NPM. Package.json. Встановлення модулів. Визначення команд

Окрім вбудованих та кастомних модулів Node.js існує величезний пласт різних бібліотек та фреймворків, різноманітних утиліт, які створюються сторонніми виробниками та які також можна використовувати у проекті, наприклад, express, grunt, gulp тощо. І вони також нам доступні в рамках Node.js. Щоб зручніше було працювати з усіма сторонніми рішеннями, вони поширюються у вигляді пакетів. Пакет насправді представляє набір функціональностей.

Для автоматизації встановлення та оновлення пакетів, як правило, застосовуються систему керування пакетами або менеджери. Безпосередньо в Node.js використовується пакетний менеджер NPM (Node Package Manager). NPM за замовчуванням встановлюється разом з Node.js, тому нічого не потрібно встановлювати. Але можна оновити встановлену версію до останньої. Для цього в командному рядку/терміналі треба запустити наступну команду:

```
npm install npm@latest -g
```

Щоб дізнатися про поточну версію npm, у командному рядку/терміналі треба ввести наступну команду:

```
npm -v
```

Для нас менеджер npm важливий у тому плані, що за його допомогою легко керувати пакетами. Наприклад, створимо на жорсткому диску нову папку *modulesapp* (У моєму випадку папка перебуватиме на шляху *C:|node|modulesapp*).

Якщо надалі нам більше не буде потрібно express, то ми його можемо видалити наступною командою:

```
npm uninstall express
```

Файл package.json

Для зручнішого керування конфігурацією та пакетами програми в npm застосовується файл конфігурації **package.json** . Так, додамо в папку проекту modulesapp новий файл *package.json* :

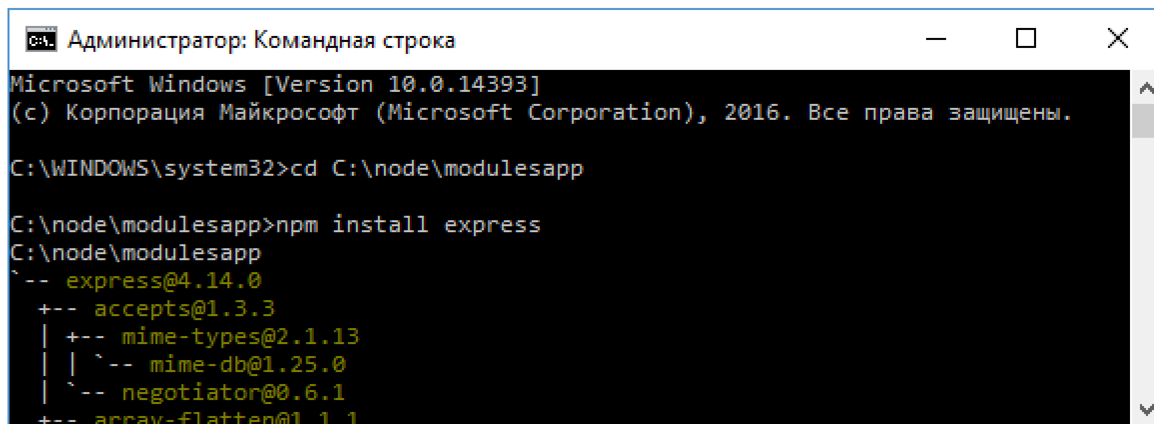
```
{
  "name": "modulesapp",
  "version": "1.0.0"
}
```

Здесь определены только две секции: имя проекта - modulesapp и его версия - 1.0.0. Это минимально необходимое определение файла package.json. Данный файл может включать гораздо больше секций. Подробнее можно посмотреть в документации.

Далее для примера установим в проект **express**. Express представляет легковесный веб-фреймворк для упрощения работы с Node.js. В данном случае мы не будем пока подробно рассматривать фреймворк Express, так как это отдельная большая тема. А используем его лишь для того, чтобы понять, как устанавливаются сторонние модули в проект.

Для установки функциональности Express в проект вначале перейдем к папке проекта с помощью команды **cd**. Затем введем команду

```
npm install express
```



После установки express в папке проекта modulesapp появится подпапка **node_modules**, в которой будут хранится все установленные внешние модули. В частности, в подкаталоге *node_modules/express* будут располагаться файлы фреймворка Express.

