

Node JS & Express Framework

Что такое среда разработки Express?

Среда разработки Express включает в себя установку Nodejs, менеджера пакетов NPM и (опционально) Express Generator на вашем локальном компьютере.

Node и менеджер пакетов NPM
устанавливаются вместе из готовых
двоичных пакетов, установщиков,
менеджеров пакетов операционной
системы или из источника.

express-generator

4.16.0 • Public • Published 5 months ago

Readme

5 Dependencies

express

Express' application generator.

npm v4.16.0

downloads 60k/m

linux passing

windows passing

NPM также можно использовать для установки Express Application Generator, удобного инструмента для создания веб-приложений скелета Express.

В отличие от других веб-фреймворков среда разработки не включает отдельный веб-сервер разработки. В Node / Express веб-приложение создает и запускает собственный веб-сервер

Другие зависимости, такие как драйверы баз данных, механизмы шаблонов, механизмы проверки подлинности и т. д., являются частью приложения и импортируются в среду приложения с помощью диспетчера пакетов NPM.

НАЧАЛО РАБОТЫ

Сперва вы устанавливаете инструмент генератора на весь сайт с помощью диспетчера пакетов NPM, как показано:

```
npm install express-generator -g
```

Генератор имеет несколько параметров, которые вы можете просмотреть в командной строке с помощью команды `-help` (или `-h`):

```
1 > express --help
2
3 Usage: express [options] [dir]
4
5 Options:
6
7   -h, --help           output usage information
8     --version          output the version number
9   -e, --ejs            add ejs engine support
10     --pug              add pug engine support
11     --hbs              add handlebars engine support
12   -H, --hogan          add hogan.js engine support
13   -v, --view <engine> add view <engine> support (ejs|hbs|hjs|jade|pug|twig|vash) (defaults to jade)
14   -c, --css <engine>  add stylesheet <engine> support (less|stylus|compass|sass) (defaults to plain css)
15     --git              add .gitignore
16   -f, --force          force on non-empty directory
```


Далее вам необходимо выбрать
движок шаблонов

Вот чем они могут отличаться:

- Наследование макета
- Поддержка «Include»
- Возможность фильтровать значения переменных на уровне шаблона
- Возможность создавать выходные форматы, отличные от HTML
- Поддержка асинхронных операций и потоковой передачи

Для примера создадим проект с использованием библиотеки шаблонов Pug и без CSS-таблицы стилей

Сначала перейдите туда, где вы хотите создать проект, а затем запустите генератор экспресс-приложений в командной строке, как показано:

```
1 | express express-locallibrary-tutorial --view=pug
```

Генератор создаст (и перечислит) файлы проекта

```
1 create : express-locallibrary-tutorial
2   create : express-locallibrary-tutorial/package.json
3   create : express-locallibrary-tutorial/app.js
4   create : express-locallibrary-tutorial/public/images
5   create : express-locallibrary-tutorial/public
6   create : express-locallibrary-tutorial/public/stylesheets
7   create : express-locallibrary-tutorial/public/stylesheets/style.css
8   create : express-locallibrary-tutorial/public/javascripts
9   create : express-locallibrary-tutorial/routes
10  create : express-locallibrary-tutorial/routes/index.js
11  create : express-locallibrary-tutorial/routes/users.js
12  create : express-locallibrary-tutorial/views
13  create : express-locallibrary-tutorial/views/index.pug
14  create : express-locallibrary-tutorial/views/layout.pug
15  create : express-locallibrary-tutorial/views/error.pug
16  create : express-locallibrary-tutorial/bin
17  create : express-locallibrary-tutorial/bin/www
18
19  install dependencies:
20    > cd express-locallibrary-tutorial && npm install
21
22  run the app:
23    > SET DEBUG=express-locallibrary-tutorial:* & npm start
```

В конце вывода генератор предоставляет инструкции по установке зависимостей (как указано в файле `package.json`), а затем о том, как запустить приложение

Запуск скелетного сайта

- Сначала установите зависимости (команда установки выберет все пакеты зависимостей, перечисленные в файле `package.json` проекта)

