выводились и на административной странице того же рода.

Объявим связанные с моделью `Comment` формы `UserCommentForm` и `GuestCommentForm`, в которые будут заносить комментарии, соответственно, зарегистрированные пользователи, выполнившие вход, и гости. Код этих форм приведен в листинге 35.2.

Листинг 35.2. Код форм `UserCommentForm` и `GuestCommentForm`

```
from captcha.fields import CaptchaField

from .models import Comment

class UserCommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        exclude = ('is_active',)
        widgets = {'bb': forms.HiddenInput}

class GuestCommentForm(forms.ModelForm):
    captcha = CaptchaField(label='Введите текст с картинки',
              error_messages={'invalid': 'Неправильный текст'})

    class Meta:
        model = Comment
        exclude = ('is_active',)
        widgets = {'bb': forms.HiddenInput}
```

Поле `is_active` (признак, будет ли комментарий выводиться на странице) уберем из форм, поскольку оно требуется лишь администрации сайта. У поля `bb`, хранящего ключ объявления, с которым связан комментарий, укажем в качестве элемента управления скрытое поле.

Теперь необходимо существенно обновить код контроллера-функции `detail()`, написанного в *главе 34*, реализовав в нем вывод комментариев и добавление нового комментария. Код обновленного контроллера приведен в листинге 35.3.

Листинг 35.3. Код обновленного контроллера-функции `detail()`

```
from .models import Comment
from .forms import UserCommentForm, GuestCommentForm

def detail(request, rubric_pk, pk):
    bb = Bb.objects.get(pk=pk)
    ais = bb.additionalimage_set.all()
```

```
    comments = Comment.objects.filter(bb=pk, is_active=True)
    initial = {'bb': bb.pk}
    if request.user.is_authenticated:
        initial['author'] = request.user.username
        form_class = UserCommentForm
    else:
        form_class = GuestCommentForm
    form = form_class(initial=initial)
    if request.method == 'POST':
        c_form = form_class(request.POST)
        if c_form.is_valid():
            c_form.save()
            messages.add_message(request, messages.SUCCESS,
                                 'Комментарий добавлен')
        else:
            form = c_form
            messages.add_message(request, messages.WARNING,
                                 'Комментарий не добавлен')
    context = {'bb': bb, 'ais': ais, 'comments': comments, 'form': form}
    return render(request, 'main/detail.html', context)
```

В поле `bb` создаваемой формы ввода комментария заносим ключ выводящегося на странице объявления — с ним будет связан добавляемый комментарий. Если текущий пользователь выполнил вход на сайт, заносим его имя в поле `author` этой формы комментария. Наконец, если текущий пользователь выполнил вход на сайт, то создаем форму на основе класса `UserCommentForm`, а если не выполнил — на основе класса `GuestCommentUser`. Все эти действия выполняет фрагмент кода:

```
initial = {'bb': bb.pk}
if request.user.is_authenticated:
    initial['author'] = request.user.username
    form_class = UserCommentForm
else:
    form_class = GuestCommentForm
form = form_class(initial=initial)
```

Объект формы сохраним в переменной с именем `form`. Форма из этой переменной впоследствии будет выведена на странице сведений об объявлении.

Далее, если полученный запрос был отправлен HTTP-методом POST, т. е. посетитель ввел комментарий и отправил его на сохранение, создаем еще один объект формы, передав конструктору полученные данные. Второй объект формы сохраняется в переменной `c_form`. После этого выполняем валидацию второй формы.

Если валидация прошла успешно, сохраняем введенный комментарий и выводим соответствующее всплывающее сообщение. Когда страница будет выведена, она будет содержать только что добавленный комментарий и пустую форму для ввода комментария из переменной `form`.

Если же валидация прошла неуспешно, переносим форму из переменной `c_form` в переменную `form`. Эта форма, хранящая некорректные данные и сообщения об ошибках ввода, впоследствии будет выведена на экран, и посетитель сразу увидит, что он ввел не так. Также выводим всплывающее сообщение о неуспешном добавлении комментария.

В шаблоне main\detail.html отыщем тег шаблонизатора `block content . . . endblock` и вставим перед закрывающим тегом код, выводящий комментарии:

```
{% block content %}
. . .
<h4 class="mt-5">Новый комментарий</h4>
<form method="post">
    {% csrf_token %}
    {% bootstrap_form form layout='horizontal' %}
    {% buttons submit='Добавить' %}{% endbuttons %}
</form>
{% if comments %}
<div class="mt-5">
    {% for comment in comments %}
    <div class="my-2 p-2 border">
        <h5>{{ comment.author }}</h5>
        <p>{{ comment.content }}</p>
        <p class="text-right font-italic">{{ comment.created_at }}</p>
    </div>
    {% endfor %}
</div>
{% endif %}
{% endblock %}
```

Стилевой класс `my-2` задает небольшие внешние отступы сверху и снизу, а стилевой класс `p-2` — небольшие внутренние отступы со всех сторон. Мы используем их, чтобы создать просветы между отдельными комментариями.

Закончив программирование, попробуем открыть страницу со сведениями о каком-либо объявлении и добавить один или два комментария. После этого выполним вход на сайт и снова попытаемся добавить комментарий. Отметим, что во втором случае форма для добавления комментария не включает поле ввода CAPTCHA.

Осталось добавить список комментариев на административную страницу сведений об объявлении. Сделайте это самостоятельно. Форму для ввода комментария и соответствующую функциональность в контроллере можно не делать — вряд ли автор объявления будет комментировать его сам...

35.4. Отправка уведомлений о новых комментариях

Отправлять уведомления о появлении новых комментариев будем в обработчике сигнала `post_save`, возникающего после сохранения записи в модели `Comment`.

Откроем модуль utilities.py пакета приложения и добавим в него объявление функции `send_new_comment_notification()`, которая и выполнит отправку уведомления. Ее код похож на код аналогичной функции `send_activation_notification()` (см. листинг 32.18) и приведен в листинге 35.4.

Листинг 35.4. Код функции `send_new_comment_notification()`

```
def send_new_comment_notification(comment):
    if ALLOWED_HOSTS:
        host = 'http://' + ALLOWED_HOSTS[0]
    else:
        host = 'http://localhost:8000'
    author = comment.bb.author
    context = {'author': author, 'host': host, 'comment': comment}
    subject = render_to_string('email/new_comment_letter_subject.txt', context)
    body_text = render_to_string('email/new_comment_letter_body.txt', context)
    author.email_user(subject, body_text)
```

Шаблоны email\new_comment_letter_subject.txt и email\new_comment_letter_body.txt, которые сформируют тему и тело электронного письма, вы, уважаемые читатели, создайте самостоятельно. Их можно написать на основе аналогичных шаблонов email\activation_letter_subject.txt и email\activation_letter_body.txt (см. листинги 32.19 и 32.20). В письме нужно указать интернет-адрес административной страницы сведений об объявлении, к которому был оставлен новый комментарий.