И после выполнения команды, если мы откроем файл *package.json*, то мы увидим информацию о пакете:

```
{
  "name": "modulesapp",
```

```
3     "version": "1.0.0",
4     "dependencies": {
5       "express": "^4.17.1"
6     }
7   }
```

Информация обо всех добавляемых пакетах, которые используются при работе приложения, добавляется в секцию **dependencies**.

Используем добавленный пакет express и для этого определим файл простейшего сервера. Для этого в корневую папку проекта modulesapp добавим новый файл *app.js*:

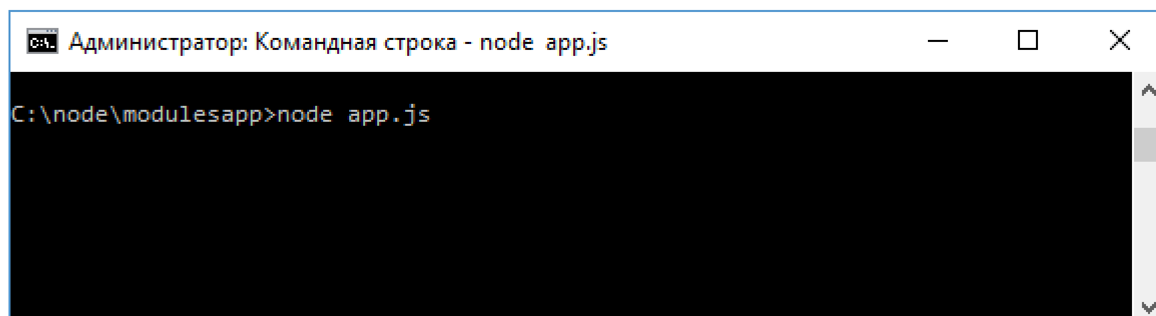
```
// получаем модуль Express
const express = require("express");
// создаем приложение
const app = express();

// устанавливаем обработчик для маршрута "/"
app.get("/", function(request, response){

    response.end("Hello from Express!");
});
// начинаем прослушивание подключений на 3000 порту
app.listen(3000);
```

Первая строка получает установленный модуль express, а вторая создает объект приложения.

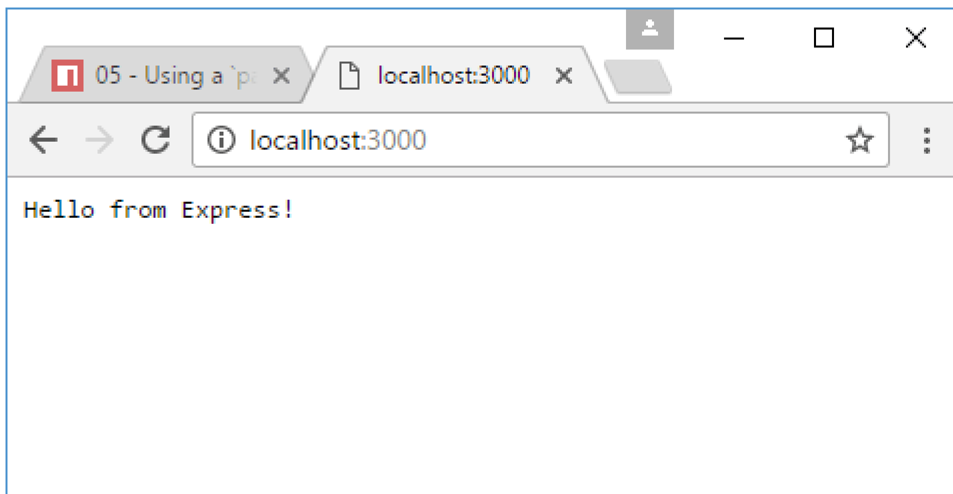
В Express мы можем связать обработку запросов с определенными маршрутами. Например, "/" - представляет главную страницу или корневой маршрут. Для обработки запроса вызывается функция `app.get()`. Первый параметр функции - маршрут, а второй - функция, которая будет обрабатывать запрос по этому маршруту.