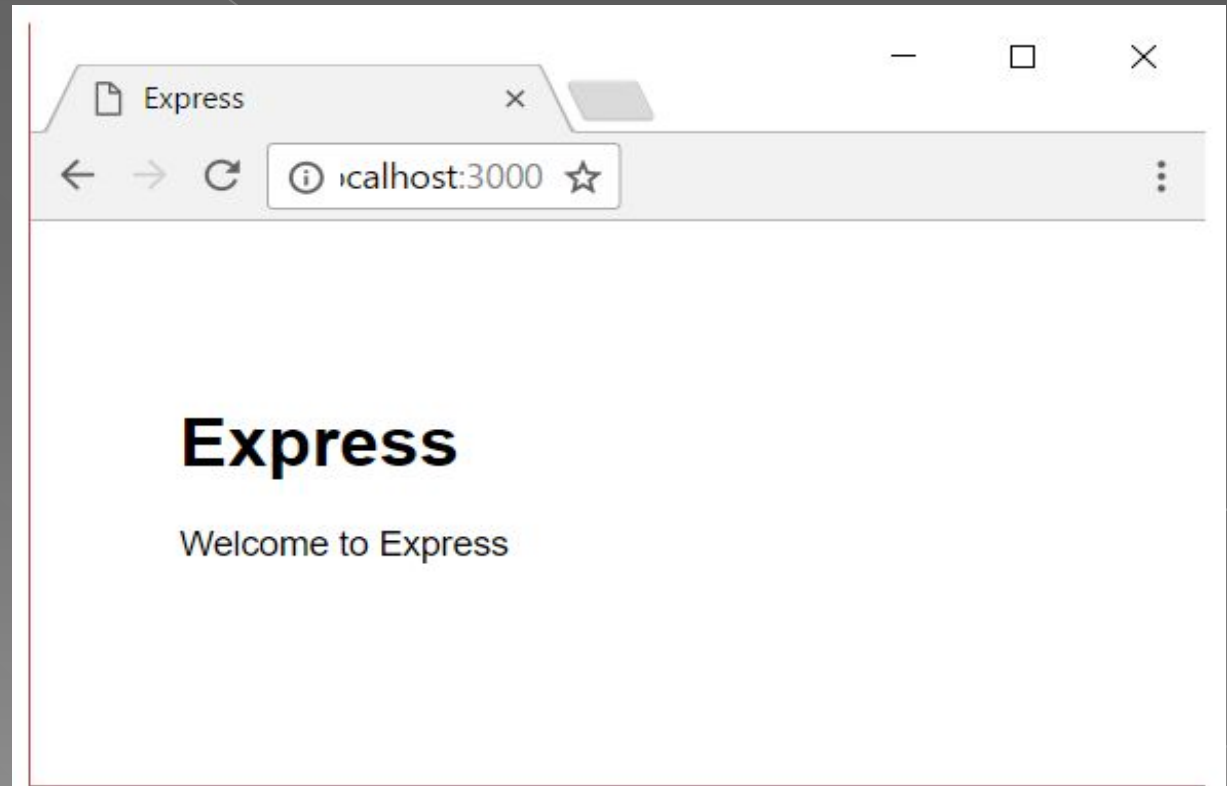
```
1 | cd express-locallibrary-tutorial  
2 | npm install
```

- Затем запустите приложение.
В Windows используйте следующую команду:

```
1 | SET DEBUG=express-locallibrary-tutorial:* & npm start
```


- Затем загрузите `http://localhost:3000/` в свой браузер, чтобы получить доступ к приложению.

Вы должны увидеть страницу браузера, которая выглядит так



Конвейер обработки запроса и middleware

```
1  const express = require("express");
2
3  const app = express();
4  app.get("/", function(request, response){
5
6      response.send("<h1>Главная страница</h1>");
7  });
8  app.get("/about", function(request, response){
9
10     response.send("<h1>О сайте</h1>");
11 });
12 app.get("/contact", function(request, response){
13
14     response.send("<h1>Контакты</h1>");
15 });
16 app.listen(3000);
```

Когда фреймворк Express получает запрос, этот запрос передается в конвейер обработки. Конвейер состоит из набора компонентов или middleware, которые получают данные запроса и решают, как его обрабатывать.

Здесь конвейер обработки состоял из вызовов `app.get()`, которые сравнивали запрошенный адрес с маршрутом, и если между адресом и маршрутом было соответствие, то данный запрос обрабатывался методом `app.get()`.

