NextJS

Real-World Next.js Copyright © 2022 Packt Publishing

Next.js — це масштабована та високопродуктивна платформа React.js для сучасної веб-розробки. Він надає широкий набір функцій, таких як гібридний рендеринг, попередня вибірка маршруту, автоматична оптимізація зображення та інтернаціоналізація.

Next.js — це цікава технологія, яку можна використовувати для багатьох цілей. Якщо ви (або ваша компанія) хочете створити платформу електронної комерції, блог або простий веб-сайт, за допомогою цієї книги ви можете дізнатися, як це зробити без шкоди для продуктивності, взаємодії з користувачем або задоволення розробника. Починаючи з основ Next.js, ви зрозумієте, як фреймворк може допомогти вам досягти ваших цілей, і ви зрозумієте, наскільки універсальним є Next.js, створюючи реальні програми з покроковими поясненнями. Ви дізнаєтеся, як вибрати відповідну методологію візуалізації для свого веб-сайту, як захистити її та як розгорнути її для різних постачальників. Ми завжди зосереджуватимемося на продуктивності та задоволенні розробників.

До кінця цієї книги ви зможете проектувати, створювати та розгортати красиві та сучасні архітектури за допомогою Next.js з будь-якою CMS або джерелом даних.

Розділ 1

Короткий вступ до Next.js

Next.js — це веб-фреймворк JavaScript з відкритим вихідним кодом для React, який постачається з багатим набором функцій із коробки, таких як рендеринг на стороні сервера, статична генерація сайту та поступова статична регенерація. Це лише деякі з багатьох вбудованих компонентів і плагінів, які роблять Next.js платформою, готовою як для додатків корпоративного рівня, так і для невеликих веб-сайтів.

У цьому розділі ми розглянемо такі теми:
- Знайомство з фреймворком Next.js
- Порівняння Next.js з іншими популярними альтернативами
- Відмінності між Next.js і React на стороні клієнта
- Анатомія проекту Next.js за замовчуванням
- Як розробляти програми Next.js за допомогою TypeScript

• Як налаштувати конфігурації Babel і webpack

Технічні вимоги

Щоб розпочати роботу з Next.js, вам потрібно встановити пару залежностей на вашій машині.

Перш за все, вам потрібно встановити **Node.js** і **npm**. Будь ласка, перегляньте цю публікацію в блозі, якщо вам потрібен детальний посібник із їх встановлення: https://www.nodejsdesignpatterns.com/blog/5-ways-to-install-node-js .

Якщо ви не хочете встановлювати Node.js на вашій локальній машині, деякі онлайн-платформи дозволять вам наслідувати приклади коду в цій книзі, використовуючи онлайн-IDE безкоштовно, наприклад https://codesandbox.io і https://repl.it .

Після встановлення обох Node.js і npm (або ви використовуєте онлайн-середовище), вам потрібно лише слідувати інструкціям, які відображаються в кожному розділі цієї книги, для встановлення необхідних залежностей для конкретного проекту за допомогою npm.

Ви можете знайти повні приклади коду на GitHub у такому репозиторії: https://github.com/PacktPublishing/Real-World-Next.js . Не соромтеся розгалужувати, клонувати та редагувати це сховище для будь-яких експериментів із Next.js.

Представляємо Next.js

За останні кілька років веб-розробка сильно змінилася. До появи сучасних фреймворків JavaScript створення динамічних веб-додатків було складним і вимагало багато різних бібліотек і конфігурацій, щоб вони працювали належним чином.

Angular, React, Vue та всі інші фреймворки сприяли дуже швидкому розвитку Інтернету та принесли з собою кілька дуже інноваційних ідей для зовнішньої веб-розробки.

React, зокрема, був створений Джорданом Волке (Jordan Walke) у Facebook і на нього сильно вплинула XHP Hack Library. XHP дозволив розробникам PHP і Hack від Facebook створювати повторно використовувані компоненти для інтерфейсу своїх програм. Бібліотека JavaScript стала відкритим вихідним кодом у 2013 році та назавжди змінила те, як ми створюємо веб-сайти, веб-програми, рідні програми (з React Native пізніше) і навіть досвід VR (з React VR). У результаті React швидко став однією з

найулюбленіших і найпопулярніших бібліотек JavaScript, і мільйони веб-сайтів використовують її для багатьох різних цілей.

Була лише одна проблема: за замовчуванням React працює на стороні клієнта (це означає, що він працює у веб-браузері), тому веб-програма, повністю написана за допомогою цієї бібліотеки, може негативно вплинути на оптимізацію пошукових систем (SEO) і початкову продуктивність завантаження, оскільки для правильного відображення на екрані потрібен деякий час. Насправді, щоб відобразити повну веб-програму, браузер повинен був завантажити весь пакет програм, проаналізувати його вміст, а потім виконати його та відобразити результат у браузері, що могло зайняти кілька секунд (для дуже великих програм).

Багато компаній і розробників почали досліджувати, як попередньо відобразити програму на сервері, дозволивши браузеру відображати відрендерену програму React як звичайний HTML, роблячи її інтерактивною, щойно пакет JavaScript буде передано клієнту.

Тоді **Vercel** придумав Next.js, який, як виявилося, змінив правила гри.

З моменту свого першого випуску фреймворк надав багато інноваційних функцій із коробки, таких як автоматичний поділ коду, рендеринг на стороні сервера, системи маршрутизації на основі файлів, попередня вибірка маршруту тощо. Next.js показав, наскільки легко має бути написання універсальних веб-додатків, дозволяючи розробникам писати багаторазовий код як для клієнтської, так і для серверної сторони та полегшуючи реалізацію дуже складних завдань (таких як розбиття коду та рендеринг на стороні сервера).

Сьогодні Next.js надає безліч нових функцій із коробки, наприклад:

• Генерація статичного сайту

• Інкрементна статична генерація

• Вбудована підтримка TypeScript

• Автоматичні полізаповнення

• Оптимізація зображення

• Підтримка інтернаціоналізації

• Аналітика продуктивності

Усе це разом із багатьма іншими чудовими функціями, які ми детально розглянемо пізніше в цій книзі.

Сьогодні Next.js використовується у виробництві такими компаніями найвищого рівня, як Netflix, Twitch, TikTok, Hulu, Nike, Uber, Elastic та багатьма іншими. Якщо вам цікаво, ви можете прочитати повний список на https://nextjs.org/showcase.

Next.js показав, наскільки універсальним може бути React для створення багатьох різних програм у будь-якому масштабі, і не дивно бачити, що його використовують як великі компанії, так і маленькі стартапи. До речі, це не єдина структура, яка дозволяє відтворювати JavaScript на стороні сервера, як ми побачимо в наступному розділі.

Порівняння Next.js з іншими альтернативами

Як вам може бути цікаво, Next.js — не єдиний гравець у світі JavaScript, що відображається на стороні сервера. Однак альтернативи можуть бути розглянуті залежно від кінцевої мети проекту.

Gatsby

Однією з популярних альтернатив є Gatsby. Ви можете розглянути цей фреймворк, якщо хочете створювати статичні веб-сайти. На відміну від Next.js, Gatsby підтримує лише статичну генерацію сайтів і робить це неймовірно добре. Кожна сторінка попередньо візуалізується під час створення та може обслуговуватися в будь-якій мережі доставки вмісту (CDN) як статичний актив, завдяки чому продуктивність буде неймовірно конкурентоспроможною порівняно з альтернативами, які динамічно відображаються на стороні сервера. Найбільшим недоліком використання Gatsby замість Next.js є те, що ви втратите можливість динамічного рендерингу на стороні сервера, що є важливою функцією для створення більш динамічних і складних веб-сайтів, керованих даними.

Razzle

Менш популярний, ніж Next.js, Razzle — це інструмент для створення додатків JavaScript, що відображаються на стороні сервера. Він спрямований на збереження простоти використання create-react-app, водночас абстрагуючи всі складні конфігурації, необхідні для відтворення програми як на стороні сервера, так і на стороні клієнта. Найважливішою перевагою використання Razzle замість Next.js (або наступних альтернатив) є те, що він не залежить від фреймворку. Ви можете вибрати свій улюблений фреймворк (або мову), такий як React, Vue, Angular, Elm або Reason-React… це ваш вибір.

Nuxt.js

Якщо у вас є досвід роботи з Vue, тоді Nuxt.js може стати дійсним конкурентом Next.js. Обидва вони пропонують підтримку візуалізації на стороні сервера, створення статичних сайтів, прогресивне керування веб-програмами тощо, без істотних відмінностей щодо продуктивності, пошукової оптимізації чи швидкості розробки. Хоча Nuxt.js і Next.js служать одній меті, Nuxt.js потребує додаткової конфігурації, що іноді непогано. У вашому конфігураційному файлі Nuxt.js ви можете визначати макети, глобальні плагіни та компоненти, маршрути тощо, тоді як у Next.js це потрібно робити у спосіб React. Крім того, вони мають багато спільних функцій, але найбільш суттєвою відмінністю є бібліотека під ними. Проте, якщо у вас уже є бібліотека компонентів Vue, ви можете розглянути Nuxt.js для її відтворення на стороні сервера.

Angular Universal

Звичайно, Angular також перейшов на сцену рендеринга на стороні сервера JavaScript і пропонує Angular Universal як офіційний спосіб для рендерингу додатків Angular на стороні сервера. Він підтримує створення статичних сайтів і рендеринг на стороні сервера, і, на відміну від Nuxt.js і Next.js, його розробила одна з найбільших компаній: Google. Отже, якщо ви любите розробку Angular і вже маєте деякі компоненти, написані за допомогою цієї бібліотеки, Angular Universal може стати природною альтернативою Nuxt.js, Next.js та іншим подібним фреймворкам.

Перехід від React до Next.js

Якщо у вас уже є певний досвід роботи з React, вам буде неймовірно легко створити свій перший веб-сайт Next.js. Його філософія дуже близька до React і передбачає підхід конвенційного налаштування для більшості своїх налаштувань, тож якщо ви хочете скористатися певною функцією Next.js, ви легко знайдете офіційний спосіб зробити це без будь-якої потреби складної конфігурації. Приклад? В одному додатку Next.js ви можете вказати, які сторінки відображатимуться на стороні сервера, а які – статично генеруватися під час збірки без необхідності писати файли конфігурації чи щось подібне. Вам просто потрібно експортувати певну функцію зі своєї сторінки й дозволити Next.js творити свою магію (ми побачимо це в Розділі 2 «Дослідження різних стратегій рендерингу»).

Найсуттєвіша відмінність між React і Next.js полягає в тому, що в той час як React — це просто бібліотека JavaScript, Next.js — це платформа для побудови насиченого та повного досвіду користувачів як на стороні клієнта, так і на стороні сервера, додаючи масу неймовірно корисних функцій. Кожна відображена на сервері або статично згенерована сторінка працюватиме на

Node.js, тому ви втратите доступ до деяких глобальних об'єктів веб-переглядача, таких як вибірка, вікно та документ, а також до деяких елементів HTML, як-от canvas. Вам завжди потрібно пам'ятати про це, коли ви пишете свої сторінки Next.js, навіть якщо фреймворк надає власний спосіб роботи з компонентами, які повинні використовувати такі глобальні змінні та елементи HTML, як ми побачимо в розділі 2.

З іншого боку, можуть бути випадки, коли ви захочете використовувати спеціальні бібліотеки або API Node.js, такі як fs або child_process, і Next.js дозволяє використовувати їх, запускаючи ваш серверний код на кожному запиті або під час збірки (залежно від способу відтворення сторінок) перед надсиланням даних клієнту.

Але навіть якщо ви хочете створити client-side rendered програму, Next.js може стати чудовою альтернативою добре відомому додатку create-react-app. Насправді Next.js можна легко використовувати як основу для написання прогресивних веб-додатків, перш за все в автономному режимі, використовуючи переваги його неймовірних вбудованих компонентів і оптимізацій. Отже, почнемо з Next.js.

Початок роботи з Next.js

Тепер, коли ми маємо базові знання про випадки використання Next.js і відмінності між React на стороні клієнта та іншими фреймворками, настав час поглянути на код. Ми почнемо зі створення нової програми Next.js і налаштування її стандартних конфігурацій webpack і Babel. Ми також побачимо, як використовувати TypeScript як основну мову для розробки програм Next.js.

Структура проекту за замовчуванням

Почати роботу з Next.js неймовірно легко. Єдина вимога до системи — наявність Node.js і npm, встановлених на вашій машині (або середовищі розробки). Команда Vercel створила та опублікувала простий, але потужний інструмент під назвою createnext-app для генерації шаблонного коду для базової програми Next.js. Ви можете використовувати його, ввівши таку команду в терміналі:

```
npx create-next-app <app-name>
```

Він встановить усі необхідні залежності та створить пару сторінок за замовчуванням. На цьому етапі ви можете просто запустити npm run dev, і сервер розробки запуститься на порту 3000, показуючи цільову сторінку.

Next.js ініціалізує ваш проект за допомогою менеджера пакетів Yarn, якщо його встановлено на вашій машині. Ви можете перевизначити цей параметр, передавши прапорець, щоб сказати create-next-app використовувати натомість npm:

npx create-next-app <app-name> --use-npm

Ви також можете попросити create-next-app ініціалізувати новий проект Next.js, завантаживши шаблонний код із репозиторію Next.js GitHub. Насправді всередині репозиторію Next.js є папка examples, яка містить масу чудових прикладів того, як використовувати Next.js із різними технологіями.

Скажімо, ви хочете провести деякі експерименти з використанням Next.js на Docker – ви можете просто передати прапорець --example генератору шаблонного коду:

npx create-next-app <app-name> --example with-docker

create-next-app завантажить код із https://github.com/vercel/next.js/tree/canary/examples/with-docker і встановить необхідні залежності для вас. На цьому етапі вам потрібно лише відредагувати завантажені файли, налаштувати їх, і ви готові до роботи.

А тепер давайте на мить повернемося до стандартної інсталяції програми create-next-app. Давайте відкриємо термінал і створимо нову програму Next.js разом:

npx create-next-app my-first-next-app --use-npm

Через кілька секунд генерація шаблону завершиться успішно, і ви знайдете нову папку під назвою my-first-next-app із такою структурою:

```
- README.md
- next.config.js
- node_modules/
- package-lock.json
- package.json
- pages/
  - _app.js
  - api/
    - hello.js
  - index.js
- public/
  - favicon.ico
  - vercel.svg
- styles/
  - Home.module.css
  - globals.css
```

Якщо ви працюєте з React, ви можете використовувати react-router або подібні бібліотеки для керування навігацією на стороні клієнта. Next.js робить навігацію ще легшою за допомогою папки pages/. Насправді кожен файл JavaScript у каталозі pages/ буде загальнодоступною сторінкою, тож якщо ви спробуєте скопіювати сторінку index.js і перейменувати її на about.js, ви зможете перейти до http://localhost:3000 /about і переглянути точну копію вашої домашньої сторінки. У наступному розділі ми детально розглянемо, як Next.js обробляє маршрути на стороні клієнта та на стороні сервера; Наразі давайте просто подумаємо про каталог pages/ як про контейнер для ваших загальнодоступних сторінок.

Папка public/ містить усі публічні та статичні ресурси, які використовуються на вашому веб-сайті. Наприклад, ви можете розмістити туди свої зображення, скомпільовані таблиці стилів CSS, скомпільовані файли JavaScript, шрифти тощо.

За замовчуванням ви також побачите каталог styles/; Хоча це дуже корисно для організації таблиць стилів вашої програми, це не обов’язково для проекту Next.js. Єдиними обов’язковими та зарезервованими каталогами є public/ та pages/, тому не видаляйте та не використовуйте їх для інших цілей.