Осталось лишь привязать к сигналу `post_save` обработчик, вызывающий функцию `send_new_comment_notification()` после добавления комментария. Код, выполняющий привязку, поместим в модуль models.py пакета приложения. Вот этот код:

```
from django.db.models.signals import post_save

from .utilities import send_new_comment_notification

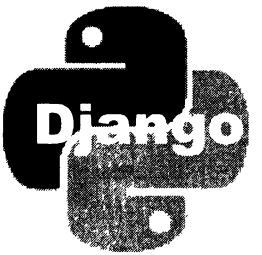
def post_save_dispatcher(sender, **kwargs):
    author = kwargs['instance'].bb.author
    if kwargs['created'] and author.send_messages:
        send_new_comment_notification(kwargs['instance'])

post_save.connect(post_save_dispatcher, sender=Comment)
```

Перед тем как отправлять оповещение, следует проверить, не запретил ли пользователь их отправку, т. е. не хранит ли поле `send_messages` модели пользователя `AdvUser` значение `False`.

Попробуем еще раз добавить комментарий к какому-либо объявлению и удостоверимся, что уведомление о новом комментарии было отправлено.

Осталось лишь объявить редактор для модели `Comment` и зарегистрировать его на административном сайте Django. Автор полагает, что читатели сделают это самостоятельно.



ГЛАВА 36

Веб-служба REST

В этой, заключительной, главе книги мы напишем веб-службу, работающую по принципам REST. Она будет выдавать список из 10 последних объявлений, сведения о выбранном объявлении, список комментариев к заданному объявлению и позволит добавить новый комментарий. Для простоты разрешим комментировать объявления только зарегистрированным пользователям.

36.1. Веб-служба

36.1.1. Подготовка к разработке веб-службы

Установим библиотеки Django REST framework и django-cors-headers, для чего наберем команды:

```
pip install djangorestframework
pip install django-cors-headers
```

Сразу же создадим новое приложение `api`, в котором и реализуем функциональность веб-службы:

```
manage.py startapp api
```

Добавим приложения `rest_framework` и `corsheaders` — программные ядра только что установленных библиотек, а также только что созданное приложение `api` в список зарегистрированных в проекте:

```
INSTALLED_APPS = [
    . . .
    'rest_framework',
    'corsheaders',
    'api.apps.ApiConfig',
]
```

Добавим в список зарегистрированных в проекте необходимый для работы посредник:

```
MIDDLEWARE = [
    . . .
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
    . . .
]
```

Не забудем указать там же, в модуле settings.py пакета конфигурации, настройки, разрешающие доступ к веб-службе с любого домена:

```
CORS_ORIGIN_ALLOW_ALL = True
CORS_URLS_REGEX = r'^/api/.*$'
```

36.1.2. Список объявлений

Создадим в пакете приложения `api` модуль serializers.py. В нем сохраним код сериализатора `BbSerializer`, формирующего список объявлений (листинг 36.1).

Листинг 36.1. Код сериализатора `BbSerializer`

```
from rest_framework import serializers

from main.models import Bb

class BbSerializer(serializers.ModelSerializer):
    class Meta:
        model = Bb
        fields = ('id', 'title', 'content', 'price', 'created_at')
```

В составе сведений о каждом объявлении ради компактности будем отправлять лишь ключ, название, описание, цену товара и временну́ю отметку создания объявления. Интернет-адрес основной иллюстрации и контакты отправим в составе сведений о выбранном объявлении.

Контроллер, который будет выдавать список объявлений, реализуем в виде функции и назовем `bbs()`. Его код приведен в листинге 36.2.

Листинг 36.2. Код контроллера-функции `bbs()`

```
from rest_framework.response import Response
from rest_framework.decorators import api_view

from main.models import Bb
from .serializers import BbSerializer

@api_view(['GET'])
def bbs(request):
    if request.method == 'GET':
        bbs = Bb.objects.filter(is_active=True)[:10]
```

```
        serializer = BbSerializer(bbs, many=True)
        return Response(serializer.data)
```

Откроем список маршрутов уровня проекта (модуль urls.py пакета конфигурации) и добавим маршрут, указывающий на приложение `api`:

```
urlpatterns = [
    . . .
    path('api/', include('api.urls')),
    path('', include('main.urls')),
]
```

В пакете приложения `api` создадим модель urls.py, в который запишем список маршрутов уровня этого приложения (листинг 36.3).

Листинг 36.3. Код модуля urls.py пакета приложения api

```
from django.urls import path

from .views import bbs

urlpatterns = [
    path('bbs/', bbs),
]
```

Пока что он содержит лишь маршрут, указывающий на только что написанный нами контроллер `bbs()`.

Сохраним код, запустим отладочный веб-сервер и попробуем получить список объявлений, перейдя по интернет-адресу **http://localhost:8000/api/bbs/**. Если мы все сделали правильно, то увидим веб-представление, показывающее последние 10 объявлений, которые были оставлены посетителями сайта.

36.1.3. Сведения о выбранном объявлении

В состав сведений о выбранном объявлении, помимо ключа записи, названия, описания, цены товара и даты создания объявления, следует включить контакты и интернет-адрес основной иллюстрации.

Учтем это при написании сериализатора `BbDetailSerializer`, выдающего сведения об объявлении (листинг 36.4). Занесем его код в модуль serializers.py пакета приложения.

Листинг 36.4. Код сериализатора BbDetailSerializer

```
class BbDetailSerializer(serializers.ModelSerializer):
    class Meta:
        model = Bb
        fields = ('id', 'title', 'content', 'price', 'created_at',
                  'contacts', 'image')
```

Контроллер, выдающий сведения о выбранном объявлении, назовем `BbDetailView` и реализуем в виде класса, производного от класса `RetrieveAPIView`. Его код, весьма компактный, приведен в листинге 36.5.

Листинг 36.5. Код контроллера-класса `BbDetailView`

```
from rest_framework.generics import RetrieveAPIView

from .serializers import BbDetailSerializer

class BbDetailView(RetrieveAPIView):
    queryset = Bb.objects.filter(is_active=True)
    serializer_class = BbDetailSerializer
```

Добавим в список маршрутов уровня приложения маршрут, который укажет на наш новый контроллер:

```
from .views import BbDetailView

urlpatterns = [
    path('bbs/<int:pk>/', BbDetailView.as_view()),
    path('bbs/', bbs),
]
```

Сохраним код и попытаемся получить сведения об объявлении с ключом 1, для чего выполним переход по интернет-адресу **http://localhost:8000/api/bbs/1/**. Далее запросим сведения о паре других объявлений.

36.1.4. Вывод и добавление комментариев

Код сериализатора `CommentSerializer`, который будет отправлять список комментариев и добавлять новый комментарий, приведен в листинге 36.6.

Листинг 36.6. Код сериализатора `CommentSerializer`

```
from main.models import Comment

class CommentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Comment
        fields = ('bb', 'author', 'content', 'created_at')
```

Он отправит клиенту ключ объявления, с которым связан комментарий, имя автора, содержимое и временну́ю отметку создания комментария.

Код контроллера-функции `comments()`, выдающего список комментариев и добавляющего новый комментарий, приведен в листинге 36.7.