При необходимости мы можем встроить в конвейер обработки запроса на любом этапе любую функцию `middleware`. Для этого применяется метод `app.use()`. Так, изменим файл `app.js` следующим образом:

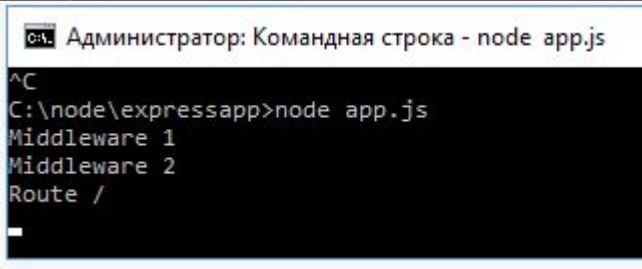
```
1  const express = require("express");
2
3  const app = express();
4  app.use(function(request, response, next){
5
6      console.log("Middleware 1");
7      next();
8  });
9  app.use(function(request, response, next){
10
11      console.log("Middleware 2");
12      next();
13  });
14
15  app.get("/", function(request, response){
16
17      console.log("Route /");
18      response.send("Hello");
19  });
20  app.listen(3000);
```

Функция, которая передается в `app.use()`, принимает три параметра:

- `request`: данные запроса
- `response`: объект для управления ответом
- `next`: следующая в конвейере обработки функция

Каждая из функций `middleware` просто выводит на консоль сообщение и в конце вызывает следующую функцию с помощью вызова `next()`.

При запуске приложения после обращения по адресу `"http://localhost:3000/"` последовательно отработают все три `middleware`:

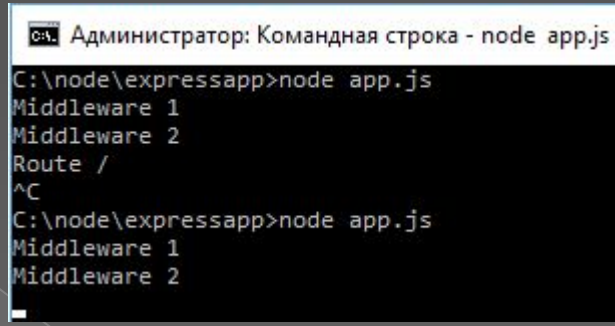


```
Администратор: Командная строка - node app.js
^C
C:\node\expressapp>node app.js
Middleware 1
Middleware 2
Route /
_
```

Однако необязательно вызывать все последующие middleware, мы можем на каком-то этапе остановить обработку:

```
1  const express = require("express");
2
3  const app = express();
4  app.use(function(request, response, next){
5
6      console.log("Middleware 1");
7      next();
8  });
9  app.use(function(request, response, next){
10
11      console.log("Middleware 2");
12      response.send("Middleware 2");
13  });
14
15  app.get("/", function(request, response){
16      console.log("Route /");
17      response.send("Hello");
18  });
19  app.listen(3000);
```

Теперь обработка завершается на Middleware 2, так как в этом методе происходит отправка ответа с помощью response.send(), а вызова следующей функции через next():



Пример Middleware

Middleware помогают выполнять некоторые задачи, которые должны быть сделаны до отправки ответа. Стандартная задача - логгирование запросов. Например, изменим файл `app.js` следующим образом:

```
1  const express = require("express");
2  const fs = require("fs");
3
4  const app = express();
5  app.use(function(request, response, next){
6
7      let now = new Date();
8      let hour = now.getHours();
9      let minutes = now.getMinutes();
10     let seconds = now.getSeconds();
11     let data = `${hour}:${minutes}:${seconds} ${request.method} ${request.url} ${request.get("user-agent")}`;
12     console.log(data);
13     fs.appendFile("server.log", data + "\n", function({});
14     next();
15 });
16
17 app.get("/", function(request, response){
18     response.send("Hello");
19 });
20 app.listen(3000);
```

Здесь с помощью объекта `request` получаем различную информацию о запросе и добавляем ее в файл `server.log`, используя модуль `fs`.

Отправка ответа

Для отправки ответа в `express` у объекта `response` можно использовать ряд функций. Самый распространенный способ отправки ответа представляет функция `send()`. В качестве параметра эта функция может принимать объект `Buffer`, строку, в том числе с `html`-кодом, объект `javascript` или массив.

```
1 const express = require("express");
2 const app = express();
3
4 app.use(function (request, response) {
5     response.send("<h2>Hello</h2>");
6 });
7
8 app.listen(3000);
```