Тим не менш, ви можете додавати більше каталогів і файлів до кореня проекту, оскільки це не заважатиме процесу збирання чи розробки Next.js. Якщо ви хочете впорядкувати свої компоненти в каталозі components/, а утиліти — у каталозі utilities/, не соромтеся додати ці папки у свій проект.

Якщо вам не подобаються шаблонні генератори, ви можете запустити нову програму Next.js, просто додавши всі необхідні залежності (як було зазначено раніше) і основну структуру папок, яку ми щойно бачили, до вашої існуючої програми React, і вона просто працювати без будь-якої іншої конфігурації.

Інтеграція TypeScript

Вихідний код Next.js написаний на TypeScript і нативно надає високоякісні визначення типів, щоб зробити роботу розробника ще кращою. Налаштувати TypeScript як мову за замовчуванням для програми Next.js дуже просто; вам просто потрібно створити файл конфігурації TypeScript (tsconfig.json) у кореневій папці вашого проекту. Якщо ви спробуєте запустити npm run dev, ви побачите такий результат:

```
It looks like you're trying to use TypeScript but do not have
the required package(s) installed.
Please install typescript and @types/react by running:
     npm install --save typescript @types/react
     If you are not trying to use TypeScript, please remove
     the tsconfig.json file from your package root (and any
     TypeScript files in your pages directory).
```

Як бачите, Next.js правильно виявив, що ви намагаєтеся використовувати TypeScript, і просить вас встановити всі необхідні залежності для використання його як основної мови для вашого проекту. Тож тепер вам просто потрібно конвертувати файли JavaScript у TypeScript, і ви готові до роботи.

Ви можете помітити, що навіть якщо ви створили порожній файл tsconfig.json, після встановлення необхідних залежностей і повторного запуску проекту Next.js заповнює його стандартними конфігураціями. Звичайно, ви завжди можете налаштувати параметри TypeScript у цьому файлі, але пам’ятайте, що Next.js використовує Babel для обробки файлів TypeScript (через @babel/plugin-transform-typescript), і він має деякі застереження, зокрема такі:

• Плагін @babel/plugin-transform-typescript не підтримує const enum, який часто використовується у TypeScript. Щоб підтримувати його, обов’язково додайте babelplugin-const-enum до конфігурації Babel (ми побачимо, як це зробити в розділі Custom Babel and webpack configuration).

• Ні export =, ні import = не підтримуються, оскільки їх неможливо скомпілювати до дійсного коду ECMAScript. Вам слід або встановити babel-plugin-replacets-export-assignment, або перетворити ваші імпорти та експорти на дійсні директиви ECMAScript, такі як import x, {y} from 'some-package' і export default x.

Є й інші застереження; Я пропоную вам прочитати їх перед тим, як використовувати TypeScript як основну мову для розробки програми Next.js: https://babeljs.io/docs/en/babel-plugin-transform-typescript#caveats.

Крім того, деякі параметри компілятора можуть дещо відрізнятися від типових у TypeScript; Я ще раз пропоную вам прочитати офіційну документацію Babel, яка завжди буде актуальною: https://babeljs.io/docs/en/babel-plugin-transformtypescript#typescript-compiler-options .

Next.js також створює файл next-env.d.ts у корені вашого проекту; не соромтеся редагувати його, якщо потрібно, але не видаляйте його.

Спеціальна конфігурація Babel і webpack

Як уже згадувалося в розділі про інтеграцію TypeScript, ми можемо налаштувати конфігурації Babel і webpack.

. . . . .

Розділ 2

Вивчення різних стратегій візуалізації

Говорячи про стратегії візуалізації, ми маємо на увазі те, як ми обслуговуємо веб-сторінку (або веб-програму) у веб-браузері. Існують фреймворки, такі як Gatsby (як показано в попередньому розділі), які неймовірно добре обслуговують статично згенеровані сторінки. Інші фреймворки полегшать створення сторінок, які відображаються на стороні сервера.

Але Next.js виводить ці концепції на абсолютно новий рівень, дозволяючи вирішувати, яку сторінку потрібно відображати під час створення, а яку – динамічно під час виконання, регенеруючи всю сторінку для кожного запиту, роблячи певні частини ваших програм неймовірно динамічними. Фреймворк також дозволяє вам вирішувати, які компоненти повинні відтворюватися виключно на стороні клієнта, що робить ваш досвід розробки надзвичайно задовільним.

У цьому розділі ми докладніше розглянемо:

• Як динамічно відтворювати сторінку для кожного запиту за допомогою відтворення на стороні сервера

• Різні способи візуалізації певних компонентів лише на стороні клієнта

• Створення статичних сторінок під час створення

• Як відновити статичні сторінки у виробництві за допомогою поступової статичної регенерації

Технічні вимоги

Щоб запустити приклади коду в цьому розділі, переконайтеся, що на вашій машині встановлено Node.js і npm. Як альтернативу ви можете

використовувати онлайн-IDE, наприклад https://repl.it або https://codesandbox.io .

Ви можете знайти код цієї глави в репозиторії GitHub: https://github.com/PacktPublishing/Real-World-Next.js

Візуалізація на стороні сервера (SSR)

Незважаючи на те, що server-side rendering (SSR) звучить як новий термін у словнику розробника, насправді це найпоширеніший спосіб обслуговування веб-сторінок. Якщо ви думаєте про такі мови, як PHP, Ruby або Python, усі вони відображають HTML на сервері перед тим, як надсилати його в браузер, що зробить розмітку динамічною після завантаження всього вмісту JavaScript.

Ну, Next.js робить те саме, динамічно відтворюючи HTML-сторінку на сервері для кожного запиту, а потім надсилаючи її веб-браузеру. Фреймворк також впроваджуватиме власні сценарії, щоб зробити відображені на стороні сервера сторінки динамічними в процесі, який називається **hydration** (гідратація).

Уявіть, що ви створюєте блог і хочете відобразити всі статті, написані певним автором, на одній сторінці. Це може бути чудовим варіантом використання SSR: користувач хоче отримати доступ до цієї сторінки, тому сервер відтворює її та надсилає отриманий HTML клієнту. На цьому етапі браузер завантажить усі сценарії, які вимагає сторінка, і гідратує DOM, роблячи його інтерактивним без будь-якого оновлення сторінки чи збоїв (ви можете прочитати більше про гідратацію React на https://reactjs.org/docs/react-dom.html#hydrate). З цього моменту, завдяки гідратації React, веб-програма також може стати **single-page application (SPA)**, використовуючи всі переваги **client-side rendering (CSR)** (як ми побачимо в наступному розділі), так і SSR .

Говорячи про переваги застосування конкретної стратегії візуалізації, SSR надає численні переваги порівняно зі стандартним React CSR:

• ***Більш безпечні веб-програми***: рендеринг сторінки на стороні сервера означає, що такі дії, як керування файлами cookie, виклик приватних API і перевірка даних, відбуваються на сервері, тому ми ніколи не відкриватимемо особисті дані клієнту.

• ***Більш сумісні веб-сайти***: веб-сайт буде доступним, навіть якщо користувач вимкнув JavaScript або використовує застарілий браузер.

• ***Покращена оптимізація пошукової системи***: оскільки клієнт отримає HTML-вміст, щойно сервер відобразить і надішле його, павукам пошукової системи (ботам, які сканують веб-сторінки) не доведеться чекати, поки сторінка буде відображена на боці клієнта. Це покращить показник SEO вашої веб-програми.

Незважаючи на ці великі переваги, іноді SSR може бути не найкращим рішенням для вашого веб-сайту. Фактично, за допомогою SSR вам потрібно буде розгорнути свою веб-програму на сервері, який повторно відобразить сторінку, щойно це буде потрібно. Як ми побачимо пізніше, за допомогою як CSR, так і статичної генерації сайтів (SSG) ви можете розгортати статичні файли HTML у будь-якому хмарному провайдері, такому як Vercel або Netlify, безкоштовно (або за мізерну плату); якщо ви вже розгортаєте свою веб-програму за допомогою спеціального сервера, ви повинні пам’ятати, що програма SSR завжди призведе до значного навантаження на сервер і витрат на обслуговування.

Інша річ, про яку слід пам’ятати, якщо ви хочете відобразити свої сторінки на стороні сервера, це те, що ви додаєте деяку затримку до кожного запиту; Вашим сторінкам, можливо, знадобиться викликати якийсь зовнішній API або джерело даних, і вони викликатимуть його для кожної сторінки. Навігація між сторінками, відображеними на стороні сервера, завжди буде трохи повільнішою, ніж переміщення між сторінками, відображеними на стороні клієнта, або статично обслуговуваними сторінками.

Звичайно, Next.js надає деякі чудові можливості для покращення продуктивності навігації, як ми побачимо в Розділі 3, Основи Next.js і вбудовані компоненти.

Ще одна річ, яку слід враховувати, це те, що за замовчуванням сторінка Next.js генерується статично під час збірки. Якщо ми хочемо зробити його більш динамічним, викликавши зовнішній API, базу даних або інші джерела даних, нам потрібно буде експортувати певну функцію з нашої сторінки:

```
function IndexPage() {
  return <div>This is the index page.</div>;
}
export default IndexPage;
```

Як бачите, сторінка друкує лише ***This is the index page***. текст всередині div. Для роботи йому не потрібно викликати зовнішні API або будь-які інші джерела даних, а його вміст завжди буде однаковим для кожного запиту. Але тепер давайте уявимо, що ми хочемо вітати користувача з кожним запитом; нам потрібно буде викликати REST API на сервері, щоб отримати певну

інформацію про користувача та передати результат клієнту за допомогою потоку Next.js. Ми зробимо це за допомогою зарезервованої функції getServerSideProps:

```
export async function getServerSideProps() {
  const userRequest =
    await fetch('https://example.com/api/user');
  const userData = await userRequest.json();
  return {
    props: {
      user: userData
    }
  };
}
function IndexPage(props) {
  return <div>Welcome, {props.user.name}!</div>;
}
export default IndexPage;
```

У попередньому прикладі ми використовували зарезервовану функцію getServerSideProps Next.js для виклику REST API на стороні сервера для кожного запиту. Давайте розберемо це на маленькі кроки, щоб краще зрозуміти, що ми робимо:

1. Ми починаємо з експорту асинхронної функції під назвою getServerSideProps. На етапі збірки Next.js шукатиме кожну сторінку, яка експортує цю функцію, і динамічно відображатиме їх на стороні сервера для кожного запиту. Весь код, написаний у межах цієї функції, завжди виконуватиметься на стороні сервера.

2. Усередині функції getServerSideProps ми повертаємо об’єкт, що містить властивість під назвою props. Це потрібно, оскільки Next.js вставлятиме ці атрибути всередину нашого компонента сторінки, роблячи їх доступними як на стороні клієнта, так і на стороні сервера. Якщо вам цікаво, нам не потрібно заповнювати API вибірки, коли ми використовуємо його на стороні сервера, оскільки Next.js вже робить це за нас.

3. Потім ми виконуємо рефакторинг функції IndexPage, яка тепер приймає параметр props, що містить усі властивості, передані функцією getServerSideProps.

І це все! Після того, як ми надішлемо цей код, Next.js завжди динамічно відтворюватиме нашу IndexPage на сервері, викликаючи зовнішній API і показуючи різні результати, щойно ми внесемо зміни в наше джерело даних.

Як було показано на початку цього розділу, SSR надає деякі значні переваги, але має деякі застереження. Якщо ви хочете використовувати будь-який компонент, який покладається на специфічні API браузера, вам потрібно буде відобразити його в браузері явно, оскільки за замовчуванням Next.js відображає весь вміст сторінки на сервері, який не надає певні API, наприклад як вікно або документ. Отже, з'являється концепція CSR.

Візуалізація на стороні клієнта (CSR)

Як було показано в попередньому розділі, стандартний додаток React відображається після того, як пакет JavaScript було передано з сервера на клієнт.

Якщо ви знайомі з додатком create-react-app (CRA), ви, можливо, помітили, що безпосередньо перед відтворенням веб-додатка вся веб-сторінка повністю біла. Це тому, що сервер обслуговує лише базову розмітку HTML, яка містить усі необхідні сценарії та стилі, щоб зробити нашу веб-програму динамічною. Давайте ближче розглянемо цей HTML, створений CRA:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
  <meta
    name="viewport"
    content="width=device-width, initial-scale=1"
  />
  <meta name="theme-color" content="#000000" />
  <meta
    name="description"
    content="Web site created using create-react-app"
  />
  <link rel="apple-touch-icon"
    href="%PUBLIC_URL%/logo192.png" />
  <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
  <title>React App</title>
</head>
<body>
  <noscript>
    You need to enable JavaScript to run this app.
  </noscript>
  <div id="root"></div>
</body>
</html>
```

Як бачите, ми можемо знайти лише один div всередині тегу body: .

На етапі збірки create-react-app додасть скомпільовані файли JavaScript і CSS на цю HTML-сторінку та використає кореневий div як цільовий контейнер для відтворення всієї програми.

Це означає, що коли ми публікуємо цю сторінку в будь-якому хостинг-провайдері (Vercel, Netlify, Google Cloud, AWS тощо), під час першого виклику потрібної URL-адреси наш браузер спочатку відтворить попередній HTML. Потім, дотримуючись тегів сценарію та посилань, що містяться в попередній розмітці (введеній CRA під час створення), браузер відтворить всю програму, зробивши її доступною для будь-якої взаємодії.

Основними перевагами CRA є:

• ***Завдяки цьому ваша програма виглядає як нативна програма***: завантаження всього пакета JavaScript означає, що ви вже завантажили кожну сторінку веб-програми у свій браузер. Якщо ви хочете перейти на іншу сторінку, він замінить вміст сторінки замість завантаження нового вмісту із сервера. Вам не потрібно оновлювати сторінку, щоб оновити її вміст.

• ***Спрощено переходи між сторінками***: навігація на стороні клієнта дозволяє переходити з однієї сторінки на іншу, не перезавантажуючи вікно браузера. Це стане в нагоді, коли ви хочете легко показати якісь круті переходи між сторінками, оскільки у вас немає перезавантаження, яке могло б перервати вашу анімацію.

• ***Відкладене завантаження та продуктивність***: за допомогою CSR браузер відображатиме лише мінімальну розмітку HTML, необхідну для роботи веб-програми. Якщо у вас є модаль, яка з'являється, коли користувач натискає кнопку, її HTML-розмітка відсутня на сторінці HTML. Він буде створений динамічно React, щойно відбудеться подія натискання кнопки.

• ***Менше робоче навантаження на стороні сервера***: враховуючи, що весь етап рендерингу делеговано браузеру, серверу потрібно лише надіслати клієнту дуже просту сторінку HTML. Тоді вам не потрібен дуже потужний сервер; дійсно, є випадки, коли ви можете розмістити свою веб-програму в безсерверних середовищах, таких як AWS Lambda, Firebase тощо.

Але всі ці переваги мають свою ціну. Як ми бачили раніше, сервер надсилає лише порожню HTML-сторінку. Якщо інтернет-з'єднання користувача повільне, завантаження файлів JavaScript і CSS триватиме кілька секунд, залишаючи користувача чекати з порожнім екраном кілька хвилин.

Це також вплине на показник SEO вашої веб-програми; павуки пошукової системи досягнуть вашої сторінки та виявлять її порожньою. Наприклад, боти Google чекатимуть, поки буде передано пакет JavaScript, але присвоять вашому веб-сайту низьку оцінку ефективності через час очікування.

За замовчуванням Next.js рендерить усі компоненти React всередині певної сторінки на стороні сервера (як показано в попередньому розділі) або під час збірки. У першому розділі, у розділі «Перехід від React до Next.js», ми побачили, що середовище виконання Node.js не надає доступ до деяких специфічних для браузера API, таких як вікно чи документ, або елементів HTML, таких як canvas, тому, якщо Якщо ви спробуєте відобразити будь-який компонент, якому потрібен доступ до цих API, процес відтворення завершиться збоєм.

Існує багато різних способів уникнути подібних проблем із Next.js, які вимагають відтворення певних компонентів у браузері.

Використання хука React.useEffect

Якщо ви працюєте з версією React до 16.8.0, можливо, ви звикли до методу componentDidMount класу React.Component. З більш сучасними версіями React, які наголошують на використанні ***функціональних компонентів***, ви можете досягти тих же результатів, використовуючи хук React.useEffect. Це дозволить вам виконувати побічні ефекти (такі як вибірка даних і ручні зміни DOM) у ваших функціональних компонентах, і це буде зроблено після монтування компонента. Це означає, що з Next.js зворотний виклик useEffect запускатиметься у браузері після гідратації React, дозволяючи виконувати певні дії лише на стороні клієнта.

Наприклад, давайте уявимо, що ми хочемо відобразити фрагмент коду на веб-сторінці за допомогою бібліотеки Highlight.js, що полегшує виділення та робить код більш читабельним. Ми могли б просто створити компонент під назвою Highlight, який би виглядав наступним чином:

```
import Head from 'next/head';
import hljs from 'highlight.js';
import javascript from 'highlight.js/lib/languages/javascript';
function Highlight({ code }) {
  hljs.registerLanguage('javascript', javascript);
  hljs.initHighlighting();
  return (
    <>
      <Head>
        <link rel='stylesheet' href='/highlight.css' />
      </Head>
```

```
      <pre>
        <code className='js'>{code}</code>
      </pre>
    </>
  );
}
export default Highlight;
```

Незважаючи на те, що цей фрагмент коду ідеально запускатиметься в програмі React на стороні клієнта, він аварійно завершуватиме роботу під час візуалізації чи збирання Next.js, оскільки Highlight.js потребує глобальної змінної документа, якої немає в Node.js, оскільки це є доступним лише браузерам.

Ви можете легко виправити це, загорнувши всі виклики hljs у хук useEffect:

```
import { useEffect } from 'react';
import Head from 'next/head';
import hljs from 'highlight.js';
import javascript from
  'highlight.js/lib/languages/javascript';
function Highlight({ code }) {
  useEffect(() => {
    hljs.registerLanguage('javascript', javascript);
    hljs.initHighlighting();
  }, []);
  return (
    <>
      <Head>
        <link rel='stylesheet' href='/highlight.css' />
      </Head>
      <pre>
        <code className='js'>{code}</code>
      </pre>
    </>
  );
}
export default Highlight;
```

Таким чином, Next.js візуалізує розмітку HTML, яку повертає наш компонент, вставляє сценарій Highlight.js на нашу сторінку, і, як тільки компонент буде змонтовано в браузері, він викличе функції бібліотеки на стороні клієнта.

Ви також можете використовувати саме цей підхід для відтворення компонента виключно на стороні клієнта, використовуючи React.useEffect і React.useState разом:

```
import {useEffect, useState} from 'react';
import Highlight from '../components/Highlight';
function UseEffectPage() {
  const [isClient, setIsClient] = useState(false);
  useEffect(() => {
    setIsClient(true);
  }, []);
  return (
    <div>
      {isClient &&
        (<Highlight
          code={"console.log('Hello, world!')"}
          language='js'
        />)
      }
    </div>
  );
}
export default UseEffectPage;
```

Таким чином, компонент Highlight відображатиметься виключно в браузері.

Використання змінної process.browser

Ще один спосіб уникнути збою процесу на стороні сервера під час використання специфічних для браузера API — це умовне виконання сценаріїв і компонентів залежно від процесу. глобальна змінна браузера. Дійсно, Next.js додає цю неймовірно корисну властивість до об’єкта процесу Node.js. Це логічне значення, яке має значення true, коли код виконується на стороні клієнта, і false, коли виконується на сервері. Давайте подивимося, як це працює:

```
function IndexPage() {
  const side = process.browser ? 'client' : 'server';
  return <div>You're currently on the {side}-side.</div>;
}
export default IndexPage;
```

Якщо ви спробуєте запустити попередній приклад, ви помітите, що на короткий момент браузер покаже такий текст: **You're currently running on the**

server-side;; його буде замінено текстом «**You're currently running on the client-side**», щойно відбудеться гідратація React.

Використання динамічного завантаження компонентів

Як ми бачили в першому розділі, Next.js розширює функціональні можливості React, додаючи кілька чудових вбудованих компонентів і службових функцій. Один із них називається *dynamic*, і це один із найцікавіших модулів, які надає фреймворк.

Пам’ятаєте компонент Highlight.js, який ми створили, щоб зрозуміти, як відобразити компонент у браузері за допомогою хука React.useEffect? Ось ще один спосіб відобразити його за допомогою динамічної функції Next.js:

```
import dynamic from 'next/dynamic';
const Highlight = dynamic(
  () => import('../components/Highlight'),
  { ssr: false }
);
import styles from '../styles/Home.module.css';
function DynamicPage() {
  return (
    <div className={styles.main}>
      <Highlight
        code={"console.log('Hello, world!')"}
        language='js'
      />
    </div>
  );
}
export default DynamicPage;
```

За допомогою попереднього коду ми імпортуємо наш компонент Highlight за допомогою динамічного імпорту, вказуючи, що ми хочемо, щоб він виконувався на клієнті лише завдяки параметру ssr: false. Таким чином, Next.js не намагатиметься відобразити цей компонент на сервері, і нам доведеться чекати, поки гідратація React зробить його доступним у браузері.

CSR може бути фантастичною альтернативою SSR для створення дуже динамічних веб-сторінок. Якщо ви працюєте над сторінкою, яку не потрібно індексувати пошуковими системами, може бути доцільним спочатку завантажити JavaScript вашої програми, а потім зі сторони клієнта отримати всі необхідні дані з сервера; це полегшить робоче навантаження на стороні

сервера, оскільки цей підхід не передбачає SSR, і ваша програма може краще масштабуватися.

Отже, ось запитання: якщо нам потрібно створити динамічну сторінку, а пошукова оптимізація не дуже важлива (сторінки адміністратора, сторінки приватного профілю тощо), чому б нам просто не надіслати статичну сторінку клієнту та завантажити всі дані після того, як сторінку було передано в браузер? Ми розглянемо цю можливість у наступному розділі.

Генерація статичного сайту

Наразі ми бачили два різні способи відтворення наших веб-програм: на стороні клієнта та на стороні сервера. Next.js надає нам третій варіант, який називається статичною генерацією сайту (SSG).

За допомогою SSG ми зможемо попередньо відобразити деякі конкретні сторінки (або навіть весь веб-сайт, якщо необхідно) під час створення; це означає, що коли ми створюємо наш веб-додаток, можуть існувати деякі сторінки, вміст яких змінюватиметься не дуже часто, тому нам має сенс використовувати їх як статичні ресурси. Next.js візуалізує ці сторінки на етапі створення та завжди обслуговуватиме той конкретний HTML, який, як і SSR, стане інтерактивним завдяки процесу гідратації React.

SSG має багато переваг у порівнянні як з CSR, так і з SSR:

• ***Легко масштабувати***: статичні сторінки — це просто файли HTML, які можна легко обслуговувати та кешувати будь-якою мережею доставки вмісту (відтепер CDN). Але навіть якщо ви хочете обслуговувати їх за допомогою власного веб-сервера, це призведе до дуже низького робочого навантаження, враховуючи те, що для обслуговування статичного активу не потрібні важкі обчислення.

• ***Видатна продуктивність***: як було сказано раніше, HTML попередньо візуалізується під час створення, тому і клієнт, і сервер можуть обійти фазу рендерингу під час виконання для кожного запиту. Веб-сервер надішле статичний файл, а браузер просто відобразить його. Не потрібно отримання даних на стороні сервера; все, що нам потрібно, уже попередньо відображено всередині статичної розмітки HTML, і це зменшує потенційну затримку для кожного запиту.

• ***Більш безпечні запити***: нам не потрібно надсилати будь-які конфіденційні дані на веб-сервер для відтворення сторінки, і це дещо

ускладнює життя зловмисним користувачам. Доступ до API, баз даних чи іншої приватної інформації не потрібен, оскільки кожна необхідна інформація вже є частиною попередньо відтвореної сторінки.

SSG, мабуть, є одним із найкращих рішень для створення продуктивних і високомасштабованих інтерфейсних програм. Найбільше занепокоєння щодо цієї техніки відтворення полягає в тому, що після створення сторінки вміст залишатиметься незмінним до наступного розгортання.

Наприклад, давайте уявимо, що ми пишемо допис у блозі та неправильно пишемо слово в заголовку. Використовуючи інші генератори статичних сайтів, такі як Gatsby або Jekyll, нам потрібно було б перебудувати весь веб-сайт, щоб змінити лише одне слово в заголовку публікації блогу, оскільки нам потрібно було б повторити етап отримання та відтворення даних під час створення. Пам'ятайте, що ми сказали на початку цього розділу: статично згенеровані сторінки створюються під час збирання та подаються як статичні ресурси для кожного запиту.

Хоча це справедливо для інших генераторів статичних сайтів, Next.js пропонує унікальний підхід для вирішення цієї проблеми: поступову статичну регенерацію (***incremental static regeneration ISR***). Завдяки ISR ми можемо вказати на рівні сторінки, скільки часу Next.js має чекати перед повторним відтворенням статичної сторінки з оновленням її вмісту.

Наприклад, припустімо, що ми хочемо створити сторінку з динамічним вмістом, але етап отримання даних з певних причин триває надто довго. Це призведе до поганої продуктивності, що дасть нашим користувачам жахливий досвід роботи. Комбінація SSG та ISR вирішила б цю проблему шляхом використання гібридного підходу між SSR та SSG.

Уявімо, що ми створили дуже складну інформаційну панель, яка може обробляти велику кількість даних… але для виконання запиту REST API для цих даних потрібно кілька секунд. У такому випадку нам пощастило, оскільки ці дані не сильно зміняться протягом цього часу, тому ми можемо кешувати їх до 10 хвилин (600 секунд) за допомогою SSG та ISR:

```
import fetch from 'isomorphic-unfetch';
import Dashboard from './components/Dashboard';
export async function getStaticProps() {
  const userReq = await fetch('/api/user');
  const userData = await userReq.json();
  const dashboardReq = await fetch('/api/dashboard');
  const dashboardData = await dashboardReq.json();
  return {
    props: {
```

```
      user: userData,
      data: dashboardData,
    },
    revalidate: 600 // time in seconds (10 minutes)
  };
}

function IndexPage(props) {
  return (
    <div>
      <Dashboard
        user={props.user}
        data={props.data}
      />
    </div>
  );
}
export default IndexPage;
```

Зараз ми використовуємо функцію під назвою getStaticProps, яка схожа на функцію getServerSideProps, яку ми бачили в попередньому розділі. Як ви вже здогадалися, getStaticProps використовується під час збірки Next.js для отримання даних і відтворення сторінки, і він не буде викликаний знову до наступної збірки. Як було сказано раніше, хоча це може бути неймовірно потужним, воно має певні витрати: якщо ми хочемо оновити вміст сторінки, нам доведеться перебудувати весь веб-сайт.

Щоб уникнути повної перебудови веб-сайту, Next.js нещодавно представив параметр під назвою revalidate, який можна встановити всередині об’єкта, що повертається, нашої функції getStaticProps. Він вказує, через скільки секунд ми повинні перебудувати сторінку після надходження нового запиту.

У попередньому коді ми встановили параметр повторної перевірки на 600 секунд, тому Next.js поводитиметься наступним чином:

1. Next.js заповнює сторінку результатами getStaticProps під час збирання, статично генеруючи сторінку під час процесу збирання.

2. Протягом перших 10 хвилин кожен користувач отримає доступ до тієї самої статичної сторінки.

3. Через 10 хвилин, якщо виникає новий запит, Next.js візуалізує цю сторінку на стороні сервера, повторно виконає функцію getStaticProps, збереже та кешує щойно відрендерену сторінку як статичний ресурс, замінюючи попередню, створену під час створення .

4. Кожен новий запит протягом наступних 10 хвилин буде обслуговуватися разом із цією новою статично згенерованою сторінкою.

Пам'ятайте, що процес ISR є ледачим, тому, якщо через 10 хвилин не буде жодного запиту, Next.js не відновить свої сторінки.

Якщо вам цікаво, на даний момент немає способу примусової повторної перевірки ISR через API; після того, як ваш веб-сайт буде розгорнуто, вам доведеться почекати час закінчення терміну дії, встановлений у параметрі повторної перевірки, щоб сторінку було відновлено.

Генерація статичних сайтів — чудовий спосіб створювати швидкі та безпечні веб-сторінки, але іноді нам може знадобитися більш динамічний вміст. Завдяки Next.js ми завжди можемо вирішити, яку сторінку потрібно відобразити під час створення (SSG) або під час запиту (SSR). Ми можемо використовувати найкраще з обох підходів, використовуючи SSG + ISR, роблячи наші сторінки «гібридом» між SSR і SSG, і це кардинально змінює правила сучасної веб-розробки.

Резюме

У цій главі ми побачили три різні стратегії візуалізації та дізналися, чому Next.js виводить їх на абсолютно новий рівень завдяки гібридному підходу до рендерингу. Ми також побачили переваги цих стратегій, коли ми хочемо їх використовувати, і як вони можуть вплинути на взаємодію з користувачем або навантаження на сервер. Ми завжди будемо стежити за цими методологіями візуалізації протягом наступних розділів, додаючи все більше прикладів і випадків використання для кожного з них. Це основні концепції, що лежать в основі вибору використання Next.js як фреймворку.

У наступному розділі ми детальніше розглянемо деякі з найкорисніших вбудованих компонентів Next.js, його систему маршрутизації та те, як динамічно керувати метаданими для покращення як SEO, так і взаємодії з користувачем.

Розділ 3

Основи Next.js і вбудовані компоненти

Next.js — це не лише рендеринг на стороні сервера. Він надає неймовірно корисні вбудовані компоненти та функції, які ми можемо використовувати для створення ефективних, динамічних і сучасних веб-сайтів.

У цій главі ми розглянемо деякі концепції ядра Next.js, такі як системи маршрутизації, навігація на стороні клієнта, обслуговування оптимізованих зображень, обробка метаданих тощо. Ці поняття будуть дуже корисними, коли ми перейдемо до створення деяких реальних додатків із цією структурою.

Ми також детальніше розглянемо сторінки _app.js і _document.js, які дозволять нам налаштувати поведінку нашої веб-програми кількома способами.

У цьому розділі ми розглянемо такі теми:

• Як працює система маршрутизації як на стороні клієнта, так і на стороні сервера

• Як оптимізувати навігацію між сторінками

• Як Next.js обслуговує статичні ресурси

• Як оптимізувати показ зображень за допомогою автоматичної оптимізації зображення та нового компонента Image

• Як динамічно обробляти метадані HTML з будь-якого компонента

• Що таке файли _app.js і _document.js і як вони можуть

бути налаштованим?

Технічні вимоги

Щоб запустити приклади коду в цьому розділі, вам потрібно інсталювати Node.js і npm на вашій локальній машині.

За бажанням ви можете використовувати онлайн-IDE, наприклад https://repl.it або https://codesandbox.io ; обидва вони підтримують Next.js, і вам не потрібно встановлювати будь-які залежності на вашому комп'ютері.

Ви можете знайти код цієї глави в репозиторії GitHub: https://github.com/PacktPublishing/Real-World-Next.js .

Система маршрутизації

Якщо ви використовуєте React на стороні клієнта, ви можете бути знайомі з такими бібліотеками, як React Router, Reach Router або Wouter. Вони дозволяють створювати лише маршрути на стороні клієнта, тобто всі сторінки створюватимуться та відображатимуться на стороні клієнта; відтворення на стороні сервера не використовується.

Next.js використовує інший підхід: сторінки та маршрути на основі файлової системи. Як показано в *Розділі 2, Вивчення різних стратегій*

візуалізації, типовий проект Next.js постачається з каталогом pages/. Кожен файл у цій папці представляє нову сторінку/маршрут для вашої програми.

Тому, говорячи про сторінку, ми маємо на увазі компонент React, експортований з будь-якого файлу .js, .jsx, .ts або .tsx у папці pages/.

Щоб було трохи зрозуміліше, припустімо, що ми хочемо створити простий веб-сайт лише з двома сторінками; перша буде домашньою сторінкою, а друга буде простою сторінкою контактів. Для цього нам потрібно лише створити два нових файли в папці pages/: index.js і contacts.js. Обидва файли повинні експортувати функцію, яка повертає деякий вміст JSX; він буде відтворений на стороні сервера та надісланий у браузер як стандартний HTML.

Як ми щойно бачили, сторінка має повертати дійсний код JSX, тому давайте створимо дуже просту та лаконічну сторінку index.js:

```
function Homepage() {
  return (
    <div> This is the homepage </div>
  )
};
export default Homepage;
```

Якщо ми запустимо yarn dev або npm run dev у нашому терміналі, а потім перейдемо до http://localhost:3000 у нашому браузері, ми побачимо лише повідомлення *This is the homepage*, що з'явиться на екрані. Ми щойно зробили нашу першу сторінку!

Ми можемо зробити те саме з нашою сторінкою контактів:

```
function ContactPage() {
  return (
    <div>
      <ul>
        <li> Email: myemail@example.com</li>
        <li> Twitter: @myusername </li>
        <li> Instagram: myusername </li>
      </ul>
    </div>
  )
};
export default ContactPage;
```

Враховуючи, що ми назвали нашу сторінку контактів contacts.js, ми можемо перейти до http://localhost:3000/contacts і переглянути список контактів, який відображається у браузері. Якщо ми хочемо перемістити цю сторінку на http://localhost:3000/contact-us , ми можемо просто перейменувати

наш файл contacts.js на contact-us.js, і Next.js автоматично перебудує сторінку, використовуючи нову назву маршруту для нас.

А тепер давайте спробуємо зробити все трохи складніше. Ми створюємо блог, тому хочемо створити маршрут для кожної публікації. Ми також хочемо створити сторінку /posts, на якій буде показано кожну публікацію на веб-сайті.

Для цього ми будемо використовувати динамічний маршрут таким чином:

```
pages/
  - index.js
  - contact-us.js
  - posts/
    - index.js
    - [slug].js
```

Ми ще не згадували, що ми можемо створювати вкладені маршрути за допомогою папок у нашому каталозі pages/. Якщо ми хочемо створити маршрут /posts, ми можемо створити новий файл index.js у папці pages/posts/, експортувати функцію, що містить код JSX, і відвідати http://localhost:3000/posts.

Потім ми хочемо створити динамічний маршрут для кожної публікації в блозі, щоб нам не доводилося вручну створювати нову сторінку кожного разу, коли ми хочемо опублікувати статтю на нашому веб-сайті. Для цього ми можемо створити новий файл у папці pages/posts/ pages/posts/[slug].js, де [slug] визначає змінну маршруту, яка може містити будь-яке значення, залежно від того, що користувач вводить у адресний рядок браузера. У цьому випадку ми створюємо маршрут, що містить змінну під назвою slug, яка може змінюватися для кожної публікації в блозі. Ми можемо експортувати просту функцію, яка повертає код JSX із цього файлу, а потім перейти до http://localhost:3000/posts/my-firstpost , http://localhost:3000/posts/foo-bar-baz або будь-якого іншого http://localhost:3000/posts/* маршрут. Незалежно від маршруту, який ви переглядаєте, він завжди відображатиме той самий код JSX.

Ми також можемо вкладати кілька динамічних маршрутів у папку pages/; припустімо, що ми хочемо, щоб структура нашої сторінки дописів була такою: /posts/[date]/[slug]. Ми можемо просто додати нову папку з назвою [date] у наш каталог pages/ і перемістити в неї файл slug.js:

```
pages/
  - index.js
  - contact-us.js
  - posts/
```

```
- index.js
- [date]/
  - [slug].js
```

Тепер ми можемо відвідати http://localhost:3000/posts/2021-01-01/my-firstpost і переглянути вміст JSX, який ми створили раніше. Знову ж таки, змінні [date] і [slug] можуть представляти все, що завгодно, тому можете експериментувати, викликаючи різні маршрути в браузері.

Досі ми завжди використовували змінні маршруту для візуалізації однієї сторінки, але ці змінні в основному призначені для створення високодинамічних сторінок із різним вмістом залежно від змінних маршруту, які ми використовуємо. Давайте подивимося, як рендерити різний вміст залежно від змінних у наступних розділах.

Використання змінних маршруту на наших сторінках

Змінні маршруту неймовірно корисні для створення дуже динамічного вмісту сторінки.

Візьмемо простий приклад: сторінку привітань. У проекті, використаному в попередньому розділі, давайте створимо такий файл: pages/greet/[name].js. Ми збираємося використовувати вбудовану функцію getServerSideProps Next.js, щоб динамічно отримати змінну [name] з URL-адреси та привітати користувача:

```
export async function getServerSideProps({ params }) {
  const { name } = params;
  return {
    props: {
    name
    }
  }
}
function Greet(props) {
  return (
    <h1> Hello, {props.name}! </h1>
  )
}
export default Greet;
```

Тепер відкрийте свій улюблений браузер і перейдіть до http://localhost:3000/greet/Mitch ; ви повинні побачити ***"Hello, Mitch!"*** повідомлення на екрані. Пам’ятайте, що ми використовуємо змінну імені, тож можете спробувати інші імена!

Важлива примітка

Використовуючи функції getServerSideProps і getStaticProps, пам’ятайте, що вони мають повертати об’єкт. Крім того, якщо ви хочете передати будь-який prop від однієї з цих двох функцій на свою сторінку, переконайтеся, що він переданий у властивість props об’єкта, що повертається.

Можливість отримати дані з URL-адреси є фундаментальною з багатьох причин. У попередньому прикладі коду ми створили просту сторінку привітань, але ми могли використати змінну [name] для інших цілей, наприклад отримати дані користувача з бази даних, щоб показати їхній профіль. Ми детальніше розглянемо вибірку даних у Розділі 4 «Організація кодової бази та отримання даних у Next.js».

Бувають випадки, коли вам потрібно отримати змінні маршруту з компонентів, а не зі сторінок. Next.js робить це легко завдяки хуку React, який ми побачимо в наступному розділі.

Використання змінних маршруту всередині компонентів

У попередньому розділі ми навчилися використовувати змінні маршруту на наших сторінках. Next.js не дозволяє нам використовувати функції getServerSideProps і getStaticProps поза нашими сторінками, тож як ми маємо використовувати їх в інших компонентах?

Next.js робить це легко завдяки хуку useRouter; ми можемо імпортувати його з файлу next/router:

```
import { useRouter } from 'next/router';
```

Він працює так само, як будь-який інший хук React (функція, яка дозволяє вам взаємодіяти зі станом React і життєвим циклом у функціональних компонентах), і ми можемо створити його екземпляр у будь-якому компоненті. Давайте переробимо попередню сторінку привітань наступним чином:

```
import { useRouter } from 'next/router';
function Greet() {
  const { query } = useRouter();
  return <h1>Hello {query.name}!</h1>;
}
export default Greet;
```

Як бачите, ми витягуємо параметр запиту з хука useRouter. Він містить як наші змінні маршруту (у цьому випадку він містить лише змінну name), так і проаналізовані параметри рядка запиту.

Ми можемо спостерігати, як Next.js передає як змінні маршруту, так і рядки запиту через хук useRouter, намагаючись додати будь-який параметр запиту до нашої URL-адреси та зареєструвати змінну запиту всередині нашого компонента:

```
import { useRouter } from 'next/router';
function Greet() {
  const { query } = useRouter();
  console.log(query);
  return <h1>Hello {query.name}!</h1>;
}
export default Greet;
```

Якщо ми зараз спробуємо викликати таку URL-адресу, http://localhost:3000/greet/Mitch?learning_nextjs=true , ми побачимо такий об'єкт, зареєстрований у нашому терміналі:

{learning_nextjs: "true", name: "Mitch"}

Важлива примітка

Next.js не видає жодних помилок, якщо ви намагаєтеся додати параметр запиту з тим самим ключем, що й ваша змінна маршрутизації. Ви можете легко спробувати це, зробивши виклик за такою URL-адресою: http://localhost:3000/greet/Mitch?name=Christine . Ви помітите, що Next.js надаватиме пріоритет вашій змінній route, тож ви побачите, що Hello, Mitch! відображається на сторінці.

Навігація на стороні клієнта

Як ми вже бачили, Next.js — це не лише рендеринг React на сервері. Він надає кілька способів оптимізації продуктивності вашого веб-сайту, і одна із цих оптимізацій полягає в тому, як він обробляє навігацію на стороні клієнта.

Насправді він підтримує стандартні теги HTML <a> для зв'язування сторінок, але також забезпечує більш оптимізований спосіб навігації між різними маршрутами: компонент Link.

Ми можемо імпортувати його як стандартний компонент React і використовувати для зв'язування різних сторінок або розділів нашого веб-сайту. Давайте розглянемо простий приклад:

```
import Link from 'next/link';
function Navbar() {
  return (
    <div>
      <Link href='/about'>Home</Link>
```

```
      <Link href='/about'>About</Link>
      <Link href='/about'>Contacts</Link>
    </div>
  );
}
export default Navbar;
```

За замовчуванням Next.js попередньо завантажить кожне окреме посилання, знайдене у вікні перегляду, тобто щойно ми натиснемо одне з посилань, браузер уже матиме всі дані, необхідні для відтворення сторінки.

Ви можете вимкнути цю функцію, передавши властивість preload={false} компоненту Link:

```
import Link from 'next/link';
function Navbar() {
  return (
    <div>
      <Link href='/about' preload={false}>Home</Link>
      <Link href='/about' preload={false}>About</Link>
      <Link href='/about' preload={false}>Contacts</Link>
    </div>
  );
}
export default Navbar;
```

Починаючи з Next.js 10, ми також можемо з легкістю зв’язувати сторінки з динамічними змінними маршруту.

Скажімо, ми хочемо зв’язати таку сторінку: /blog/[date]/[slug].js. У попередніх версіях Next.js нам потрібно було додати два різні властивості:

```
<Link href='/blog/[date]/[slug]'
  as='/blog/2021-01-01/happy-new-year'>
  Read post
</Link>
```

Атрибут href повідомляє Next.js, яку сторінку ми хочемо відобразити, а атрибут as вкаже, як ми хочемо відобразити її в адресному рядку браузера.

Завдяки вдосконаленням, представленим у Next.js 10, нам більше не потрібно використовувати атрибут as, оскільки атрибута href достатньо для налаштування сторінки, яку ми хочемо відобразити, та URL-адреси, що відображається в адресному рядку браузера. Наприклад, тепер ми можемо написати наші посилання так:

```
<Link href='/blog/2021-01-01/happy-new-year'> Read post </Link>
<Link href='/blog/2021-03-05/match-update'> Read post </Link>
<Link href='/blog/2021-04-23/i-love-nextjs'> Read post </Link>
```

Важлива примітка

Хоча застарілий метод зв'язування динамічних сторінок за допомогою компонента Link все ще працює в Next.js >10, найновіша версія фреймворку значно полегшує його роботу. Якщо у вас є певний досвід роботи з попередніми версіями Next.js або ви бажаєте оновити до версії >10, пам'ятайте про цю нову функцію, оскільки вона спростить розробку компонентів, зокрема динамічних посилань.

Якщо ми створюємо складні URL-адреси, ми також можемо передати об'єкт атрибуту href:

```
<Link
  href={{
    pathname: '/blog/[date]/[slug]'
    query: {
      date: '2020-01-01',
      slug: 'happy-new-year',
      foo: 'bar'
    }
  }}
/>
  Read post
</Link>
```

Коли користувач клацне це посилання, Next.js перенаправить браузер на таку URL-адресу: http://localhost:3000/blog/2020-01-01/happy-new-year?foo=bar.

Використання методу router.push

Існує ще один спосіб переходу між сторінками веб-сайту Next.js: за допомогою хука useRouter.

Давайте уявимо, що ми хочемо надати доступ до даної сторінки лише користувачам, які ввійшли в систему, і у нас уже є хук useAuth для цього. Ми можемо використовувати хук useRouter для динамічного перенаправлення користувача, якщо в цьому випадку він не ввійшов у систему:

```
import { useEffect } from 'react';
import { useRouter } from 'next/router';
import PrivateComponent from '../components/Private';
import useAuth from '../hooks/auth';
function MyPage() {
```

```
  const router = useRouter();
  const { loggedIn } = useAuth();
  useEffect(() => {
    if (!loggedIn) {
    router.push('/login')
    }
  }, [loggedIn]);
  return loggedIn
    ? <PrivateComponent />
    : null;
}
export default MyPage;
```

Як бачите, ми використовуємо хук useEffect для запуску коду лише на стороні клієнта. У такому випадку, якщо користувач не ввійшов у систему, ми використовуємо метод router.push, щоб перенаправити його на сторінку входу.

Як і з компонентом Link, ми можемо створювати складніші маршрути сторінок, передаючи об’єкт методу push:

```
router.push({
  pathname: '/blog/[date]/[slug]',
  query: {
    date: '2021-01-01',
    slug: 'happy-new-year',
    foo: 'bar'
  }
});
```

Після виклику функції router.push браузер буде перенаправлено на сторінку http://localhost:3000/blog/2020-01-01/happy-new-year?foo=bar.

Важлива примітка

Next.js не зможе попередньо отримати всі пов’язані сторінки, як це робиться з компонентом Link.

Використання методу router.push зручно, коли вам потрібно перенаправити користувача на стороні клієнта після виконання певної дії, але не рекомендується використовувати його як спосіб за замовчуванням для обробки навігації на стороні клієнта.

Наразі ми бачили, як Next.js обробляє навігацію через статичні та динамічні маршрути та як примусово переспрямовувати навігацію як на стороні клієнта, так і на стороні сервера програмно.

У наступному розділі ми розглянемо, як Next.js допомагає нам обслуговувати статичні ресурси та оптимізувати зображення на льоту для покращення продуктивності та показників SEO.

Обслуговування статичних ресурсів

Використовуючи термін статичний ресурс, ми маємо на увазі всі ці нединамічні файли, такі як зображення, шрифти, піктограми, скомпільовані файли CSS і JS.

Найпростіший спосіб обслуговувати ці активи — використовувати стандартну папку /public, яку надає Next.js. Насправді кожен файл у цій папці розглядатиметься та обслуговуватиметься як статичний ресурс.

Ми можемо це довести, створивши новий файл під назвою index.txt і помістивши його в папку /public:

```
echo "Hello, world!" >> ./public/index.txt
```

Якщо ми зараз спробуємо запустити сервер, коли ми перейдемо до http://localhost:3000/index.txt , ми побачимо текст Hello, world! в браузері.

У розділі 4 «Організація кодової бази та отримання даних у Next.js» ми докладніше розглянемо організацію загальнодоступної папки для обслуговування звичайних файлів CSS і JS, зображень, піктограм і всіх інших типів статичних файлів.

Обслуговувати статичні активи відносно легко. Однак певний тип файлу може критично вплинути на ефективність вашого веб-сайту (і SEO): файл зображення.

У більшості випадків розміщення неоптимізованих зображень погіршить роботу з користувачем, оскільки їхнє завантаження може зайняти деякий час, і як тільки вони завантажаться, вони перемістять частину макета після візуалізації, що може спричинити багато проблем з точки зору UX. Коли це відбувається, ми говоримо про сукупний зсув макета (**Cumulative Layout Shift, CLS**). Ось просте представлення того, як працює CLS:

Рис. 3.1 Відображення того, як працює CLS

У першій вкладці браузера зображення ще не завантажено, тому дві текстові області виглядають досить близько одна до одної. Після завантаження зображення друга текстова область зсувається вниз. Якби користувач читав другу область тексту, він легко пропустив би позначку.

Важлива примітка

Якщо ви хочете дізнатися більше про CLS, я б порекомендував таку статтю: https://web.dev/cls .

Звичайно, Next.js дозволяє легко уникнути CLS, і це робиться за допомогою нового вбудованого компонента Image. Ми розглянемо це в наступному розділі.

Автоматична оптимізація зображення Next.js

Починаючи з Next.js 10, фреймворк представив новий корисний компонент Image і автоматичну оптимізацію зображення.

Перш ніж Next.js представив ці дві нові функції, нам довелося оптимізувати кожне зображення за допомогою зовнішнього інструменту, а потім записати складну властивість srcset для кожного HTML-тегу <img>, щоб установити адаптивні зображення для різних розмірів екрана.

Дійсно, автоматична оптимізація зображень подбає про надання ваших зображень у сучасних форматах (наприклад, **WebP**) у всіх тих браузерах, які її підтримують. Але він також зможе повернутися до старих форматів зображень, таких як png або jpg, якщо браузер, який ви використовуєте, не підтримує їх. Він також змінює розмір ваших зображень, щоб уникнути надання клієнту важких зображень, оскільки це негативно вплине на швидкість завантаження ресурсу.

Варто пам’ятати про те, що автоматична оптимізація зображень працює за вимогою, оскільки вона оптимізує, змінює розмір і відображає зображення лише тоді, коли це запитує браузер. Це важливо, оскільки він працюватиме з будь-яким зовнішнім джерелом даних (будь-якою CMS або службою зображень, як-от Unsplash або Pexels), і не сповільнить етап створення.

Ми можемо спробувати цю функцію на нашій локальній машині за кілька хвилин, щоб особисто побачити, як вона працює. Скажімо, ми хочемо подати таке зображення:

Рис. 3.2 - Зображення Łukasz Rawa на Unsplash (https://unsplash.com/@lukasz_rawa)

Використовуючи стандартні теги HTML, ми могли б просто зробити наступне:

```
<img
  src='https://images.unsplash.com/photo-1605460375648-
  278bcbd579a6'
  alt='A beautiful English Setter'
/>
```

Однак ми також можемо захотіти використовувати властивість srcset для адаптивних зображень, тому нам насправді потрібно буде оптимізувати зображення для різних роздільних здатностей екрана, що передбачає деякі додаткові кроки для обслуговування наших ресурсів.

Next.js робить це дуже легким, просто налаштувавши файл next.config.js і використовуючи компонент Image. Ми щойно сказали, що хочемо обслуговувати зображення, які надходять із Unsplash, тому давайте додамо цю назву хоста служби до нашого файлу next.config.js у властивості images:

```
module.exports = {
  images: {
    domains: ['images.unsplash.com']
```

```
  }
}
```

Таким чином, щоразу, коли ми використовуємо зображення, що надходить із цього імені хоста всередині компонента Image, Next.js автоматично оптимізує його для нас.

Тепер давайте спробуємо імпортувати це зображення на сторінку:

```
import Image from 'next/image';
function IndexPage() {
  return (
    <div>
      <Image
        src='https://images.unsplash.com/photo-
          1605460375648-278bcbd579a6'
        width={500}
        height={200}
        alt='A beautiful English Setter'
      />
    </div>
  );
}
export default IndexPage;
```

Відкривши веб-переглядач, ви помітите, що зображення розтягнуто, щоб підходити до ширини та висоти, указаних у вашому компоненті Image.

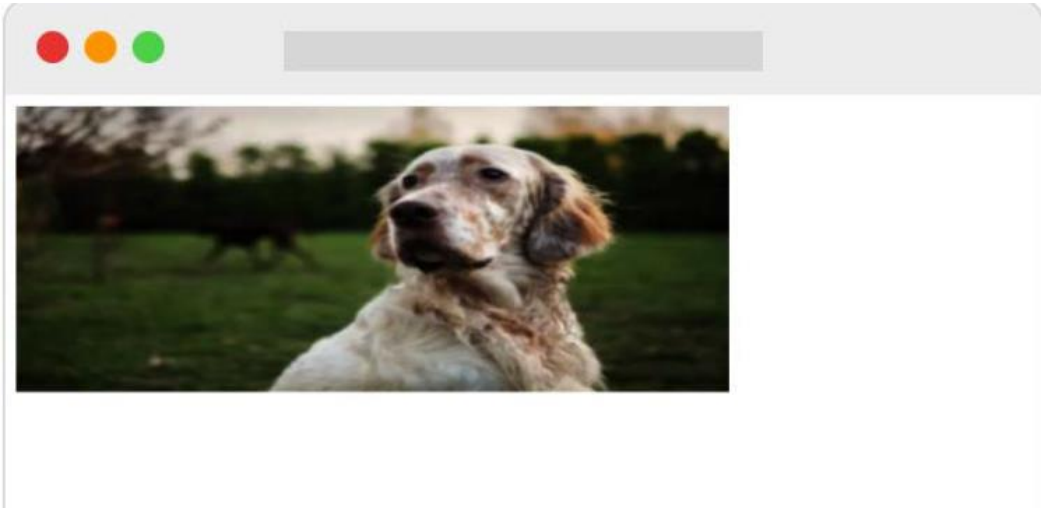

Рис. 3.3 - Представлення компонента Image, який ми щойно створили

Ми можемо обрізати наше зображення відповідно до потрібних розмірів за допомогою додаткової опори макета. Він приймає чотири різні значення: fixed, intrinsic, responsive, та fill. Давайте розглянемо їх докладніше:

• fixed працює так само, як HTML-тег img. Якщо ми змінимо розмір вікна перегляду, він збереже той самий розмір, що означає, що воно не забезпечить адаптивне зображення для менших (або більших) екранів.

• responsive працює протилежно фіксованому; коли ми змінюємо розмір вікна перегляду, воно буде показувати різні оптимізовані зображення для розміру нашого екрана.

• intrinsic знаходиться посередині між фіксованим і чуйним; він обслуговуватиме зображення різних розмірів, коли ми змінюємо розмір вікна перегляду, але найбільше зображення залишатиметься недоторканим на великих екранах.

• fill розтягне зображення відповідно до ширини та висоти його батьківського елемента; однак ми не можемо використовувати fill поряд із атрибутами ширини та висоти. Ви можете використовувати fill або ширину та висоту).

Тож тепер, якщо ми хочемо виправити зображення нашого англійського сетера, щоб воно належним чином відображалося на нашому екрані, ми можемо змінити наш компонент Image наступним чином:

```
import Image from 'next/image';
function IndexPage() {
  return (
    <div>
      <div
        style={{ width: 500, height: 200, position:
          'relative' }}
      >
        <Image
          src='https://images.unsplash.com/photo-
            1605460375648-278bcbd579a6'
          layout='fill'
          objectFit='cover'
          alt='A beautiful English Setter'
        />
      </div>
    </div>
  );
}
export default IndexPage;
```

Як бачите, ми обернули компонент Image у div фіксованого розміру, а для властивості CSS position було встановлено значення relative. Ми також

видалили атрибути ширини та висоти з нашого компонента Image, оскільки він розтягуватиметься відповідно до розмірів батьківського div.

Ми також додали властивість cover для objectFit, щоб воно обрізало зображення відповідно до розміру його батьківського div, і це остаточний результат.

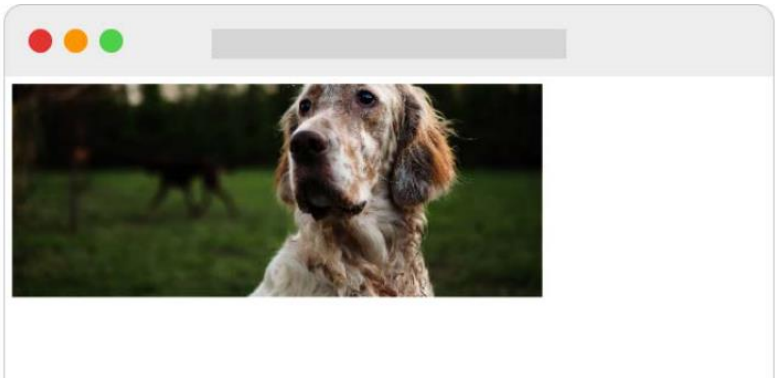

Рис. 3.4 – Представлення компонента Image з атрибутом layout, встановленим на "fill"

Якщо ми тепер спробуємо перевірити отриманий HTML у браузері, ми побачимо, що компонент Image згенерував багато різних розмірів зображень, які будуть обслуговуватися за допомогою властивості srcset стандартного тегу HTML img:

```
<div style="..."
<img alt="A beautiful English Setter"src="/_next/
image?url=https%3A%2F%2Fimages.unsplash.com%2Fphoto-
1605460375648-278bcbd579a6&w=3840&q=75" decoding="async"
sizes="100vw" srcset="/_next/image?url=https%3A%2F%2Fimages.
unsplash.com%2Fphoto-1605460375648-278bcbd579a6&w=640&q=75
640w, /_next/image?url=https%3A%2F%2Fimages.unsplash.
com%2Fphoto-1605460375648-278bcbd579a6&w=750&q=75 750w, /_
next/image?url=https%3A%2F%2Fimages.unsplash.com%2Fphoto-
1605460375648-278bcbd579a6&w=828&q=75 828w, /_next/
image?url=https%3A%2F%2Fimages.unsplash.com%2Fphoto-
1605460375648-278bcbd579a6&w=1080&q=75 1080w, /_next/
image?url=https%3A%2F%2Fimages.unsplash.com%2Fphoto-
1605460375648-278bcbd579a6&w=1200&q=75 1200w, /_next/
image?url=https%3A%2F%2Fimages.unsplash.com%2Fphoto-
1605460375648-278bcbd579a6&w=1920&q=75 1920w, /_next/
image?url=https%3A%2F%2Fimages.unsplash.com%2Fphoto-
1605460375648-278bcbd579a6&w=2048&q=75 2048w, /_next/
image?url=https%3A%2F%2Fimages.unsplash.com%2Fphoto-
1605460375648-278bcbd579a6&w=3840&q=75 3840w" style="..."
</div>
```

І останнє, про що варто згадати, це те, що якщо ми перевіримо формат зображення в Google Chrome або Firefox, ми побачимо, що воно обслуговувалося як WebP, навіть якщо оригінальне зображення, надане з

Unsplash, було jpeg. Якщо ми зараз спробуємо відобразити ту саму сторінку в iOS за допомогою Safari, Next.js обслуговуватиме оригінальний формат jpeg, оскільки (на момент написання) цей браузер iOS ще не підтримує формат WebP.

Як було сказано на початку цього розділу, Next.js запускає автоматичну оптимізацію зображення на вимогу, тобто якщо дане зображення ніколи не запитується, воно ніколи не буде оптимізовано.

Весь етап оптимізації відбувається на сервері, де запущено Next.js. Якщо ви використовуєте веб-програму, яка містить безліч зображень, це може вплинути на продуктивність вашого сервера. У наступному розділі ми побачимо, як делегувати етап оптимізації зовнішнім службам.

Запуск автоматичної оптимізації зображення на зовнішніх службах

За замовчуванням автоматична оптимізація зображень виконується на тому ж сервері, що й Next.js. Звичайно, якщо ваш веб-сайт працює на невеликому сервері з низькими ресурсами, це потенційно може вплинути на його продуктивність. З цієї причини Next.js дозволяє запускати автоматичну оптимізацію зображень у зовнішніх службах, встановлюючи параметр завантажувача у вашому next. файл config.js:

. . . . .

Обробка метаданих

Правильна обробка метаданих є важливою частиною сучасної веб-розробки. Щоб було просто, давайте подумаємо, коли ми ділимося посиланням у Facebook чи Twitter. Якщо ми поділимося веб-сайтом React (https://reactjs.org) у Facebook, у нашій публікації з’явиться така картка:

Рис. 3.5 - Дані Open Graph https://reactjs.org

Щоб знати, які дані мають відображатися всередині картки, Facebook використовує протокол Open Graph (https://ogp.me). Щоб надати цю інформацію будь-якій соціальній мережі чи веб-сайту, нам потрібно додати деякі метадані на наші сторінки.

Поки що ми ще не говорили про те, як динамічно встановлювати дані відкритого графіка, заголовки HTML або метатеги HTML. Хоча технічно веб-сайт міг би працювати навіть без цих даних, пошукові системи покарали б ваші сторінки, оскільки вони пропускали б важливу інформацію. Також це може негативно вплинути на взаємодію з користувачем, оскільки ці метатеги допоможуть веб-переглядачу створити оптимізовану роботу для наших користувачів.

Знову ж таки, Next.js пропонує чудовий спосіб вирішення цих проблем: вбудований компонент Head. Дійсно, цей компонент дозволяє нам оновлювати розділ <head> нашої HTML-сторінки з будь-якого компонента, тобто ми можемо динамічно змінювати, додавати або видаляти будь-які метадані, посилання або сценарій під час виконання залежно від навігації нашого користувача.

Ми можемо почати з однієї з найпоширеніших динамічних частин наших метаданих: тегу HTML <title>. Давайте налаштуємо новий проект Next.js, а потім створимо дві нові сторінки.

Перша сторінка, яку ми створимо, це index.js:

. . . . . p. 52

Групування спільних метатегів

На даний момент ми можемо додати багато інших метатегів до нашого веб-сайту, щоб покращити його ефективність SEO. Проблема полягає в тому, що ми легко можемо створити величезні компоненти сторінки, які містять в основному однакові теги. З цієї причини прийнято створювати один або кілька компонентів (залежно від ваших потреб) для обробки більшості поширених метатегів head.

Скажімо, ми хочемо додати розділ блогу на наш сайт. Можливо, ми захочемо додати підтримку даних відкритих графіків, карток Twitter та інших метаданих для наших публікацій у блозі, щоб ми могли легко групувати всі ці загальні дані в компоненті PostHead.

Давайте створимо новий файл, components/PostHead.js, і додамо такий сценарій:

. . . . . p.57

Налаштування сторінок _app.js і _document.js

Існують певні випадки, коли вам потрібно контролювати ініціалізацію сторінки, щоб кожного разу, коли ми візуалізуємо сторінку, Next.js потрібно було виконувати певні операції, перш ніж надсилати кінцевий HTML клієнту. Для цього фреймворк дозволяє нам створити два нових файли під назвою _app.js і _document.js у нашому каталозі pages/

. . . . p. 61

Частина 2. Практичний Next.js

У цій частині ми почнемо писати невеликі програми Next.js, зосереджуючись на основній темі кожного розділу. Ми побачимо, як приймати правильні рішення під час прийняття фреймворків інтерфейсу користувача, методів стилізації, стратегій тестування тощо.

Цей розділ складається з наступних розділів:

• Розділ 4, Організація кодової бази та отримання даних у Next.js

• Розділ 5, Керування локальними та глобальними станами в Next.js

• Розділ 6, CSS і вбудовані методи стилізації

• Розділ 7, Використання UI Frameworks

• Розділ 8, Використання спеціального сервера

• Розділ 9, Тестування Next.js

• Розділ 10, Робота з пошуковою системою пошукових систем і керування ефективністю

• Розділ 11, Різні платформи розгортання

Розділ 4. Організація кодової бази та отримання даних у Next.js

Next.js спочатку став популярним завдяки своїй здатності полегшувати візуалізацію сторінок React на сервері, а не лише на клієнті. Однак для

відтворення конкретних компонентів нам часто потрібні деякі дані, що надходять із зовнішніх джерел, таких як API та бази даних.

У цій главі ми спочатку побачимо, як організувати нашу структуру папок, оскільки це буде визначальним фактором для збереження акуратності потоку даних Next.js під час керування станом програми (як ми побачимо в главі 5, Керування локальними та глобальними станами в Next.js), а потім ми побачимо, як інтегрувати зовнішні API REST і GraphQL як на стороні клієнта, так і на стороні сервера.

У міру зростання нашої програми її складність неминуче зростатиме, і ми повинні бути готові до цього ще на етапі завантаження проекту. Щойно ми запровадимо нові функції, нам потрібно буде додати нові компоненти, утиліти, стилі та сторінки. З цієї причини ми детальніше розглянемо організацію наших компонентів на основі принципів атомарного дизайну, службових функцій, стилів і того, як зробити вашу кодову базу готовою до швидкої та акуратної обробки стану програми.

Ми детально розглянемо такі теми:

• Організація наших компонентів за принципом атомарного дизайну

• Організація наших utility функцій

• Охайна організація статичних ресурсів

• Вступ до організації файлів стилів

• Що таке файли lib і як їх упорядкувати

• Використання REST API лише на стороні сервера

• Використання REST API лише на стороні клієнта

• Налаштування Apollo на використання GraphQL API як на клієнті, так і на сервері

До кінця цього розділу ви знатимете, як організувати свою кодову базу, дотримуючись принципів атомарного проектування ваших компонентів, і як логічно розділити різні службові файли. Ви також дізнаєтеся, як використовувати API REST і GraphQL.

Технічні вимоги

Щоб запустити приклади коду в цьому розділі, вам потрібно інсталювати Node.js і npm на вашій локальній машині. Якщо ви віддаєте перевагу, ви можете використовувати онлайн-IDE, наприклад https://repl.it або https://codesandbox.io , оскільки вони обидва підтримують Next.js і вам не потрібно встановлювати будь-яку залежність на вашому комп’ютері.

Ви можете знайти базу коду для цієї глави на GitHub: https://github.com/PacktPublishing/Real-World-Next.js .

Організація структури папок

Охайна та зрозуміла організація структури папок вашого нового проекту надзвичайно важлива з точки зору збереження бази коду масштабованою та зручною для обслуговування.

Як ми вже бачили, Next.js змушує вас розміщувати деякі файли та папки в певних місцях бази коду (подумайте про файли _app.js і _documents.js, каталоги pages/ і public/ тощо). але це також надає можливість налаштувати їх розміщення у сховищі вашого проекту.

Ми це вже бачили, але давайте коротко нагадаємо структуру папок Next.js за замовчуванням:

```
next-js-app
  - node_modules/
  - package.json
  - pages/
  - public/
  - styles/
```

Читаючи зверху вниз, коли ми створюємо новий додаток Next.js за допомогою create-nextapp, ми отримуємо такі папки:

• node_modules/: папка за замовчуванням для залежностей проекту Node.js

• pages/: каталог, де ми розміщуємо наші сторінки та будуємо систему маршрутизації для нашої веб-програми

• public/: каталог, куди ми розміщуємо файли, які будуть обслуговуватися як статичні ресурси (скомпільовані файли CSS і JavaScript, зображення та піктограми)

• styles/: каталог, де ми розміщуємо наші модулі стилів, незалежно від їх формату (CSS, SASS, LESS)

Звідси ми можемо почати налаштовувати структуру нашого сховища, щоб полегшити навігацію. Перше, що потрібно знати, це те, що Next.js дозволяє переміщувати наш каталог pages/ всередину папки src/. Ми також можемо перемістити всі інші каталоги (крім public/one і node_modules, звісно) у src/, роблячи наш кореневий каталог трохи охайнішим.

Важлива примітка

Пам’ятайте, що якщо у вашому проекті є каталоги pages/ і src/pages/, Next.js ігноруватиме src/pages/, оскільки каталог pages/ кореневого рівня має пріоритет.

У наступному розділі ми розглянемо деякі популярні угоди щодо організації всієї бази коду, починаючи з компонентів React.

Організація компонентів

Тепер давайте розглянемо приклад реальної структури папок, включно з деякими ресурсами стилю (розділ 6, CSS і вбудовані методи стилю) і тестові файли (розділ 9, Тестування Next.js).

Наразі ми обговоримо лише структуру папок, яка може допомогти нам легко писати та знаходити конфігураційні файли, компоненти, тести та стилі. Ми розглянемо технології, процитовані раніше, у відповідних розділах.

У нас є різні способи налаштування структури папок. Ми можемо почати з розділення компонентів на три різні категорії, а потім поміщення стилів і тестів в одну папку для кожного компонента.

Для цього створіть нову папку components/ в кореневому каталозі. Потім, переміщаючись всередині нього, створіть такі папки:

```
mkdir components && cd components
mkdir atoms
mkdir molecules
mkdir organisms
mkdir templates
```

Як ви могли помітити, ми дотримуємося принципу атомарного дизайну, де ми хочемо розділити наші компоненти на різні рівні, щоб краще організувати нашу кодову базу. Це лише популярна угода, і ви можете використовувати будь-який інший підхід до організації свого коду.

Ми розділимо наші компоненти на чотири категорії:

• *atoms*: це найпростіші компоненти, які ми коли-небудь запишемо в нашу кодову базу. Іноді вони діють як обгортка для стандартних елементів HTML, таких як button, input і p, але ми також можемо додавати animations, кольорові палітри тощо до цієї категорії компонентів.

• *molecules*: це невелика група атомів, об’єднаних для створення трохи складніших структур з мінімальною корисністю. Вхідний input атом і атом label разом можуть бути прямим прикладом того, що таке молекула.

• *organisms*: молекули й атоми об'єднуються, створюючи складні структури, такі як реєстраційна форма, нижній колонтитул і карусель.

• *templates*: ми можемо розглядати шаблони як скелет наших сторінок. Тут ми вирішуємо, куди помістити організми, атоми та молекули, щоб створити остаточну сторінку, яку переглядатиме користувач.

Якщо вам цікаво дізнатися більше про атомарний дизайн, ось гарна стаття, де це докладно пояснюється: https://bradfrost.com/blog/post/atomic-web-design .

Тепер давайте уявимо, що ми хочемо створити компонент Button. Коли ми створюємо новий компонент, нам часто потрібні принаймні три різні файли: сам компонент, його стиль і тестовий файл. Ми можемо створити ці файли, перемістивши компоненти/атоми/, а потім створивши нову папку під назвою Button/. Після створення цієї папки можна переходити до створення файлів компонентів:

```
cd components/atoms/Button
touch index.js
touch button.test.js
touch button.styled.js # or style.module.css
```

Організація наших компонентів таким чином дуже допоможе нам, коли нам потрібно шукати, оновлювати чи виправляти певний компонент. Скажімо, ми помітили помилку у виробництві, яка стосується нашого компонента Button. Ми можемо легко знайти компонент у нашій кодовій базі, знайти його файли тестування та стилю та виправити їх.

Звичайно, дотримання принципу атомарного проектування не є обов'язковим, але я особисто рекомендую його, оскільки це допомагає підтримувати структуру проекту охайною та легкою для підтримки з часом.

Організація службових функцій

Є певні файли, які не експортують жодного компонента; це просто модульні сценарії, які використовуються для різних цілей. Ми тут говоримо про службові скрипти.

Давайте уявимо, що у нас є кілька компонентів, метою яких є перевірити, чи минула певна година доби для відображення певної інформації. Немає сенсу писати ту саму функцію всередині кожного компонента. Таким чином, ми можемо написати загальну службову функцію, а потім імпортувати її в кожен компонент, який потребує такої функції.

Ми можемо помістити всі наші службові функції в папку utility/, а потім розділити наші службові програми на різні файли відповідно до їх призначення. Наприклад, припустимо, що нам потрібні чотири службові функції: перша виконуватиме обчислення на основі поточного часу, друга виконуватиме певні операції в localStorage, третя працюватиме з JWT (JSON Web Token), а остання допоможе нам писати кращі журнали для наших програм.

Ми можемо продовжити, створивши чотири різні файли всередині каталогу utilities/:

```
cd utilities/
touch time.js
touch localStorage.js
touch jwt.js
touch logs.js
```

Тепер, коли ми створили наші файли, ми можемо продовжити створення відповідних тестових файлів:

```
touch time.test.js
touch localStorage.test.js
touch jwt.test.js
touch logs.test.js
```

На даний момент наші утиліти згруповані за їх сферою дії, що дозволяє легко запам’ятати, з якого файлу нам потрібно імпортувати певну функцію під час процесу розробки.

Можуть існувати й інші підходи до організації службових файлів. Ви можете створити папку для кожного файлу утиліти, щоб ви могли розміщувати в ній тести, стилі та інші речі, тим самим роблячи вашу базу коду ще більш упорядкованою. Це повністю залежить від вас!

Організація статичних активів

Як було показано в попередньому розділі, Next.js полегшує обслуговування статичних файлів, оскільки вам потрібно лише помістити їх у папку public/, а фреймворк зробить все інше.

З цього моменту нам потрібно визначити, які статичні файли нам потрібно обслуговувати з нашої програми Next.js.

На стандартному веб-сайті ми можемо захотіти обслуговувати принаймні такі статичні ресурси:

• Зображення

• Зкомпільовані файли JavaScript

• Скомпільовані файли CSS

• Icons (зокрема значки favicon і веб-додатків)

• manifest.json, robot.txt та інші статичні файли

Переходячи всередину нашої папки public/, ми можемо створити новий каталог під назвою assets/:

```
cd public && mkdir assets
```

І в цьому щойно створеному каталозі ми створимо нову папку для кожного типу статичного ресурсу:

```
cd assets
mkdir js
mkdir css
mkdir icons
mkdir images
```

Ми розмістимо скомпільовані файли JavaScript постачальника в каталозі js/ і зробимо те ж саме зі скомпільованими файлами CSS постачальника (звичайно, у каталозі css/). Під час запуску нашого сервера Next.js ми зможемо отримати доступ до загальнодоступних файлів за адресами http://localhost:3000/assets/js/<any-js-file> і http://localhost:3000/assets/css/<будь-який-css-файл> . Ми також зможемо отримати доступ до кожного загальнодоступного зображення, звернувшись за такою URL-адресою, http://localhost:3000/assets/image/<any-image-file> , але я пропоную вам обслуговувати ці типи активів за допомогою вбудованого у компоненті Image, як показано в попередньому розділі.

Каталог icons/ в основному використовуватиметься для обслуговування піктограм маніфесту веб-додатків. Маніфест веб-програми — це файл JSON, який містить деяку корисну інформацію про прогресивну веб-програму, яку ви створюєте, наприклад назву програми та піктограми, які слід використовувати під час її встановлення на мобільному пристрої. Ви можете дізнатися більше про маніфест веб-програми на сторінці https://web.dev/add-manifest .

Ми можемо легко створити цей файл маніфесту, увійшовши в папку public/ і додавши новий файл під назвою manifest.json:

```
cd public/ && touch manifest.json
```

На цьому етапі ми можемо заповнити файл JSON базовою інформацією. Візьмемо для прикладу наступний JSON:

```
{
  "name": "My Next.js App",
  "short_name": "Next.js App",
  "description": "A test app made with next.js",
  "background_color": "#a600ff",
  "display": "standalone",
  "theme_color": "#a600ff",
  "icons": [
    {
      "src": "/assets/icons/icon-192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "/assets/icons/icon-512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ]
}
```

Ми можемо включити цей файл за допомогою метатегу HTML, як показано в Розділі 3, Основи Next.js і вбудовані компоненти:

```
<link rel="manifest" href="/manifest.json">
```

Таким чином, користувачі, які переглядають вашу програму Next.js із мобільного пристрою, зможуть установити її на свої смартфони чи планшети.

Організація стилів

Організація стилю дійсно може залежати від стека, який ви хочете використовувати для стилізації програми Next.js.

Починаючи з фреймворків CSSinJS, таких як Emotion, styled-components, JSS і подібних, один поширений підхід полягає у створенні конкретного файлу стилю для кожного компонента; таким чином нам буде легше знайти певний стиль компонента в нашій базі коду, коли нам потрібно буде внести деякі зміни.

Однак, незважаючи на те, що розділення файлів стилів залежно від відповідних компонентів може допомогти нам упорядкувати нашу базу коду, нам може знадобитися створити деякі загальні стилі або службові файли, такі як колірні палітри, теми та медіа-запити.

У такому випадку може бути корисним повторно використати каталог styles/ за замовчуванням, який постачається разом із інсталяцією Next.js за замовчуванням. Ми можемо помістити наші загальні стилі в цю папку та імпортувати їх в інші файли стилів лише тоді, коли вони нам потрібні.

Тим не менш, насправді не існує стандартного способу організації файлів стилів. Ми докладніше розглянемо ці файли в Розділі 6, CSS і вбудовані методи стилізації, і Розділі 7, Використання UI Frameworks.

Lib файли

Говорячи про файли lib, ми маємо на увазі сценарії, які явно загортають бібліотеки сторонніх розробників у файли lib. Хоча сценарії службових програм є дуже загальними та можуть використовуватися багатьма різними компонентами та бібліотеками, файли lib є специфічними для певної бібліотеки. Щоб зробити концепцію більш зрозумілою, давайте на мить поговоримо про GraphQL.

Як ми побачимо в останньому розділі цієї глави Отримання даних, нам потрібно буде ініціалізувати клієнт GraphQL, зберегти деякі запити та мутації GraphQL локально тощо. Щоб зробити ці сценарії більш модульними, ми будемо зберігати їх у новій папці під назвою graphql/, яка знаходиться всередині каталогу lib/ у корені нашого проекту.

Якщо ми спробуємо візуалізувати структуру папок для попереднього прикладу, ми отримаємо таку схему:

```
next-js-app
  - lib/
    - graphql/
      - index.js
      - queries/
        - query1.js
        - query2.js
      - mutations/
        - mutation1.js
        - mutation2.js
```

Інші сценарії lib можуть включати всі ці файли, які підключаються до Redis, RabbitMQ тощо та надсилають запити до них, або функції, специфічні для будь-якої зовнішньої бібліотеки.

Хоча організована структура папок здається поза контекстом, коли йдеться про потік даних Next.js, вона насправді може допомогти нам керувати

станом програми, як ми побачимо в Розділі 5, Керування локальними та глобальними станами у Next.js

Але якщо говорити про стан програми, ми хочемо, щоб наші компоненти були динамічними більшу частину часу, тобто вони могли відтворювати вміст і поводитися по-різному залежно від глобального стану програми або даних, що надходять із зовнішніх служб. Фактично, у багатьох випадках нам потрібно викликати зовнішні API, щоб динамічно отримувати вміст веб-додатків. У наступному розділі ми побачимо, як отримати дані на стороні клієнта та сервера за допомогою клієнтів GraphQL і REST.

Отримання даних

Як було показано в попередніх розділах, Next.js дозволяє нам отримувати дані як на стороні клієнта, так і на стороні сервера. Отримання даних на стороні сервера може відбуватися в два різні моменти: під час створення (з використанням getStaticProps для статичних сторінок) і під час виконання (з використанням getServerSideProps для сторінок, відтворених на стороні сервера).

Дані можуть надходити з кількох ресурсів: баз даних, пошукових систем, зовнішніх API, файлових систем та багатьох інших джерел. Навіть якщо для Next.js технічно можливо отримати доступ до бази даних і запитувати певні дані, особисто я б не рекомендував такий підхід, оскільки Next.js має дбати лише про інтерфейс нашої програми.

Давайте візьмемо приклад: ми створюємо блог і хочемо відобразити сторінку автора, де вказано його ім’я, посаду та біографію. У цьому прикладі дані зберігаються в базі даних MySQL, і ми можемо легко отримати до них доступ за допомогою будь-якого клієнта MySQL для Node.js

Незважаючи на те, що доступ до цих даних із Next.js може бути відносно простим, це зробить наш додаток менш безпечним. Зловмисний користувач потенційно може знайти спосіб використовувати наші дані за допомогою невідомої вразливості фреймворку, впровадження шкідливого коду та використання інших методів для викрадення наших даних.

З цієї причини я наполегливо пропоную делегувати підключення до бази даних і запити до зовнішніх систем (іншими словами, CMS, таких як WordPress, Strapi та Contentful) або серверних фреймворків (іншими словами, Spring, Laravel і Ruby on Rails), які переконаються, що дані надходять із надійного джерела, очистять введені користувачем дані, виявляючи

потенційно шкідливий код, і встановлять безпечне з'єднання між вашою програмою Next.js та її API.

У наступних розділах ми побачимо, як інтегрувати REST і GraphQL API як на стороні клієнта, так і на стороні сервера.

Отримання даних на стороні сервера

Як ми вже бачили, Next.js дозволяє нам отримувати дані на стороні сервера за допомогою вбудованих функцій getStaticProps і getServerSideProps.

Враховуючи, що Node.js не підтримує API отримання JavaScript, як це роблять браузери, у нас є два варіанти для виконання HTTP-запитів на сервері:

1. Використання вбудованої http-бібліотеки Node.js: ми можемо використовувати цей модуль, не встановлюючи будь-яких зовнішніх залежностей, але навіть якщо його API дійсно прості та якісно створені, він потребує трохи додаткової роботи порівняно з третьою системою. сторонні HTTP-клієнти.
2. Використання бібліотек HTTP-клієнтів: для Next.js є кілька чудових HTTP-клієнтів, завдяки яким надсилати HTTP-запити із сервера дуже просто. Популярні бібліотеки включають isomorphic-unfetch (це рендерить API вибірки JavaScript, доступний на Node.js), Undici (офіційний клієнт Node.js HTTP 1.1) і Axios (дуже популярний клієнт HTTP, який працює як на клієнті, так і на сервері з тим самим API).

У наступному розділі ми будемо використовувати Axios для виконання запитів REST, оскільки це, ймовірно, один із найбільш часто використовуваних HTTP-клієнтів як для клієнта, так і для сервера (з ~ 17 000 000 завантажень на тиждень на npm), і існує велика ймовірність, що ви рано чи пізно скористається ним.

Використання REST API на стороні сервера

Обговорюючи інтеграцію REST API, ми повинні розділити їх на загальнодоступні та приватні API. Загальнодоступні доступні будь-кому без будь-якого дозволу, а приватні завжди потребують авторизації для повернення деяких даних.

Крім того, метод авторизації не завжди однаковий (і для різних API можуть знадобитися різні методи авторизації), оскільки це залежить від того, хто розробив API та вибір, який вони зробили. Наприклад, якщо ви хочете використовувати будь-який із API Google, вам потрібно буде запустити процес

під назвою OAuth 2.0, який є галузевим стандартом для захисту API під час автентифікації користувача. Ви можете прочитати більше про OAuth 2.0 в офіційній документації Google: https://developers.google.com/identity/protocols/oauth2 .

Інші API, як-от Pexels API (https://www.pexels.com/api/documentation), дозволяють використовувати їхній вміст за допомогою ключа API, який, по суті, є маркером авторизації, який вам потрібно надіслати у своєму запиті.

Можуть бути й інші способи авторизації ваших запитів, але Oauth 2.0, JWT і ключ API є найпоширенішими способами, з якими ви, ймовірно, зіткнетеся під час розробки програм Next.js.

Якщо після прочитання цього розділу ви захочете поекспериментувати з різними API та методами авторизації, ось чудовий репозиторій GitHub, який містить список безкоштовних REST API: https://github.com/public-apis/public-apis.

Наразі ми будемо використовувати спеціальний API, спеціально створений для цієї книги: https://api.realworldnextjs.com (або, якщо вам більше подобається: https://api.rwnjs.com). Ми можемо почати зі створення нового проекту Next.js:

```
npx create-next-app ssr-rest-api
```

Після запуску сценарію ініціалізації Next.js ми можемо додати axios як залежність, оскільки ми будемо використовувати його як HTTP-клієнт для виконання запитів REST:

```
cd ssr-rest-api
yarn add axios
```

На цьому етапі ми можемо легко редагувати стандартну сторінку індексу Next.js. Тут ми перерахуємо деяких користувачів, які використовують загальнодоступний API, показуючи лише їхні імена користувачів та особисті ідентифікатори. Після того, як ми натиснемо одне з імен користувачів, ми будемо перенаправлені на сторінку з детальною інформацією, щоб побачити більше особистих даних наших користувачів.

Почнемо зі створення макета сторінки pages/index.js:

```
import { useEffect } from 'react';
import Link from 'next/link';
export async function getServerSideProps() {
   // Here we will make the REST request to our APIs
}
function HomePage({ users }) {
```

```
  return (
    <ul>
      {
        users.map((user) =>
          <li key={user.id}>
            <Link
              href={`/users/${user.username}`}
              passHref
            >
              <a> {user.username} </a>
            </Link>
          </li>
        )
      }
    </ul>
  )
}
export default HomePage;
```

Якщо ми спробуємо запустити попередній код, ми побачимо помилку, оскільки ми ще не маємо даних наших користувачів. Нам потрібно викликати REST API із вбудованого getServerSideProps і передати результат запиту як проп компоненту HomePage:

```
import { useEffect } from 'react';
import Link from 'next/link';
import axios from 'axios';
export async function getServerSideProps() {
  const usersReq =
    await axios.get('https://api.rwnjs.com/04/users')
  return {
    props: {
      users: usersReq.data
    }
  }
}
function HomePage({ users }) {
  return (
    <ul>
      {
        users.map((user) =>
          <li key={user.id}>
            <Link
              href={`/users/${user.username}`}
               passHref
            >
              <a> {user.username} </a>
```

```
            </Link>
          </li>
      )
    }

      </ul>
    )
}
export default HomePage;
```

Тепер запустіть сервер, а потім перейдіть до http://localhost:3000 . У браузері має з'явитися такий список користувачів:

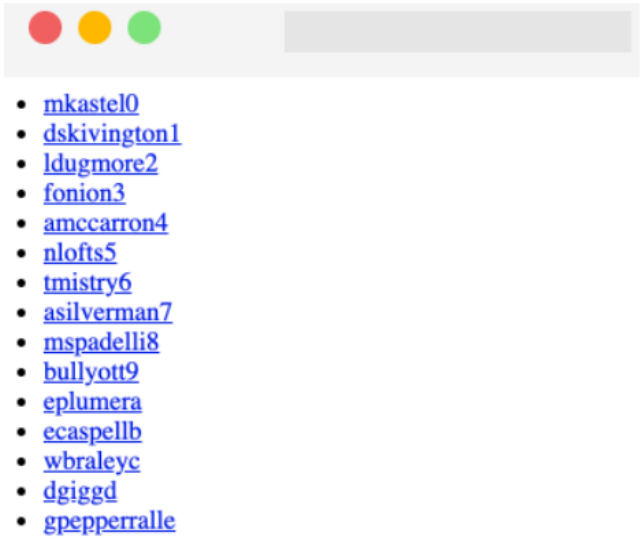

Рис. 4.1 - Результат API, відтворений у браузері

Якщо ми зараз спробуємо клацнути на одному з користувачів у списку, ми будемо перенаправлені на сторінку 404, оскільки ми ще не створили жодного користувача сторінки.

Ми можемо вирішити цю проблему, створивши новий файл pages/users/[username].js і викликавши інший REST API для отримання даних одного користувача.

Щоб отримати дані окремого користувача, ми можемо викликати наступну URL-адресу https://api.rwnjs.com/04/users/[username] , де [ім'я користувача] — це змінна маршруту, що представляє користувача, дані якого ми хочемо отримати. .

Давайте перейдемо до файлу pages/users/[username].js і додамо такий вміст, починаючи з функції getServerSideProps:

```
import Link from 'next/link';
import axios from 'axios';
```

```
export async function getServerSideProps(ctx) {
  const { username } = ctx.query;
  const userReq =
    await axios.get(
      `https://api.rwnjs.com/04/users/${username}`
    );
  return {
    props: {
      user: userReq.data
    }
  };
}
```

Тепер у той самий файл додамо функцію UserPage, яка буде шаблоном сторінки для нашого маршруту /users/[username]:

```
function UserPage({ user }) {
  return (
    <div>
      <div>
        <Link href="/" passHref>
          Back to home
        </Link>
      </div>
      <hr />
      <div style={{ display: 'flex' }}>
        <img
          src={user.profile_picture}
          alt={user.username}
          width={150}
          height={150}
        />
        <div>
          <div>
            <b>Username:</b> {user.username}
          </div>
          <div>
            <b>Full name:</b>
              {user.first_name} {user.last_name}
          </div>
          <div>
            <b>Email:</b> {user.email}
          </div>
          <div>
            <b>Company:</b> {user.company}
          </div>
          <div>
```

```
            <b>Job title:</b> {user.job_title}
          </div>
        </div>
      </div>
      </div>
  );
}
export default UserPage;
```

Але все ще є проблема: якщо ми спробуємо відобразити сторінку одного користувача, ми отримаємо помилку на стороні сервера, оскільки ми не авторизовані отримувати дані з цього API. Пам'ятаєте, що ми говорили на початку цього розділу? Не всі API є загальнодоступними, що має великий сенс, оскільки іноді ми хочемо отримати доступ до дуже конфіденційної інформації, а компанії та розробники захищають цю інформацію, обмежуючи доступ до своїх API лише авторизованим людям.

У такому випадку нам потрібно передати дійсний маркер як заголовок авторизації HTTP під час виконання запиту API, щоб сервер знав, що ми маємо доступ до цієї інформації:

```
export async function getServerSideProps(ctx) {
  const { username } = ctx.query;
  const userReq = await axios.get(
    `https://api.rwnjs.com/04/users/${username}`,
    {
      headers: {
        authorization: process.env.API_TOKEN
      }
    }
  );
  return {
    props: {
      user: userReq.data
    }
  };
}
```

Як бачите, axios дуже спрощує додавання HTTP-заголовка до запиту, оскільки нам потрібно лише передати об'єкт як другий аргумент його методу get, який містить властивість під назвою headers, яка є об'єктом, що включає всі HTTP-заголовки. заголовки, які ми хочемо надіслати на сервер у нашому запиті.

Вам може бути цікаво, що означає process.env.API_TOKEN. Хоча можна передати жорстко закодований рядок як значення для цього заголовка, це погана практика з таких причин:

1. Під час фіксації вашого коду за допомогою Git або будь-якої іншої системи контролю версій кожен, хто має доступ до цього сховища, зможе прочитати особисту інформацію, таку як маркер авторизації (навіть сторонні співавтори). Розглядайте це як пароль, який слід тримати в секреті.
2. У більшості випадків токени API змінюються залежно від етапу, на якому ми запускаємо нашу програму: запускаючи нашу програму локально, ми можемо захотіти отримати доступ до API за допомогою тестового токена, а під час розгортання використовувати робочий. Використання змінної середовища полегшить нам використання різних токенів залежно від середовища. Те саме стосується кінцевих точок API, але ми побачимо це пізніше в цьому розділі.
3. Якщо маркер API змінюється з будь-якої причини, ви можете легко відредагувати його за допомогою спільного файлу середовища для всієї програми замість того, щоб змінювати значення маркера в кожному запиті HTTP.

Отже, замість того, щоб вручну записувати конфіденційні дані в наші файли, ми можемо створити новий файл із назвою .env у кореневій папці нашого проекту та додати всю інформацію, необхідну для роботи нашої програми.

Ніколи не фіксуйте (Commit) свій файл .env

Файл .env містить конфіденційну та конфіденційну інформацію, і його ніколи не можна фіксувати за допомогою будь-якого програмного забезпечення для контролю версій. Обов’язково додайте .env до своїх .gitignore, .dockerignore та інших подібних файлів перед розгортанням або фіксацією коду.

Тепер давайте створимо та відредагуємо файл .env, додавши такий вміст:

```
API_TOKEN=realworldnextjs
API_ENDPOINT=https://api.rwnjs.com
```

Next.js має вбудовану підтримку файлів .env і .env.local, тому вам не потрібно встановлювати зовнішні бібліотеки для доступу до цих змінних середовища.

Після редагування файлу ми можемо перезапустити сервер Next.js і клацнути будь-якого користувача, указаного на домашній сторінці,

отримуючи доступ до сторінки з інформацією про користувача, яка має виглядати так:

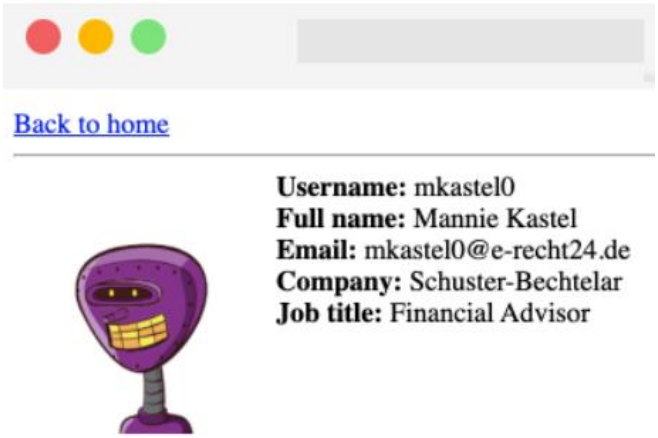

Рис. 4.2 – Сторінка інформації про користувача

Якщо ми спробуємо перейти на таку сторінку, як http://localhost:3000/users/mitch , ми отримаємо повідомлення про помилку, оскільки користувач із ім'ям mitch не існує, а REST API поверне код статусу 404. Ми можемо легко виявити цю помилку та повернути сторінку Next.js за замовчуванням 404, просто додавши такий сценарій до функції getServerSideProps:

```
export async function getServerSideProps(ctx) {
  const { username } = ctx.query;
  const userReq = await axios.get(
    `${process.env.API_ENDPOINT}/04/users/${username}`,
    {
      headers: {
        authorization: process.env.API_TOKEN
      }
    }
  );
  if (userReq.status === 404) {
    return {
      notFound: true
    };
  }
  return {
    props: {
      user: userReq.data
    }
  };
}
```

Таким чином, Next.js автоматично перенаправить нас на сторінку 404 за замовчуванням, не потребуючи додаткових налаштувань.

Отже, ми побачили, як Next.js дозволяє нам отримувати дані виключно на стороні сервера за допомогою вбудованої функції getServerSideProps. Замість цього ми могли б використати функцію getStaticProps, тобто сторінка була б статично відтворена під час збирання, як показано в Розділі 2, Вивчення різних стратегій відтворення.

У наступному розділі ми побачимо, як отримати дані лише на стороні клієнта.

Отримання даних на стороні клієнта

Отримання даних на стороні клієнта є важливою частиною будь-якої динамічної веб-програми. Хоча отримання даних на стороні сервера може бути відносно безпечним (якщо це робити з обережністю), отримання даних у браузері може додати деякі додаткові складності та вразливості.

Здійснення HTTP-запитів на сервері приховує від користувачів кінцеву точку API, параметри, заголовки HTTP та, можливо, маркери авторизації. Однак, роблячи це з браузера, можна розкрити цю конфіденційну інформацію, що полегшить зловмисним користувачам здійснення безлічі можливих атак, які використовують ваші дані.

Під час виконання HTTP-запитів у браузерах деякі конкретні правила необов’язкові:

1. Надсилайте HTTP-запити лише до надійних джерел. Ви завжди повинні досліджувати, хто розробляє API, які ви використовуєте, і їхні стандарти безпеки.

2. Викликати HTTP API лише за умови захисту сертифікатом SSL. Якщо віддалений API не захищено за протоколом HTTPS, ви наражаєте себе та своїх користувачів на численні атаки, такі як людина посередині, коли зловмисник може перехопити всі дані, що передаються від клієнта та сервера за допомогою простий проксі.

3. Ніколи не підключайтеся до віддаленої бази даних із браузера. Це може здатися очевидним, але для JavaScript технічно можливий доступ до віддалених баз даних. Це наражає вас і ваших користувачів на високий ризик, оскільки будь-хто потенційно може використати вразливість і отримати доступ до вашої бази даних.

У наступному розділі ми детальніше розглянемо використання REST API на стороні клієнта.

Використання REST API на стороні клієнта

Як і на стороні сервера, отримання даних на стороні клієнта відносно просте, і якщо ви вже маєте досвід роботи з React або будь-яким іншим фреймворком або бібліотекою JavaScript, ви можете повторно використовувати свої поточні знання для виконання запитів REST із браузера без будь-яких ускладнень. .

У той час як фаза отримання даних на стороні сервера в Next.js відбувається лише тоді, коли вона оголошена у вбудованих функціях getServerSideProps і getStaticProps, якщо ми робимо запит на вибірку всередині даного компонента, він буде виконаний на стороні клієнта за замовчуванням.

Зазвичай ми хочемо, щоб наші клієнтські запити запускалися у двох випадках:

• Відразу після монтування компонента

• Після певної події

В обох випадках Next.js не змушує вас виконувати ці запити інакше, ніж React, тому ви можете зробити HTTP-запит, використовуючи вбудований у браузері API отримання або зовнішню бібліотеку, таку як axios, як ми бачили в попередньому розділі. Давайте спробуємо відтворити ту саму просту програму Next.js із попереднього розділу, але перемістимо всі виклики API на сторону клієнта.

Створіть новий проект Next.js і відредагуйте файл pages/index.js таким чином:

```
import { useEffect, useState } from 'react';
import Link from 'next/link';
function List({users}) {
  return (
    <ul>
     {
        users.map((user) =>
          <li key={user.id}>
            <Link
              href={`/users/${user.username}`}
              passHref
            >
              <a> {user.username} </a>
```

```
            </Link>
          </li>
        )
      }
    </ul>
    )
  }
function Users() {
  const [loading, setLoading] = useState(true);
  const [data, setData] = useState(null);
  useEffect(async () => {
    const req =
      await fetch('https://api.rwnjs.com/04/users');
    const users = await req.json();
    setLoading(false);
    setData(users);
    }, []);
  return (
    <div>
      {loading &&<div>Loading users...</div>}
      {data &&<List users={data} />}
    </div>
  )
}
export default Users;
```

Чи можете ви помітити відмінності між цим компонентом і його аналогом SSR?

• HTML-код, створений на стороні сервера, містить текст Завантаження користувачів..., оскільки це початковий стан нашого компонента HomePage.

• Ми зможемо побачити список користувачів лише після гідратації React. Нам потрібно буде дочекатися монтування компонента на стороні клієнта та створення запиту HTTP за допомогою API отримання браузера.

Тепер нам потрібно реалізувати сторінку одного користувача наступним чином:

1. Давайте створимо новий файл pages/users/[ім’я користувача].js і почнемо писати функцію getServerSideProps, де ми отримуємо змінну [ім’я користувача] з маршруту та маркер авторизації з файлу .env:

```
import { useEffect, useState } from 'react'
import Link from 'next/link';
export async function getServerSideProps({ query }) {
  const { username } = query;
  return {
```

```
    props: {
      username,
      authorization: process.env.API_TOKEN
    }
  }
}
```

2. Тепер у тому самому файлі створимо компонент UserPage, де ми будемо виконувати функцію отримання даних на стороні клієнта:

```
function UserPage({ username, authorization }) {
  const [loading, setLoading] = useState(true);
  const [data, setData] = useState(null);
  useEffect(async () => {
    const req = await fetch(
      `https://api.rwnjs.com/04/users/${username}`,
      { headers: { authorization } }
    );
    const reqData = await req.json();
    setLoading(false);
    setData(reqData);
  }, []);
  return (
    <div>
      <div>
        <Link href="/" passHref>
          Back to home
        </Link>
      </div>
      <hr />
      {data && <UserData user={data} />}
    </div>
  );
}
export default UserPage;
```

Як ви могли помітити, коли ми встановлюємо дані за допомогою функції підключення setData, ми візуалізуємо компонент <UserData />.

3. Створіть останній компонент, завжди в одному компоненті pages/users/[username].js:

```
function UserData({ user }) {
  return (
    <div style={{ display: 'flex' }}>
      <img
        src={user.profile_picture}
```

```
          alt={user.username}
          width={150}
          height={150}
        />
      <div>
        <div>
          <b>Username:</b> {user.username}
        </div>
        <div>
          <b>Full name:</b>
          {user.first_name} {user.last_name}
        </div>
        <div>
          <b>Email:</b> {user.email}
        </div>
        <div>
          <b>Company:</b> {user.company}
        </div>
        <div>
          <b>Job title:</b> {user.job_title}
        </div>
      </div>
    </div>
  )
}
```

Як бачите, ми використовуємо той самий підхід, що й для домашньої сторінки, роблячи HTTP-запит, щойно компонент монтується на стороні клієнта. Ми також передаємо API_TOKEN від сервера до клієнта за допомогою getServerSideProps, щоб ми могли використовувати його для авторизованого запиту. Однак, якщо ви спробуєте запустити попередній код, ви побачите принаймні дві проблеми. Перший пов'язаний з CORS.

CORS (що розшифровується як Cross-Origin Resource Sharing) — це механізм безпеки, реалізований у браузерах, який має на меті контролювати запити, зроблені з доменів, відмінних від домену API. У нашому компоненті HomePage ми змогли викликати https://api.rwnjs.com/04/users API з іншого домену (localhost, домен replit.co, домен CodeSandbox тощо) як сервер дозволяє будь-якому домену отримувати доступ до своїх ресурсів для цього конкретного маршруту.

Однак у цьому випадку браузер накладає деякі обмеження на кінцеву точку https://api.rwnjs.com/04/users/[username] , і ми не можемо викликати цей API безпосередньо з клієнта, оскільки нас блокують за політикою CORS. CORS іноді може бути складним, і я раджу вам прочитати більше про це на

сторінці мережі Mozilla Developer Network, присвяченій цій політиці безпеки: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

Друга проблема стосується надання маркера авторизації клієнту. Фактично, якщо ми відкриємо інструменти розробника Google Chrome і перейдемо до Network, ми зможемо вибрати HTTP-запит для кінцевої точки та побачити маркер авторизації у вигляді звичайного тексту в розділі Request Headers:

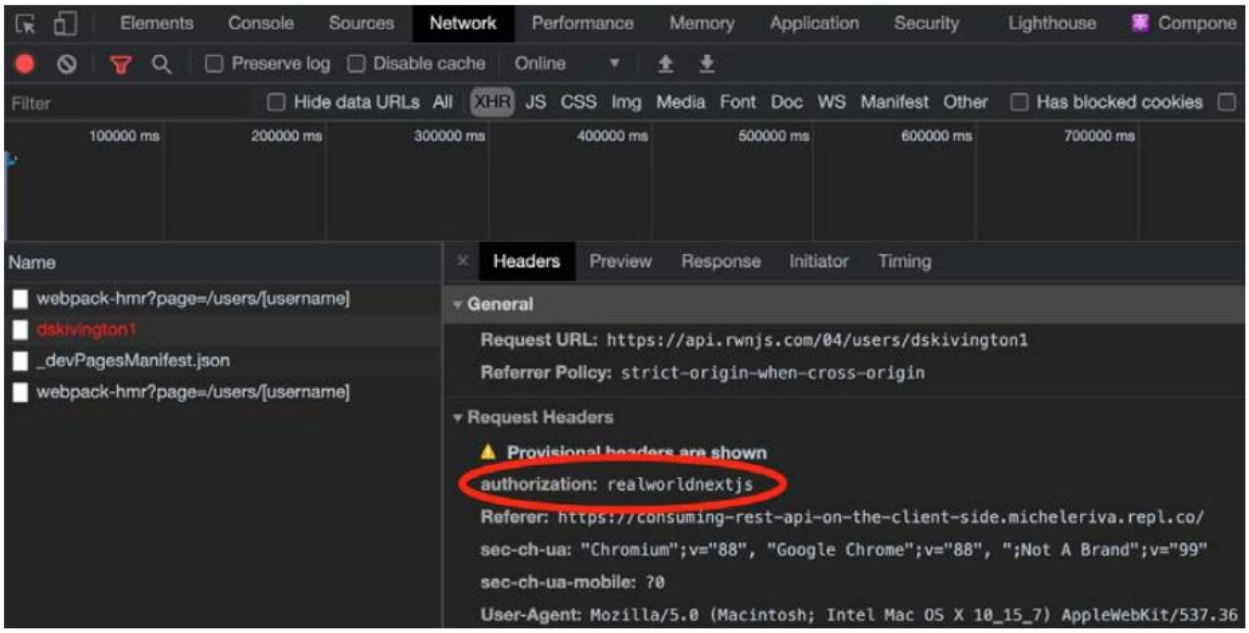

Рис. 4.3 – The HTTP Request Headers

Ну що в цьому поганого?

Уявіть, що ви платите за послугу, яка показує поточну інформацію про погоду через API, і уявіть, що це коштує 1 долар США за кожні 100 запитів.

Зловмисний користувач, який хоче скористатися цією самою послугою без оплати, може легко знайти ваш приватний маркер авторизації в заголовку запиту та використати його для роботи свого веб-додатку погоди. Таким чином, якщо зловмисник зробить 1000 запитів, ви заплатите 10 доларів, фактично не скориставшись його послугами.

Ми можемо швидко вирішити обидві проблеми завдяки сторінкам API Next.js, які дозволяють нам швидко створити REST API, зробивши запит HTTP для використання на стороні сервера та повернувши результат клієнту.

Давайте створимо нову папку всередині pages/ під назвою api/ та новий файл pages/api/singleUser.js:

```
import axios from 'axios';
export default async function handler(req, res) {
  const username = req.query.username;

  const API_ENDPOINT = process.env.API_ENDPOINT;
  const API_TOKEN = process.env.API_TOKEN;
  const userReq = await axios.get(
    `${API_ENDPOINT}/04/users/${username}`,
    { headers: { authorization: API_TOKEN } }
  );
  res
    .status(200)
    .json(userReq.data);
}
```

Як бачите, у цьому випадку ми представляємо просту функцію, яка приймає два аргументи:

- req: екземпляр Node.js *http.IncomingMessage* (https://nodejs.org/api/http.html#http_class_http_incomingmessage), об'єднаний із деякими попередньо створеними проміжними програмами, такими як req.cookies, req.query та req.body.
- res: екземпляр Node.js *http.serverResponse* (https://nodejs.org/api/http.html#http_class_http_serverresponse), об'єднаний із деяким попередньо створеним проміжним програмним забезпеченням, таким як res.status (code) для налаштування HTTP status code, res.json(json) для повернення дійсного JSON, res.send(body) для надсилання HTTP-відповіді, що містить рядок, об'єкт або Buffer, і res.redirect([status,] path) для перенаправлення на певну сторінку з заданим (і необов'язковим) кодом статусу.

Кожен файл у каталозі pages/api/ розглядатиметься Next.js як маршрут API.

Тепер ми можемо змінити (refactor) наш компонент UserPage, змінивши кінцеву точку API на щойно створену:

```
function UserPage({ username }) {
  const [loading, setLoading] = useState(true);
  const [data, setData] = useState(null);
  useEffect(async () => {
    const req = await fetch(
      `/api/singleUser?username=${username}`,
    );
    const data = await req.json();
    setLoading(false);
```

```
    setData(data);
  }, []);
  return (
    <div>
      <div>
        <Link href="/" passHref>
          Back to home
        </Link>
      </div>
      <hr />
      {loading && <div>Loading user data...</div>}
      {data && <UserData user={data} />}
    </div>
  );
}
```

Якщо ми зараз спробуємо запустити наш веб-сайт, ми побачимо, що обидві наші проблеми вирішено!

Але є ще дещо, на що ми повинні звернути увагу. Ми приховали маркер API, написавши своєрідний проксі-сервер для API для одного користувача, але зловмисник усе одно зможе використовувати маршрут /api/singleUser для легкого доступу до приватних даних.

Щоб вирішити цю конкретну проблему, ми можемо діяти різними способами:

• Візуалізуйте список компонентів виключно на сервері, як у попередньому розділі: таким чином зловмисник не викличе приватний API або не вкраде секретний маркер API. Однак є випадки, коли ви не можете запустити такі виклики API лише на сервері; якщо вам потрібно зробити запит REST після того, як користувач натисне кнопку, ви змушені зробити це на стороні клієнта.

• Використовуйте метод автентифікації, щоб надати автентифікованим користувачам доступ лише до певного API (JWT, ключ API тощо).

• Використовуйте бекенд-фреймворк, такий як Ruby on Rails, Spring, Laravel, Nest.js і Strapi: усі вони надають різні способи захисту ваших викликів API від клієнта, що робить нам набагато зручнішим створювати безпечні програми Next.js .

У розділі 13 «Створення веб-сайту електронної комерції за допомогою Next.js і GraphCMS» ми побачимо, як використовувати Next.js як інтерфейс для різних CMS і платформ електронної комерції, а також розглянемо автентифікацію користувачів і безпечні виклики API . Наразі в цьому розділі

ми зосередимося лише на тому, як робити HTTP-запити як від сервера, так і від клієнта.

У наступному розділі ми побачимо, як прийняти GraphQL як альтернативу REST для отримання даних у Next.js.

Резюме

У цьому розділі ми розглянули дві ключові теми, коли говоримо про Next.js: організацію структури проекту та різні способи отримання даних. Навіть якщо ці дві теми здаються непов'язаними, вміння логічно розділяти компоненти та утиліти та отримувати дані різними способами є важливими навичками, які дозволять вам краще зрозуміти наступну главу, главу 5, Керування локальними та глобальними станами в Next.js . Як ми бачили в цій главі, складність будь-якої програми може лише зростати з часом, оскільки ми додаємо нові функції, виправляємо помилки тощо. Наявність добре організованої структури папок і чіткого потоку даних може допомогти нам відстежувати стан нашої програми.

Ми також розглянули, як отримати дані за допомогою GraphQL. Це захоплююча тема, оскільки в наступному розділі ми побачимо, як використовувати клієнт Apollo як менеджер стану, відмінний від клієнта GraphQL.

Розділ 5. Управління локальними та глобальними станами в Next.js

Управління станом є однією з центральних частин будь-якої програми React, включаючи програми Next.js. Говорячи про стан, ми маємо на увазі ті динамічні фрагменти інформації, які дозволяють нам створювати високоінтерактивні інтерфейси користувача (UI), роблячи роботу наших клієнтів максимально красивою та приємною.

Розглядаючи сучасні веб-сайти, ми можемо помітити зміни стану в багатьох частинах інтерфейсу користувача: перехід зі світлої теми на темну означає, що ми змінюємо стан теми інтерфейсу користувача, заповнення форми електронної комерції інформацією про доставку означає, що ми змінюємо у цьому стані форми навіть натискання простої кнопки потенційно може змінити локальний стан, оскільки це може змусити наш інтерфейс користувача реагувати різними способами, залежно від того, як розробники вирішили керувати цим оновленням стану.