Листинг 36.7. Код контроллера-функции comments()

```
from rest_framework.decorators import permission_classes
from rest_framework.status import HTTP_201_CREATED, HTTP_400_BAD_REQUEST
from rest_framework.permissions import IsAuthenticatedOrReadOnly

from main.models import Comment
from .serializers import CommentSerializer

@api_view(['GET', 'POST'])
@permission_classes((IsAuthenticatedOrReadOnly,))
def comments(request, pk):
    if request.method == 'POST':
        serializer = CommentSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=HTTP_201_CREATED)
        else:
            return Response(serializer.errors,
                            status=HTTP_400_BAD_REQUEST)
    else:
        comments = Comment.objects.filter(is_active=True, bb=pk)
        serializer = CommentSerializer(comments, many=True)
        return Response(serializer.data)
```

Ранее мы условились, что разрешим добавлять комментарии только зарегистрированным пользователям, а просматривать их, напротив, позволим всем. Поэтому мы пометили контроллер декоратором `permission_classes()`, в котором указали класс разграничения доступа `IsAuthenticatedOrReadOnly`.

Маршрут, который укажет на новый контроллер и который мы поместим в список уровня приложения, будет выглядеть так:

```
from .views import comments

urlpatterns = [
    path('bbs/<int:pk>/comments/', comments),
    path('bbs/<int:pk>/', BbDetailView.as_view()),
    . . .
]
```

Проверим, удастся ли получить список комментариев, оставленных под объявлением с ключом 1. Какой интернет-адрес при этом нужно набирать в веб-обозревателе, сообразим самостоятельно.

36.2. Тестовый фронтенд

Для тестирования нашей веб-службы создадим простой фронтенд, применив популярный клиентский веб-фреймворк Angular.

36.2.1. Введение в Angular

Angular — веб-фреймворк, предназначенный для написания клиентской части веб-сайтов, в том числе фронтендов. Сайт, написанный с применением Angular, представляет собой совокупность компонентов, служб, метамодулей, маршрутизатора и одной-единственной веб-страницы.

- *Компонент* — представляет пользовательский интерфейс фронтенда или его часть. Непосредственно взаимодействует с пользователем. Может включать в свой состав другие компоненты.

  Компоненты Angular независимы и изолированы друг от друга. Чтобы реализовать обмен данными между компонентами, следует явно предусмотреть в них соответствующие средства.

  Каждый компонент связан с определенным парным тегом (*тегом компонента*). Чтобы вывести компонент на экран, достаточно вставить этот тег в HTML-код.

  В нашем случае один компонент будет выводить список объявлений, другой — сведения о выбранном объявлении.

- *Компонент приложения* — выводится на экран сразу при открытии Angular-сайта (*приложением* в терминологии Angular называется сам сайт). Генерируется при создании нового проекта Angular.

  Как правило, компонент приложения выводит другие компоненты, непосредственно отображающие какие-либо данные.

- *Служба* — бизнес-логика фронтенда или ее часть. Может осуществлять математические вычисления (например, окончательный подсчет выводимых итогов), валидацию введенных данных, взаимодействие с бэкендом и пр. Используется компонентами для работы, но не связана с каким-либо конкретным компонентом.

- *Метамодуль* (в оригинальной документации — ngModule) — выполняет две задачи:

  - объединяет компоненты и службы, составляющие сайт или его раздел (подобно приложению Django);
  - инициализирует входящие в его состав компоненты, службы и прочие сущности, готовя их к работе.

  Проект Angular-сайта может включать произвольное количество метамодулей, как минимум, один.

  Сущности — компоненты и службы, — объявленные в одном метамодуле, не доступны в других. Чтобы сделать какую-либо сущность доступной в других

метамодулях, нужно явно выполнить ее *метаэкспорт*. Тогда другие метамодули, выполнив *метаимпорт* этого метамодуля, смогут их использовать (не путать с импортом обычных модулей на уровне языка TypeScript).

Сам Angular реализован в виде набора метамодулей, содержащих ключевые службы.

- *Метамодуль приложения* — стартует сразу при открытии Angular-сайта, запускает ключевые службы и выводит на экран компонент приложения. Генерируется при создании нового проекта Angular.
- Маршрутизатор — при переходе по интернет-адресу, совпадающему с определенным шаблонным путем, выводит на экран соответствующий компонент (как, собственно, и в Django). Реализован в виде службы в одном из метамодулей Angular.
- *Стартовая веб-страница* — единственная веб-страница, которая входит в состав Angular-сайта и на которой выводятся все входящие в его состав компоненты.

  При открытии стартовой страницы запускается привязанный к ней стартовый веб-сценарий, инициализирующий и запускающий метамодуль приложения, который, в свою очередь, выводит на экран компонент приложения.

Компоненты, службы и метамодули Angular реализуются в виде классов.

Программный код Angular-сайтов пишется на языке TypeScript — дальнейшем развитии JavaScript.

В состав любого проекта Angular входят компилятор, транслирующий TypeScript-код в обычный JavaScript, и отладочный веб-сервер.

> ***НА ЗАМЕТКУ***
>
> Полное описание фреймворка Angular находится на сайте **https://angular.io/**, а описание языка TypeScript — на сайте **https://www.typescriptlang.org/**.

36.2.2. Подготовка к разработке фронтенда

Сначала установим исполняющую среду Node.js. Ее дистрибутивные комплекты можно отыскать по интернет-адресу **https://nodejs.org/en/download/current/**.

Далее установим утилиту командной строки ng, с помощью которой выполняется создание Angular-проектов, программных модулей различных типов, запуск отладочного веб-сервера Angular и ряд других служебных задач. Установить ng можно, набрав в командной строке команду:

```
npm install -g @angular/cli
```

Теперь создадим проект нашего Angular-фронтенда, который назовем `bbclient`. Пользуясь командной строкой, перейдем в папку, в которой будет располагаться папка проекта, и подадим команду:

```
ng new bbclient --defaults --skip-git --skip-tests
```

Ключ `--defaults` указывает создать проект с настройками по умолчанию — без создания маршрутизации (мы сами ее сделаем) и с поддержкой таблиц стилей, написанных на языке CSS. Ключи `--skip-git` и `--skip-tests` указывают не создавать локальный репозиторий GIT и тестовые модули (они нам не нужны).

В результате будет создана папка проекта bbclient, содержащая с десяток служебных файлов и три папки. Откроем папку src, в которой хранятся файлы с исходным кодом сайта. Там находится файл index.html, содержащий HTML-код стартовой веб-страницы. Откроем этот файл и посмотрим на код, создающий секцию тела страницы (тег `<body>`):

```
<body>
  <app-root></app-root>
</body>
```

Парный тег `<app-root>` — это тег компонента приложения, носящего имя `AppComponent`. Встретив его, программное ядро Angular создаст объект этого компонента и выведет его в данном месте страницы.

Более в папке src ничего интересного нет. Перейдем во вложенную в нее папку app, в которой хранятся все основные программные модули, написанные на TypeScript. Именно с содержимым этой папки мы и будем иметь дело в дальнейшем.

Изначально проект Angular включает лишь компонент приложения и метамодуль приложения. Нам понадобятся еще два компонента и служба.