И чтобы сервер начал прослушивать подключения, надо вызвать метод `app.listen()`, в который передается номер порта.

Запустим сервер командой `node app.js`:



И в адресной строке браузера введем адрес *http://localhost:3000/*:



Добавление множества пакетов

Файл *package.json* играет большую роль и может облегчить работу с пакетами в различных ситуациях. Например, мы планируем использовать множество пакетов. Но вводить для установки каждого пакета в консоли соответствующую команду не очнь удобно. В этом случае мы можем определить все пакеты в файле package.json и потом одной командой их установить.

Например, изменим файл package.json следующим образом:

```
{
  "name": "modulesapp",
  "version": "1.0.0",
  "dependencies": {
    "express": "^4.17.1",
    "react": "^16.9.0",
    "react-dom": "^16.9.0"
  }
}
```

Здесь добавлены определения двух пакетов, которые представляют библиотеку React.

Затем для загрузки всех пакетов выполнить команду

```
npm install
```

Эта команда возьмет определение всех пакетов из секций dependencies и загрузит их в проект. Если пакет с нужной версией уже есть проекте, как в данном случае express, то по новой он не загружается.

devDependencies

Кроме пакетов, которые применяются в приложении, когда оно запущено и находится в рабочем состояни, например, express, то есть в состоянии "production", есть еще пакеты,

которые применяются при разработке приложения и его тестировании. Такие пакеты, как правило, добавляются в другую секцию - **devDependencies**.

Например, загрузим в проект пакет **jasmine-node**, который используется для тестирования приложения:

```
npm install jasmine-node --save-dev
```

Флаг `--save-dev` указывается, что информацию о пакете следует сохранить именно в секции devDependencies файла package.json:

```
1  {
2    "name": "modulesapp",
3    "version": "1.0.0",
4    "dependencies": {
5      "express": "^4.17.1",
6      "react": "^16.9.0",
7      "react-dom": "^16.9.0"
8    },
9    "devDependencies": {
10     "jasmine-node": "^3.0.0"
11   }
12 }
```

Удаление пакетов

Для удаления пакетов используется команда `npm uninstall`. Например:

```
npm uninstall express
```

При этом не важно, где располагается информация о пакете - в секции dependencies или devDependencies, пакет удаляется из любой из этих секций.

Если нам надо удалить не один пакет, а несколько, то мы можем удалить их определение из файла package.json и ввести команду **npm install**, и удаленые из package.js пакеты также будут удалены из папки node_modules.

Например, изменим файл package.json следующим образом:

```
1 {
2   "name": "modulesapp",
3   "version": "1.0.0",
4   "dependencies": {
5   }
6 }
```

Здесь больше нет определения никаких пакетов. И введем команду

```
npm install
```

Причем мы также можем одновременно некоторые пакеты добавлять в package.json, а некоторые, наоборот, удалять. И при выполнении команды **npm install** пакетный менеджер новые пакеты установит, а удаленные из package.json пакеты удалит.

Семантическое версионирование

При определении версии пакета применяется семантическое версионирование. Номер версии, как правило, задается в следующем формате "major.minor.patch". Если в приложении или пакете обнаружен какой-то баг и он исправляется, то увеличивается на единицу число "patch". Если в пакет добавляется какая-то новая функциональность, которая совместима с предыдущей версией пакета, то это небольшое изменение, и увеличивается число "minor". Если же в пакет вносятся какие-то большие изменения, которые несовместимы с предыдущей версией, то увеличивается число "major". То есть глядя на разные версии пакетов, мы можем предположить, насколько велики в них различия.

В примере с express версия пакета содержала, кроме того, дополнительный символ карет: "^4.14.0". Этот символ означает, что при установке пакета в проект с помощью команды `npm install` будет устанавливаться последняя доступная версия от 4.14.0. Фактически это будет последняя доступная версия в промежутке от 4.14.0 до 5.0.0 (>=4.14.0 и <5.0.0).

Команды npm

NPM позволяет определять в файле package.json команды, которые выполняют определенные действия. Например, определим следующий файл **app.js**:

```
let name = process.argv[2];
let age = process.argv[3];

console.log("name: " + name);
console.log("age: " + age);
```

В данном случае мы получаем переданные при запуске приложению параметры.