В командной строке перейдем в папку проекта и последовательно подадим три команды для создания:

- компонента списка объявлений `BbListComponent`:

  ```
  ng generate component bb-list --flat
  ```

  Флаг `--flat` задает размещение файлов с модулем непосредственно в папке src\app, а не во вложенной папке (проект у нас несложный, и разносить его код по разным вложенным папкам не имеет смысла);

- компонента сведений о выбранном объявлении `BbDetailComponent` (он же выведет список комментариев и форму для добавления нового комментария):

  ```
  ng generate component bb-detail --flat
  ```

- службы `BbService` — для "общения" с бэкендом, написанным в *разд. 36.1*:

  ```
  ng generate service bb
  ```

36.2.3. Метамодуль приложения *AppModule*. Маршрутизация в Angular

Метамодуль запускает в работу компоненты и службы, зарегистрированные в нем (подробности — в *разд. 36.2.1*). Метамодуль приложения стартует непосредственно при открытии сайта, запускает и выводит на экран компонент приложения.

Код метамодуля приложения хранится в файле app.module.ts (не забываем, что все ключевые TypeScript-модули находятся в папке src\app). Класс метамодуля приложения носит имя AppModule. Его изначальный код приведен в листинге 36.8.

Листинг 36.8. Код метамодуля приложения AppModule до изменений

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { BbListComponent } from './bb-list.component';
import { BbDetailComponent } from './bb-detail.component';

@NgModule({
  declarations: [
    AppComponent,
    BbListComponent,
    BbDetailComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Языковые конструкции import . . . from выполняют импорт сущностей из различных модулей (язык TypeScript является модульным, как и Python).

Класс метамодуля практически всегда пуст — не имеет ни свойств, ни методов. Все параметры метамодуля указываются в вызове декоратора NgModule из TypeScript-модуля @angular/core, в виде объекта класса Object со свойствами:

- declarations — массив классов компонентов, регистрируемых в метамодуле.

  Отметим, что там уже присутствуют, помимо компонента приложения AppComponent, еще и созданные вручную компоненты BbListComponent и BbDetailComponent. Их добавила туда утилита ng сразу при их создании;

- providers — массив классов регистрируемых служб (у нас — пуст);

- bootstrap — массив компонентов приложения. Практически всегда содержит лишь компонент AppComponent (как и в нашем случае);

- imports — массив метаимпортируемых метамодулей.

  У нас он включает лишь метамодуль BrowserModule, содержащий программное ядро фреймворка, ключевые службы, директивы и фильтры (речь о которых пойдет позже).

Нам нужно сделать следующее:

- зарегистрировать службу `Bb`, чтобы она заработала. Для этого добавим в метамодуль следующий код:

```
. . .
import { BbService } from './bb.service';

@NgModule({
  . . .
  providers: [
    BbService
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

- выполнить метаимпорт метамодуля `FormsModule` из модуля `@angular/forms` и метамодуля `HttpClientModule` из модуля `@angular/common/http`. Первый обеспечивает работу веб-форм (включая двустороннее связывание данных, о котором поговорим позже), второй — взаимодействие с бэкендом. Добавим следующий код:

```
. . .
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  . . .
  imports: [
    BrowserModule,
    HttpClientModule,
    FormsModule
  ],
  . . .
})
export class AppModule { }
```

- настроить проект на поддержку русского языка, добавив код:

```
. . .
import { registerLocaleData } from '@angular/common';
import localeRu from '@angular/common/locales/ru';
import localeRuExtra from '@angular/common/locales/extra/ru';
import { LOCALE_ID } from '@angular/core';

registerLocaleData(localeRu, 'ru', localeRuExtra);

@NgModule({
  providers: [
    BbService,
```

```
    {provide: LOCALE_ID, useValue: 'ru'}
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Служба `LOCALE_ID` из модуля `@angular/core` хранит обозначение текущего языка, по умолчанию — `'en-US'` (американский английский). Указав в массиве `providers` объект класса `Object`, свойство `provide` которого хранит ссылку на эту службу, а свойство `useValue` — строку `'ru'`, мы изменим текущий язык на русский.

Функция `registerLocaleData()` из модуля `@angular/common` непосредственно задает модули, откуда Angular будет брать настройки, характерные для выбранного языка. В первом параметре мы указали модуль `localeRu` из пакета `@angular/common/locales/ru`, содержащий основные настройки, во втором — строковое обозначение нужного языка `'ru'`, в третьем — модуль с дополнительными настройками `localeRuExtra`, объявленный в пакете `@angular/common/locales/extra/ru`;

- написать список маршрутов, связав:
  - шаблонный путь */<ключ объявления>/* — с компонентом `BbDetailComponent`, который выведет сведения об объявлении с заданным *ключом*;
  - "корень" сайта — компонентом списка объявлений `BbListComponent`.

Добавим в метамодуль такой код:

```
. . .
import { Routes } from '@angular/router';
. . .
const appRoutes: Routes = [
  {path: ':pk', component: BbDetailComponent},
  {path: '', component: BbListComponent}
];

@NgModule({
  . . .
})
export class AppModule { }
```

Список маршрутов в виде массива мы сохранили в константе `appRoutes`. Каждый маршрут описывается объектом класса `Object` со свойствами `path` (шаблонный путь в виде строки) и `component` (связанный с ним компонент). В шаблонном пути первого маршрута ключ объявления будет передаваться в URL-параметре с именем `pk`.

У константы мы указали тип `Routes` из модуля `@angular/router`, записав его после двоеточия. TypeScript поддерживает статическую типизацию, как в языках C++, C#, Delphi и др. Тип `Routes` описывает значение как массив объектов класса

Object, содержащих свойства path, component и некоторые другие, необязательные, не используемые здесь;

- инициализировать маршрутизатор, вставив код:

```
. . .
import { RouterModule } from '@angular/router';
. . .
@NgModule({
  . . .
  imports: [
    RouterModule.forRoot(appRoutes),
    BrowserModule,
    . . .
  ],
  . . .
})
export class AppModule { }
```

У метамодуля RouterModule из модуля @angular/router мы вызвали статический метод forRoot(), передав ему созданный ранее список маршрутов. Метод вернет объект программно сгенерированного метамодуля с готовым к работе маршрутизатором, который мы метаимпортировали в метамодуль приложения.

36.2.4. Компонент приложения *AppComponent*

Компонент — это часть интерфейса сайта, написанного на Angular. Компонент приложения выводится на стартовой странице сразу после открытия сайта. Как правило, он включает другие компоненты, выводящие различные данные.

Код компонента приложения хранится в модуле app.component.ts. Класс этого компонента называется AppComponent. Его изначальный код приведен в листинге 36.9.

Листинг 36.9. Код компонента приложения AppComponent до изменений

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'bbclient';
}
```

Класс компонента может содержать свойства, значения которых будут выводиться на экран, и методы, которые вычисляют выводимые на экран значения или выполняют какие-либо иные действия. Изначально класс компонента AppComponent содержит лишь свойство title, хранящее имя проекта.

Параметры компонента указываются в вызове декоратора `Component`, объявленного в модуле `@angular/core`, в виде объекта класса `Object` со свойствами:

- `selector` — тег компонента;
- `templateUrl` — путь к файлу с шаблоном компонента;
- `styleUrls` — массив путей к файлам таблиц стилей, определяющих представление компонента. Изначально там присутствует одна таблица стилей — "пустая", т. е. не содержащая никакого кода.

Удалим из компонента свойство `title`, нам совершенно не нужное:

```
export class AppComponent {
  title = 'bbclient';
}
```

Шаблон компонента, хранящийся в файле app.component.html в той же папке src\app, довольно велик, поскольку формирует сложную страницу приветствия.

Нам нужно, чтобы компонент приложения выводил заголовок "Доска объявлений", под которым будет находиться *выпуск* (outlet) — место на странице, куда маршрутизатор Angular станет выводить тот или иной компонент в процессе навигации по сайту. Откроем файл его шаблона и переделаем согласно листингу 36.10.

Листинг 36.10. Код шаблона компонента `AppComponent`

```
<header>
    <h1>Доска объявлений</h1>
</header>
<router-outlet></router-outlet>
```

Выпуск обозначается парным тегом `<router-outlet>`.

36.2.5. Служба *BbService*. Внедрение зависимостей. Объекты-обещания

Служба Angular реализует бизнес-логику, не привязанную к какому-либо конкретному компоненту: подсчет итогов, валидацию введенных данных, взаимодействие с бэкендом и др. Последним как раз и будет заниматься служба `BbService`, которую мы сейчас напишем.

Код, объявляющий класс службы `BbService`, хранится в файле bb.service.ts. Его изначальный вид приведен в листинге 36.11.

Листинг 36.11. Код службы `BbService` до изменений

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
```

```
export class BbService {

  constructor() { }
}
```

Класс службы содержит лишь "пустой" конструктор. Декоратор `Injectable`, объявленный в модуле `@angular/core`, собственно, помечает этот класс как службу Angular. Еще он указывает у нее параметр `providedIn` со значением `'root'` — это значит, что служба будет обрабатываться основным обработчиком фреймворка (такой режим работы подходит в большинстве случаев).

Над классом службы мы проделаем следующие действия:

- объявим частное строковое свойство `url`, хранящее интернет-адрес хоста, на котором работает бэкенд:

  ```
  export class BbService {
    private url: String = 'http://localhost:8000';
    . . .
  }
  ```

  Ключевое слово `private` делает свойство частным — доступным только внутри объектов текущего класса;

- создадим частное свойство `http`, хранящее объект службы `HttpService` (объявлена в модуле `@angular/common/http`), посредством которой будет выполняться взаимодействие с бэкендом. Сделать это проще всего, исправив код конструктора следующим образом:

  ```
  . . .
  import { HttpClient } from '@angular/common/http';
  . . .
  export class BbService {
    . . .
    constructor(private http: HttpClient) { }
  }
  ```

  Встретив в конструкторе класса такое объявление параметра, Angular проверит, был ли ранее создан объект класса `HttpService`, и, если не был, создаст его. После этого фреймворк создаст в текущем объекте частное свойство `http` и присвоит ему данный объект класса.

  Отметим, что если служба `HttpService` требуется для работы сразу нескольким другим службам, то все они будут использовать один-единственный ее объект. А как только необходимость в службе отпадет, ее объект будет удален.

  Так работает система *внедрения зависимостей* (dependency injection), встроенная в Angular. Она обрабатывает единым образом все службы — и встроенные во фреймворк, и написанные разработчиком сайта. Необходимо лишь пометить класс службы декоратором `Injectable` и либо зарегистрировать службу в метамодуле, либо выполнить метаимпорт метамодуля, в котором она зарегистрирована;

- объявим метод `getBbs()`, получающий и выдающий перечень объявлений:

```
. . .
import { Observable } from 'rxjs';
. . .
export class BbService {
  . . .
  getBbs(): Observable<Object[]> {
    return this.http.get<Object[]>(this.url + '/api/bbs/');
  }
}
```

Метод `get(<интернет-адрес>)` класса `HttpService` отправляет запрос HTTP-методом GET по заданному *интернет-адресу* и возвращает загруженные данные в качестве результата. Метод имеет две важные особенности.

Во-первых, результат он возвращает в виде объекта класса `Observable` из модуля `rxjs`. Этот класс представляет значение, которое будет получено не прямо сейчас, а позже, спустя неопределенный промежуток времени. Назовем этот объект *обещанием*.

Во-вторых, в вызове метода `get()` обязательно следует указать тип возвращаемого значения, представляемого объектом-обещанием. Тип указывается после имени метода в угловых скобках. Мы указали тип `<Object[]>` — массив объектов класса `Object` (именно в таком виде бэкенд и отправит перечень объявлений).

В объявлении метода `getBbs()`, после имени самого метода и двоеточия, мы указали тип возвращаемого значения: `Observable<Object[]>` — массив объектов класса `Object`, представляемый обещанием — объектом класса `Observable`;

- объявим метод `getBb(<ключ объявления>)`, получающий и выдающий объявление с заданным *ключом*:

```
export class BbService {
  . . .
  getBb(pk: Number): Observable<Object> {
    return this.http.get<Object>(this.url + '/api/bbs/' + pk);
  }
}
```

Результатом этого метода станет единичный объект класса `Object`, представляемый обещанием;

- объявим служебный метод `handleError()`, который понадобится для обработки ошибок, которые могут возникнуть при добавлении новых комментариев:

```
. . .
import { of } from 'rxjs';
. . .
export class BbService {
  . . .
```

```
  handleError() {
    return (error: any): Observable<Object> => {
        window.alert(error.message);
        return of(null);
    }
  }
}
```

Метод, обрабатывающий ошибки, должен возвращать в качестве результата функцию, которая принимает в качестве параметра объект ошибки, возвращает объект-обещание с каким-либо значением, которое будет использоваться в дальнейших вычислениях, и, собственно, каким-то образом обрабатывает ошибку.

У возвращаемой методом `handleError()` функции мы указали параметр `error`, принимающий значения любого — `any` — типа, и возвращаемый результат в виде объекта класса `Object`, заключенного в обещание. Сама функция выводит окно с текстовым описанием возникшей ошибки и возвращает обещание с "нулевой" ссылкой `null`. Заключить значение в обещание можно с помощью функции `of()` из модуля `rxjs`;

- объявим метод `addComment()`, вызываемый в формате:

```
addComment(<ключ объявления>, <имя пользователя>, <пароль>,
                              <содержание комментария>)
```

и добавляющий новый комментарий:

```
. . .
import { catchError } from 'rxjs/operators';
import { HttpHeaders } from '@angular/common/http';
. . .
export class BbService {
  . . .
  addComment(bb: String, author: String, password: String,
    content: String): Observable<Object> {
    const comment = {'bb': bb, 'author': author, 'content': content};
    const options = {headers: new HttpHeaders(
      {'Content-Type': 'application/json',
      'Authorization': 'Basic ' + window.btoa(author + ':' +
      password)})};
    return this.http.post<Object>(this.url + '/api/bbs/' + bb +
      '/comments/', comment,
      options).pipe(catchError(this.handleError()));
  }
}
```

В константе `comment` сохраняем объект класса `Object`, содержащий все необходимые сведения для создания нового комментария. В константе `options` сохраняем объект того же класса, содержащий параметры отправляемого запроса. У нас этот объект содержит лишь свойство `headers`, которое хранит заголовки

запроса, формируемые объектом класса `HttpHeaders` из модуля `@angular/common/http`. Далее вызываем метод `post(<интернет-адрес>, <отправляемые данные> [, <параметры запроса>])` класса `HttpService`, который отправит запрос HTTP-методом POST.