И определим следующий файл **package.json**:

```
{
  "name": "modulesapp",
  "version": "1.0.0",
  "scripts" : {
    "start" : "node app.js",
    "dev" : "node app.js Tom 26"
```

```
7     }
8 }
```

Здесь добавлена секция **scripts**, которая определяет две команды. Вообще команд может быть много в соответствии с целями и задачами разработчика.

Первая команда называется **start**. Она по сути выполняет команду `node app.js`, которая выполняет код в файле app.js

Вторая команда назвывается **dev**. Она также выполняет тот же файл, но при этом также передает ему два параметра.

Назви команд можуть бути довільними. Але тут треба зважати на один момент. Є умовно кажучи, є зарезервовані назви для команд, наприклад, `start`, `test`, `run`і ряд інших. Їх не дуже багато. І саме перша команда з вище визначеного файлу package.json називається start. І для виконання подібних команд у терміналі/командному рядку треба виконати команду

```
1 npm [название_команды]
```

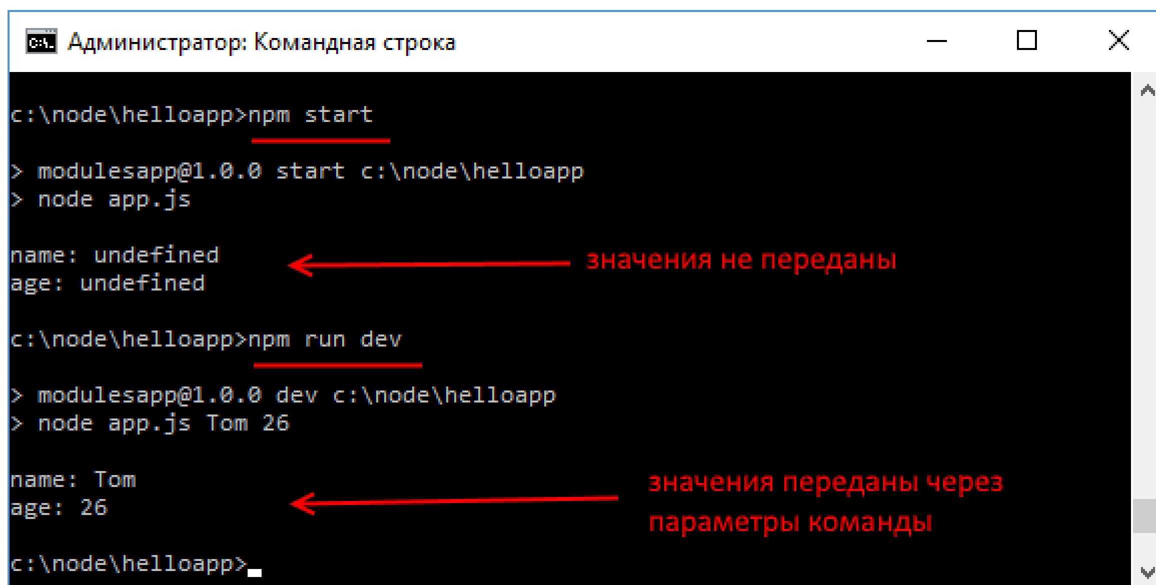
Наприклад, для запуску команди`start`

```
1 npm start
```

Команди з іншими назвами, як, наприклад, "dev" у вищевизначеному файлі, запускаються так:

```
1 npm run [название_команды]
```

Наприклад, послідовно виконаємо обидві команди:



[Назад](#) [Зміст](#) [Вперед](#)

ALSO ON **METANIT.COM**

Создание клиента для ChatGPT бота

8 дней назад · 5 комментар…

Создание клиента для чат-бота ChatGPT с помощью библиотеки openai в …

Создание клиента для ChatGPT бота

7 дней назад · 9 комментар…

Создание консольного клиента для чат-бота ChatGPT в приложении …

Создание библиотеки классов в Visual Studio

месяц назад · 1 комментарий

Создание библиотеки классов в C# и .NET в Visual Studio и …

Отпра

2 месяц

Отправ Core с г