Метод `post()` также возвращает обещание, заключающее в себе полученные от бэкенда данные. У этого обещания в вызове метода `pipe(<операция>)` посредством функции `catchError()` из модуля `rxjs/operators` задаем объявленный ранее метод `handleError()` в качестве обработчика ошибок;

- объявим метод `getComments(<ключ объявления>)`, загружающий список комментариев к объявлению с заданным *ключом*:

```
. . .
export class BbService {
  . . .
  getComments(pk: Number): Observable<Object[]> {
    return this.http.get<Object[]>(this.url + '/api/bbs/' + pk +
      '/comments/');
  }
}
```

36.2.6. Компонент списка объявлений *BbListComponent*. Директивы. Фильтры. Связывание данных

Класс компонента списка объявлений `BbListComponent` хранится в файле bb-list.component.ts. Откроем его и посмотрим на код, непосредственно объявляющий класс (остальной код аналогичен таковому у компонента `AppComponent`):

```
export class BbListComponent implements OnInit {
  constructor() { }

  ngOnInit() {
  }
}
```

Этот класс реализует интерфейс `OnInit` из модуля `@angular/core`. *Интерфейсом* в TypeScript называется описание набора методов — их имена, принимаемые параметры и возвращаемый результат (если таковые есть), — которые обязательно должны быть объявлены в классе, реализующем этот интерфейс.

Интерфейс `OnInit` содержит описание метода `ngOnInit()`. Следовательно, класс `BbListComponent`, реализующий этот интерфейс, должен содержать метод `ngOnInit()` (кстати, в коде класса уже имеется сгенерированная утилитой ng "заготовка" для его написания).

Программное ядро Angular, инициализируя очередной компонент, проверяет, реализует ли он интерфейс `OnInit`, и, если это так, вызывает у компонента метод `ngOnInit()` сразу после вызова конструктора. Этот метод — наилучшее место для выполнения кода, инициализирующего компонент.

Реализуем в компоненте следующее:

- объявим частное свойство `bbs` типа "массив объектов класса `Object`" для хранения списка объявлений:

```
export class BbListComponent implements OnInit {
  private bbs: Object[];
  . . .
}
```

- укажем в конструкторе, что нам понадобится частное свойство `bbservice` с объектом службы `BbService`, написанной в *разд. 36.2.5*:

```
. . .
import { BbService } from './bb.service';
. . .
export class BbListComponent implements OnInit {
  . . .
  constructor(private bbservice: BbService) { }
  . . .
}
```

Внедрение зависимостей работает также и в случае служб, написанных разработчиками сайтов;

- напишем в методе `ngOnInit()` код, загружающий перечень объявлений:

```
export class BbListComponent implements OnInit {
  . . .
  ngOnInit() {
    this.bbservice.getBbs().subscribe(
      (bbs: Object[]) => {this.bbs = bbs;}
    );
  }
}
```

Метод `getBbs()` службы `BbService` возвращает объект-обещание с массивом объявлений. Чтобы извлечь этот массив из обещания, воспользуемся методом `subscribe(<функция>)` класса `Observable`. Он задает функцию, которая выполнится после того, как значение, заключенное в обещании, будет реально получено, и примет это значение в единственном параметре. У нас эта функция присвоит полученный массив объявлений свойству `bbs` компонента, объявленному ранее.

Займемся шаблоном компонента `BbListComponent`. Откроем хранящий его файл bb-list.component.html и удалим весь имеющийся там код. Создадим в шаблоне вот что:

- заголовок второго уровня "Последние 10 объявлений" и блок, формирующий одно объявление:

```
<h2>Последние 10 объявлений</h2>
<div>
</div>
```

- сделаем так, чтобы блок повторялся столько раз, сколько объявлений присутствует в полученном от бэкенда массиве:

```
. . .
<div *ngFor="let bb of bbs">
</div>
```

Директива `*ngFor` Angular по назначению аналогична тегу `for . . . endfor` шаблонизатора Django. Заданное у нее значение `let bb of bbs` указывает ей повторить HTML-тег, в котором она присутствует, столько раз, сколько элементов имеется в массиве `bbs`, и на каждой итерации занести очередной элемент массива в переменную `bb`, доступную внутри содержащего ее тега;

- выведем в блоке заголовок третьего уровня с названием товара и абзац с его описанием:

```
. . .
<div *ngFor="let bb of bbs">
    <h3>{{bb.title}}</h3>
    <p>{{bb.content}}</p>
</div>
```

Директива `{{`*значение*`}}` Angular, подобно аналогичной директиве Django, выводит заданное *значение* в том месте шаблона, в котором присутствует. В качестве значения может быть указано свойство компонента, константа или простейшее выражение TypeScript.

Если значение свойства компонента, выводимого этой директивой, было программно изменено, то оно будет выведено повторно. Говорят, что директива `{{`*значение*`}}` создает связь между помеченным ей местом в шаблоне и свойством компонента в направлении "свойство → место в шаблоне" (*одностороннее связывание данных*);

- выведем абзац с ценой товара в рублях:

```
. . .
<div *ngFor="let bb of bbs">
    . . .
    <p class="price">{{bb.price|currency:'RUR'}}</p>
</div>
```

Фильтр `currency` выводит число в виде денежной суммы в формате, обозначение которого указано в параметре (у нас: `'RUR'` — российские рубли);

- выведем абзац с временно́й отметкой публикации объявления:

```
. . .
<div *ngFor="let bb of bbs">
    . . .
    <p class="date">{{bb.created_at|date:'medium'}}</p>
</div>
```

Фильтр `date` со значением `'medium'` выводит временну́ю отметку в формате:

```
<число> <сокращенное название месяца> <год> г., ↵
<часы>:<минуты>:<секунды>
```

- превратим заголовок третьего уровня, в котором выводится название товара, в гиперссылку на компонент со сведениями об объявлении:

```
. . .
<div *ngFor="let bb of bbs">
    <h3><a [routerLink]="[bb.id]">{{bb.title}}</a></h3>
    . . .
</div>
```

Директива `[routerLink]` вставляет в тег `<a>` атрибут `href` с интернет-адресом, составленным из элементов массива, который задан в качестве ее значения. У нас этот массив содержит всего один элемент — ключ объявления, извлекаемый из свойства `id` объекта, хранящегося в свойстве `bb` компонента, поэтому интернет-адреса будут иметь формат */<ключ объявления>/*.

> ***На заметку***
>
> Директивы и фильтры Angular также объявляются в виде классов, помеченных особыми декораторами. Разработчик сайта может создать свои директивы и фильтры, зарегистрировать их в метамодуле и использовать наряду со встроенными.

Осталось оформить наш компонент. Откроем файл bb-list.component.css, где хранится его таблица стилей, и запишем туда код из листинга 36.12.

Листинг 36.12. Код таблицы стилей компонента `BbListComponent`

```
div {
    margin: 10px 0px;
    padding: 0px 10px;
    border: grey thin solid;
}
div p.price {
    font-size: larger;
    font-weight: bold;
    text-align: right;
}
div p.date {
    font-style: italic;
    text-align: right;
}
```

36.2.7. Компонент сведений об объявлении *BbDetailComponent*. Двустороннее связывание данных

Модуль с объявлением класса компонента `BbDetailComponent` хранится в файле bb-detail.component.ts. Откроем его и сделаем следующее:

- объявим в классе частные свойства bb (объект со сведениями об объявлении, для простоты укажем у него любой — any — тип) и comments (массив объектов, содержащий перечень комментариев):

```
. . .
export class BbDetailComponent implements OnInit {
  private bb: any;
  private comments: Object[];
  . . .
}
```

- объявим в классе частные строковые свойства author (имя пользователя, который станет автором добавляемого комментария), password (пароль) и content (содержание добавляемого комментария):

```
. . .
export class BbDetailComponent implements OnInit {
  . . .
  private author: String = '';
  private password: String = '';
  private content: String = '';
  . . .
}
```

- укажем в конструкторе создать частные свойства bbservice с объектом службы BbService и ar — с объектом службы ActivatedRoute из модуля @angular/router:

```
. . .
import { ActivatedRoute } from '@angular/router';

import { BbService } from './bb.service';
. . .
export class BbDetailComponent implements OnInit {
  . . .
  constructor(private bbservice: BbService,
    private ar: ActivatedRoute) { }
  . . .
}
```

Служба ActivatedRoute позволяет получить сведения об интернет-адресе, по которому был выполнен переход, включая значения присутствующих в нем URL-параметров;

- объявим метод getComments(), загружающий перечень комментариев:

```
. . .
export class BbDetailComponent implements OnInit {
  . . .
  getComments() {
    this.bbservice.getComments(this.bb.id).subscribe(
        (comments: Object[]) => {this.comments = comments;}
    );
  }
}
```

Ключ объявления, комментарии к которому нужно загрузить, извлекаем из свойства `id` объекта, хранящегося в свойстве `bb` компонента, объявленном ранее;

- напишем в методе `ngOnInit()` код, загружающий с бэкенда объявление с полученным в URL-параметре `pk` ключом и список комментариев к нему:

```
. . .
export class BbDetailComponent implements OnInit {
  . . .
  ngOnInit() {
    const pk = this.ar.snapshot.params.pk;
    this.bbservice.getBb(pk).subscribe((bb: Object) => {
        this.bb = bb;
        this.getComments();
    });
  }
  . . .
}
```

Объект службы `ActivatedRoute` содержит свойство `snapshot`, хранящее объект со сведениями об интернет-адресе, по которому был выполнен переход. Этот объект, в свою очередь, поддерживает свойство `params` с объектом, хранящим все URL-параметры.

После получения сведений об объявлении вызываем ранее объявленный метод `getComments()`, чтобы загрузить комментарии к объявлению;

- объявим метод `submitComment()`, вызываемый в формате:

```
submitComment(<ключ объявления>, <имя пользователя>, <пароль>,
                                 <содержание комментария>)
```

и добавляющий новый комментарий:

```
. . .
export class BbDetailComponent implements OnInit {
  . . .
  submitComment() {
    this.bbservice.addComment(this.bb.id, this.author, this.password,
      this.content).subscribe((comment: Object) => {
        if (comment) {
            this.content = '';
            this.getComments();
        }
      });
  }
}
```

После успешного добавления комментария очищаем свойство `content`, хранящее его содержание, и перезагружаем перечень комментариев.

Шаблон этого компонента хранится в файле bb-detail.component.html. Откроем файл и занесем в него код из листинга 36.13.

Листинг 36.13. Код шаблона компонента BbDetailComponent до изменений

```
<div class="image"><img src="{{bb.image}}"></div>
<div class="others">
    <h2>{{bb.title}}</h2>
    <p>{{bb.content}}</p>
    <p class="price">{{bb.price|currency:'RUR'}}</p>
    <p>{{bb.contacts}}</p>
    <p class="date">{{bb.created_at|date:'medium'}}</p>
</div>
<h3>Новый комментарий</h3>
<form>
    <p>Имя: <input name="author" required></p>
    <p>Пароль: <input type="password" name="password" required></p>
    <p>Содержание:<br><textarea name="content" required></textarea></p>
    <p><input type="submit" value="Добавить"></p>
</form
<div class="comment" *ngFor="let comment of comments">
    <h4>{{comment.author}}</h4>
    <p>{{comment.content}}</p>
    <p class="date">{{comment.created_at|date:'medium'}}</p>
</div>
```

Шаблон можно разделить на три части. В верхней, находящейся выше заголовка третьего уровня "Новый комментарий", выводятся сведения об объявлении. Средняя включает сам этот заголовок и веб-форму, куда заносятся имя пользователя, пароль и содержание добавляемого комментария. В нижней части выводятся уже добавленные комментарии. Все это реализуется уже знакомыми нам приемами.

На данный момент компонент лишь выводит сведения об объявлении и список комментариев. Добавление комментария пока не работает. Кроме того, при выводе компонента в консоли веб-обозревателя появляется сообщение об ошибке доступа к свойству `image` значения `undefined`, хранящегося в свойстве `bb` компонента. Это происходит потому, что компонент выводится на экран раньше, чем успевает загрузить с бэкенда объявление и записать содержащий его объект в свойство `bb`.

Исправим все это следующим образом:

- укажем, чтобы верхняя часть шаблона выводилась на экран только после загрузки объявления и сохранения его в свойстве `bb`:

```
<ng-container *ngIf="bb">
    <div class="image"><img src="{{bb.image}}"></div>
    <div class="others">
        . . .
    </div>
</ng-container>
<h3>Новый комментарий</h3>
. . .
```

Парный тег `<ng-container>` создает *псевдоконтейнер* Angular, объединяющий произвольное количество элементов страницы, но не преобразующийся при выводе на экран в какой-либо HTML-тег. Директива `*ngIf` выводит на экран элемент, в котором присутствует, только в том случае, если заданное для нее значение равно `true`. В результате верхняя часть шаблона будет выведена только после того, как свойство `bb` получит значение, отличное от `undefined`;

- свяжем поля ввода **Имя**, **Пароль** и область редактирования **Содержание** формы со свойствами `author`, `password` и `content` соответственно:

```
. . .
<form>
    <p>Имя: <input [(ngModel)]="author" name="author" required></p>
    <p>Пароль: <input type="password" [(ngModel)]="password"
    name="password" required></p>
    <p>Содержание:<br><textarea [(ngModel)]="content" name="content"
    required></textarea></p>
    <p><input type="submit" value="Добавить"></p>
</form>
. . .
```

Директива `[(ngModel)]` связывает элемент управления, в теге которого присутствует, со свойством компонента, имя которого задано в качестве ее значения. При изменении значения в элементе управления оно сразу же копируется в свойство, а при программном изменении значения свойства — копируется в элемент управления (*двустороннее связывание данных*, которое можно обозначить как "свойство ←→ элемент управления");

- сделаем так, чтобы при нажатии кнопки **Добавить** формы введенный в нее комментарий отправлялся бэкенду для добавления в базу данных:

```
. . .
<form (ngSubmit)="submitComment()">
    . . .
</form>
. . .
```

Директива `(ngSubmit)` задает для элемента, в котором присутствует, обработчик события `submit`. У нас — это вызов метода `submitComment()` компонента.

Осталось написать таблицу стилей компонента `BbDetailComponent`. Она хранится в файле bb-detail.component.css и изначально "пуста". Сделайте это самостоятельно, оформив компонент по своему вкусу.

Как упоминалось ранее, Angular-проект включает в свой состав отладочный веб-сервер. Запустить его можно, перейдя в папку проекта и задав в командной строке команду:

```
ng serve
```

Поскольку тестовый Angular-сайт в процессе работы "общается" с веб-службой, написанной на Django и Django REST framework, также необходимо запустить отладочный веб-сервер Django.

Отладочный веб-сервер Angular работает через TCP-порт 4200. Следовательно, для перехода на наш сайт нужно набрать интернет-адрес **http://localhost:4200/**.

Проверим наш небольшой фронтенд в действии. После чего завершим работу обоих отладочных серверов. Отладочный сервер Angular останавливается нажатием комбинации клавиш <Ctrl>+<Break>, как и сервер Django.

Заключение

Вот и закончена книга о программировании веб-сайтов с помощью великолепного веб-фреймворка Django. Мы изучили практически все, что нужно для создания сайтов, и даже сделали в качестве практического занятия сайт доски объявлений. И теперь можем с уверенностью и гордостью именовать себя настоящими программистами!

Автор описал в книге все основные возможности Django, без которых не обойтись. Однако нельзя объять необъятное, и кое-что все-таки осталось "за кадром". В частности, не были описаны:

- подсистема комментирования;
- подсистема GeoDjango, предназначенная для разработки геоинформационных систем;
- подсистема для разработки средствами фреймворка обычных, статических веб-сайтов;
- средства для создания карты сайта;
- средства для формирования лент новостей в формате RSS и Atom;
- инструменты для экспорта данных в форматах CSV и Adobe PDF;
- средства для написания своих миграций;
- всевозможные вспомогательные инструменты;
- множество полезных дополнительных библиотек;
- разработка дополнительных библиотек для Django.

Обо всем этом рассказано в официальной документации, представленной на домашнем сайте фреймворка. И любой желающий ознакомиться с этими инструментами всегда может обратиться к ней. Что касается дополнительных библиотек, то их в огромном количестве можно найти в PyPI — стандартном репозитории Python.

О да, на изучение всех возможностей Django стоит потратить время. Фреймворк Django, как и язык Python, на котором он написан, с честью выдержал проверку

временем и занял свое место под солнцем. Он имеет огромную установочную базу — написанных с его применением сайтов — и внушительную армию поклонников. К которым, автор смеет надеяться, присоединитесь и вы, уважаемые читатели.

И — автор полностью уверен в этом — Django еще долгое время будет применяться в веб-строительстве, и если и уйдет когда-нибудь, так сказать, в отставку, то лишь после появления достойного конкурента. Пока их не предвидится...

Так что за будущее Django переживать не стоит — наш любимый фреймворк будет применяться еще очень и очень долго. И изучать его имеет смысл, как читая эту книгу, так и обращаясь к тематическим интернет-ресурсам. В табл. 3.1 приведен список таких ресурсов.

Таблица 3.1. *Интернет-ресурсы, посвященные Django*

Интернет-адрес	Описание
https://www.djangoproject.com/	Официальный сайт фреймворка Django. Дистрибутивы, документация, поддержка
https://djangopackages.org/	Подборка дополнительных библиотек и утилит для Django
https://www.djbook.ru/	Русскоязычная документация по Django
https://vk.com/django_framework	Группа "ВКонтакте", посвященная Django
https://www.python.org/	Официальный сайт языка Python. Дистрибутивы, документация, поддержка
https://pypi.org/	Официальный репозиторий Python, содержащий огромное количество дополнительных библиотек
https://vk.com/itcookies/	Группа "ВКонтакте", посвященная программированию, в том числе и на Python
https://pythonworld.ru/	Русскоязычный сайт для Python-программистов

Интернет-адреса официальных сайтов использованных в книге сторонних библиотек приведены в тексте книги, в разделах, посвященных этим библиотекам.

Исходные коды разработанного в *части IV* книги веб-сайта электронной доски объявлений находятся в сопровождающем книгу электронном архиве, который можно скачать с FTP-сервера издательства "БХВ-Петербург" по ссылке **ftp://ftp.bhv.ru/9785977566919.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение*).

На этом все. Автор книги прощается и желает вам успехов в веб-программировании.

Владимир Дронов

ПРИЛОЖЕНИЕ

Описание электронного архива

Электронный архив к книге выложен на FTP-сервер издательства «БХВ-Петербург» по интернет-адресу: **ftp://ftp.bhv.ru/9785977566919.zip**. Ссылка на него доступна и со страницы книги на сайте **www.bhv.ru**.

Содержимое архива описано в табл. П.1.

Таблица П.1. *Содержимое электронного архива*

Каталог, файл	**Описание**
bboard	Папка с исходным кодом веб-сайта электронной доски объявлений, разрабатываемого на протяжении *части IV* книги на Python и Django
bbclient	Папка с исходным кодом тестового фронтенда, используемого для отладки веб-службы и написанного с применением веб-фреймворка Angular
readme.txt	Файл с описанием архива и инструкциями по развертыванию обоих веб-сайтов

Предметный указатель

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

А

Б

В

Г

Д

Ж

З

И

К

М

Н

О

П

Р

С

Т

У

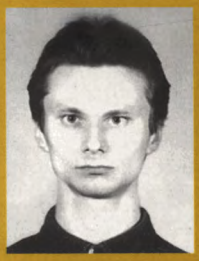
Ф

Ш

Django 3.0
Практика создания веб-сайтов на Python

Python и Django — веб-разработка на высоком уровне

Книга посвящена разработке веб-сайтов на языке Python с применением веб-фреймворка Django 3.0. Представлены новинки Django 3.0 и дано наиболее полное описание его инструментов: моделей, контроллеров, шаблонов, средств обработки пользовательского ввода, включая выгруженные файлы, разграничения доступа, посредников, сигналов, инструментов для отправки электронной почты, кэширования и пр. Рассмотрены дополнительные библиотеки, выполняющие обработку BBCode-тегов, CAPTCHA, вывод графических миниатюр, аутентификацию через социальные сети (в частности, «ВКонтакте»), интеграцию с Bootstrap. Рассказано о программировании веб-служб REST, настройке и использовании административного веб-сайта Django, публикации сайтов с помощью веб-сервера Uvicorn, работе с базами данных PostgreSQL, кэшировании сайтов с помощью Memcached и Redi. Подробно описано создание полнофункционального веб-сайта – электронной доски объявлений, веб-службы, работающей в его составе, и тестового фронтенда для него, написанного на Angular.



Дронов Владимир Александрович, профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года. Автор более 30 популярных компьютерных книг, в том числе «Laravel. Быстрая разработка современных динамических Web-сайтов на PHP, MySQL, HTML и CSS», «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Python 3. Самое необходимое», «Python 3 и PyQt 5. Разработка приложений» и серии книг-уроков «JavaScript. 20 уроков для начинающих», «HTML и CSS. 25 уроков для начинающих», «PHP и MySQL. 25 уроков для начинающих». Его статьи публикуются в журналах «Мир ПК» и «ИнтерФейс» (Израиль) и на интернет-порталах «IZ City» и «TheVista.ru».

Полный исходный код описанного в книге веб-сайта можно скачать по ссылке ftp://ftp.bhv.ru/9785977566919.zip, а также со страницы книги на сайте www.bhv.ru.

ISBN 978-5-9775-6691-9

9 785977 566919

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

bhv®