Отправка объекта:

```
1 response.send({id:6, name: "Tom"});
```

Отправка массива:

```
1 response.send(["Tom", "Bob", "Sam"]);
```

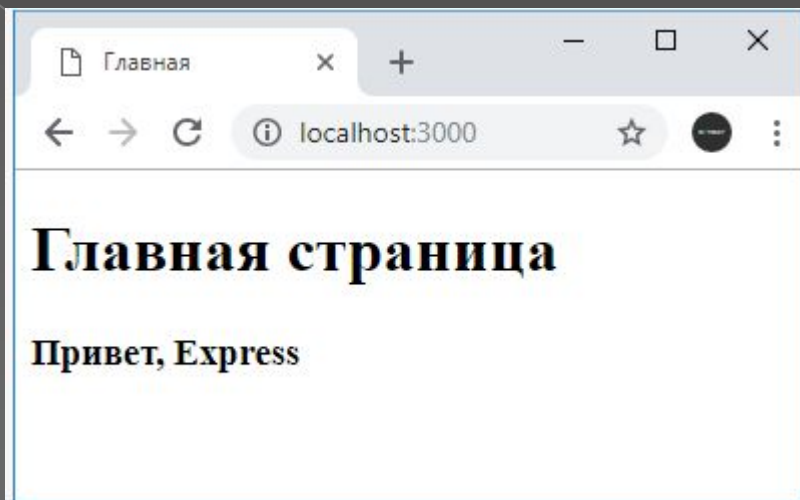
Отправка объекта:

```
1 response.send(Buffer.from("Hello Express"));
```

Объект `Buffer` формально представляет некоторые бинарные данные. Так, в случае выше при выполнении кода по умолчанию браузер загрузит файл, в котором будет строка "Hello Express".

sendFile()

Метод `send` удобен для отправки строк, некоторого кода `html` небольшой длины, однако если отправляемый код `html` довольно большой, то соответственно код приложения тоже становится громоздким. Например, мы можем написать так:



```
1 const express = require("express");
2 const app = express();
3
4 app.use(function (request, response) {
5   response.send(`<!DOCTYPE html>
6   <html>
7   <head>
8     <title>Главная</title>
9     <meta charset="utf-8" />
10  </head>
11  <body>
12    <h1>Главная страница</h1>
13    <h3>Привет, Express</h3>
14  </body>
15  <html>`);
16 });
17
18 app.listen(3000);
```

Однако гораздо лучше определять код `html` в отдельных файлах и затем эти файлы отправлять с помощью функции **`sendFile()`**. Например, определим в папке проекта новый файл **`index.html`**:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Главная</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <h1>Главная страница</h1>
9     <h3>Привет, Express</h3>
10  </body>
11 </html>
```

Отправим этот файл с помощью функции `sendFile`:

```
1 const express = require("express");
2 const app = express();
3
4 app.use(function (request, response) {
5   response.sendFile(__dirname + "/index.html");
6 });
7
8 app.listen(3000);
```

В итоге мы получим тот же самый результат. Следует учитывать, что в функцию `sendFile` необходимо передавать абсолютный путь к файлу, именно для этого с помощью `__dirname` получаем абсолютный путь к текущему проекту и затем добавляем к нему путь к файлу в рамках текущего проекта.

Отправка статусных кодов

Функция **`sendStatus()`** отправляет пользователю определенный статусный код с некоторым сообщением по умолчанию. Например, отправим статусный код 404, который говорит, что ресурс не найден:


```
1 const express = require("express");
2 const app = express();
3
4 app.use("/home/foo/bar",function (request, response) {
5   response.sendStatus(404)
6 });
7
8 app.listen(3000);
```

localhost:3000/home/foo/bar

Not Found

Name	Status	Type	Size	Time	Ca...	Waterfall
bar	404	document	22...	103 ...		

1 requests | 220 B transferred | Finish: 103 ms | DOMContentLoaded: 413 ms | ...

Как видно из скриншота, при отправке статусного кода 404 также отправляется сообщение "Not Found". Но, возможно, мы захотим отправлять какие-то свои более информативные сообщения. В этом случае можно использовать комбинацию функции **status()**, которая также отправляет статусный код, и функции **send()**:

```
1 const express = require("express");
2 const app = express();
3
4 app.use("/home/foo/bar",function (request, response) {
5   response.status(404).send(`Ресурс не найден`);
6 });
7
8 app.listen(3000);
```

Маршрутизация

При обработке запросов фреймворк Express опирается на систему маршрутизации. В приложении определяются маршруты, а также обработчики этих маршрутов. Если запрос соответствует определенному маршруту, то вызывается для обработки запроса соответствующий обработчик. Для обработки данных по определенному маршруту можно использовать ряд функций, в частности:

- use
- get
- post
- put
- delete

В качестве первого параметра эти функции могут принимать шаблон адреса, запрос по которому будет обрабатываться. Вторым параметром функций представляет функцию, которая будет обрабатывать запрос по совпавшему с шаблоном адресу. Например:

```
1  const express = require("express");
2  const app = express();
3
4  // обработка запроса по адресу /about
5  app.get("/about", function(request, response){
6
7      response.send("<h1>О сайте</h1>");
8  });
9
10 // обработка запроса по адресу /contact
11 app.use("/contact", function(request, response){
12
13     response.send("<h1>Контакты</h1>");
14 });
15
16 // обработка запроса к корню веб-сайта
17 app.get("/", function(request, response){
18
19     response.send("<h1>Главная страница</h1>");
20 });
21 app.listen(3000);
```

СИМВОЛЫ ПОДСТАНОВОК

Используемые шаблоны адресов могут содержать регулярные выражения или специальные символы подстановок. В частности, мы можем использовать такие символы, как `?`, `+`, `*` и `()`.

К примеру, символ `?` указывает, что предыдущий символ может встречаться 1 раз или отсутствовать. И мы можем определить следующую функцию:

```
1 app.get("/bo?k", function (request, response) {
2   response.send(request.url)
3 });
```

Такой маршрут будет соответствовать строке запроса `/bk` или `/bok`.

Символ `+` указывает, что предыдущий символ может встречаться 1 и более раз:

```
1 app.get("/bo+k", function (request, response) {
2   response.send(request.url)
3 });
```

Такой маршрут будет соответствовать запросам `/bok`, `/book`, `/boook` и так далее.

Символ звездочка `*` указывает, что на месте данного символа может находиться любое количество символов:

```
1 app.get("/bo*k", function (request, response) {
2   response.send(request.url)
3 });
```

Такой маршрут будет соответствовать запросам `/bork`, `/bonk`, `/bor.dak`, `/bor/ok` и так далее.

Скобки () позволяют оформить группу символов, которые могут встречаться в запросе:

```
1 app.get("/book(.html)?", function (request, response) {  
2   response.send(request.url)  
3 });
```

Выражение `(.html)?` указывает, что подстрока `".html"` может встречаться или отсутствовать в строке запроса. И такой маршрут будет соответствовать запросам `"/book"` и `"/book.html"`. Также вместо определения маршрутов мы можем указывать регулярные выражения. Например, необходимо перехватить запрос ко всем файлам `html` или всем путям, которые в конце имеют `".html"`:

```
1 app.get(/.*(\.)html$/, function (request, response) {  
2   response.send(request.url)  
3 });
```

Переадресация

Для переадресации применяется метод **redirect()**: `redirect([status,] path)`

В качестве параметра `path` передается путь, на который будет перенаправляться пользователь. Дополнительный параметр `status` задает статусный код переадресации. Если этот параметр не задан, тогда по умолчанию отправляется статусный код 302, который предполагает временную переадресацию.

С помощью данного метода можно выполнять переадресацию как по относительным путям, так и по абсолютным, в том числе на другие домены.

Переадресация по абсолютному пути:

```
1 | const express = require("express");
2 | const app = express();
3 |
4 | app.use("/index", function (request, response) {
5 |   response.redirect("https://metanit.com")
6 | });
7 |
8 | app.listen(3000);
```

Переадресация по относительным путям также не очень сложна, но здесь важно учитывать, как именно определяется адрес для редиректа. Рассмотрим редирект относительно текущего пути, с которого производится редирект. Например:

```
1 const express = require("express");
2 const app = express();
3
4 app.use("/home",function (request, response) {
5   response.redirect("about")
6 });
7 app.use("/about", function (request, response) {
8   response.send("<h1>About</h1>");
9 });
10
11 app.listen(3000);
```

В данном случае будет идти перенаправление с ресурса `/home` на ресурс `/about`, то есть, условно говоря, с `http://localhost:3000/home` на `http://localhost:3000/about`.

Еще несколько примеров. Переадресация относительно текущего адреса на адрес на том же уровне:

```
1 app.use("/home/foo/bar",function (request, response) {
2   response.redirect("../about")
3 });
```

Здесь идет переадресация с `http://localhost:3000/home/foo/bar` на `http://localhost:3000/home/foo/about`

Переадресация на адрес, который располагается уровнем выше:

```
1 app.use("/home/foo/bar",function (request, response) {
2   response.redirect("../about")
3 });
```

Передача данных приложению. Параметры строки запроса

Одним из способов передачи данных в приложение представляет использование параметров строки запроса. Строка запроса (query) - фактически это часть запрошенного адреса, которая идет после знака вопроса. Например, в запросе

http://localhost:3000/about?id=3&name=Tome

часть **id=3&name=Tome** представляет строку запроса.

В Express мы можем получить параметра строки запроса через свойство **query** объекта `request`, который передается в функцию обработки запроса. Например:

```
1  const express = require("express");
2
3  const app = express();
4  app.get("/", function(request, response){
5
6      response.send("<h1>Главная страница</h1>");
7  });
8  app.use("/about", function(request, response){
9
10     let id = request.query.id;
11     let userName = request.query.name;
12     response.send("<h1>Информация</h1><p>id=" + id + "</p><p>name=" + userName + "</p>");
13 });
14
15 app.listen(3000);
```

POST-запросы и отправка форм

При отправке каких-то сложных данных обычно используются формы. Рассмотрим, как получать отправленные данные в Express. Для получения данных форм из запроса необходимо использовать специальный пакет [body-parser](#). Поэтому вначале добавим его в проект с помощью команды `npm install body-parser --save`

В файле `app.js` определим следующий код:

```
1  const express = require("express");
2  const bodyParser = require("body-parser");
3
4  const app = express();
5
6  // создаем парсер для данных application/x-www-form-urlencoded
7  const urlencodedParser = bodyParser.urlencoded({extended: false});
8
9  app.get("/register", urlencodedParser, function (request, response) {
10     response.sendFile(__dirname + "/register.html");
11 });
12 app.post("/register", urlencodedParser, function (request, response) {
13     if(!request.body) return response.sendStatus(400);
14     console.log(request.body);
15     response.send(`${request.body.userName} - ${request.body.userAge}`);
16 });
17
18 app.get("/", function(request, response){
19     response.send("Главная страница");
20 });
```


Прежде всего для получения отправленных данных необходимо создать парсер:

```
const urlencodedParser = bodyParser.urlencoded({extended: false});
```

Поскольку данные отправляются с помощью формы, то для создания парсера применяется функция `urlencoded()`. В эту функцию передается объект, устанавливающий параметры парсинга. Значение `extended: false` указывает, что объект - результат парсинга будет представлять набор пар ключ-значение, а каждое значение может быть представлено в виде строки или массива.

Так как данные отправляются с помощью метода `POST`, то для обработки определяем функцию **`app.post("/register", ...)`**. Первый параметр функции - адрес, на который идет отправка - `"/register"`. Стоит отметить, что в данном случае с одним адресом `"/register"` связаны две функции, только одна обрабатывает запросы `get`, а другая - запросы `post`. Второй параметр - выше созданный парсер. Третий параметр - обработчик:

```
1 app.post("/register", urlencodedParser, function (request, response) {
2   if(!request.body) return response.sendStatus(400);
3   console.log(request.body);
4   response.send(`${request.body.userName} - ${request.body.userAge}`);
5 });
```

Параметры маршрута

Параметры маршрута представляют именованные сегменты URL-адреса. Не стоит их путать с параметрами строки запроса. Например:

```
1 localhost:3000/about/user?id=3&name=Tome
```

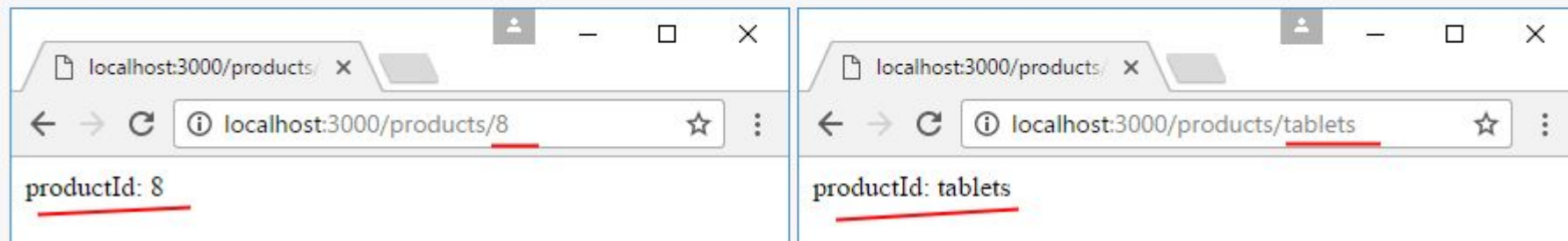
Здесь параметры строки запроса - это то, что идет после вопросительного знака - `id=3&name=Tome`. Остальная часть, которая идет до вопросительного знака может содержать параметры маршрута.

Название параметра должно включать символы из диапазона `[A-Za-z0-9_]`. В определении маршрута параметры предваряются знаком двоеточия:

```
1 const express = require("express");
2 const app = express();
3
4 app.get("/products/:productId", function (request, response) {
5   response.send("productId: " + request.params["productId"])
6 });
7
8 app.listen(3000);
```

В данном случае параметр называется "productId". Через коллекцию **request.params** можно получить все параметры и, в частности, значение параметра productId.

Если нам потребуется передать для этого параметра значение, то оно указывается в качестве последнего сегмента в строке запроса:



Router

Router позволяет определить дочерние подмаршруты со своими обработчиками относительно некоторого главного маршрута. Например, определим следующее приложение:

```
1  const express = require("express");
2  const app = express();
3
4  app.use("/about", function (request, response) {
5    response.send("О сайте");
6  });
7
8  app.use("/products/create",function (request, response) {
9    response.send("Добавление товара");
10 });
11 app.use("/products/:id",function (request, response) {
12   response.send(`Товар ${request.params.id}`);
13 });
14 app.use("/products/",function (request, response) {
15   response.send("Список товаров");
16 });
17
18 app.use("/", function (request, response) {
19   response.send("Главная страница");
20 });
21 app.listen(3000);
```

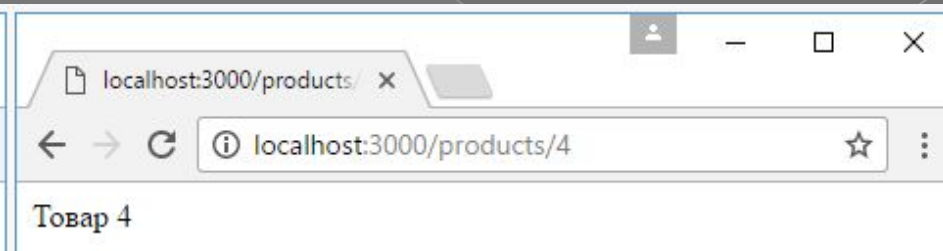
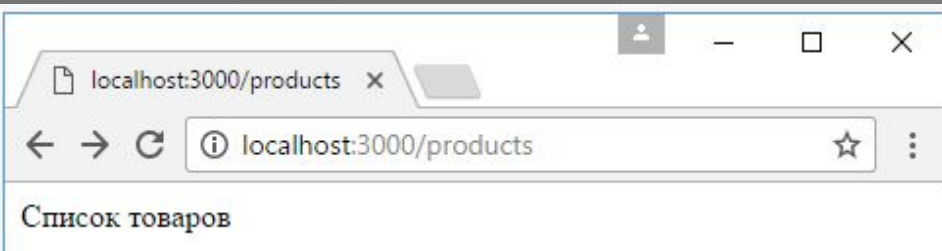
Здесь у нас есть пять маршрутов, которые обрабатываются различными обработчиками. Но три из этих маршрутов начинаются с `"/products"` и условно относятся к некоторому функционалу по работе с товарами (просмотр списка товаров, просмотр одного товара по `id` и добавление товара). Объект `Router` позволяет связать подобный функционал в одно целое и упростить управление им. Например, перепишем предыдущий пример с использованием объекта `Router`:

```

1  const express = require("express");
2  const app = express();
3
4  // определяем Router
5  const productRouter = express.Router();
6
7  // определяем маршруты и их обработчики внутри роутера
8  productRouter.use("/create", function(request, response){
9    response.send("Добавление товара");
10 });
11 productRouter.use("/:id", function(request, response){
12   response.send(`Товар ${request.params.id}`);
13 });
14 productRouter.use("/", function(request, response){
15   response.send("Список товаров");
16 });
17 // сопоставляем роутер с конечной точкой "/products"
18 app.use("/products", productRouter);
19
20 app.use("/about", function (request, response) {
21   response.send("О сайте");
22 });
23
24 app.use("/", function (request, response) {
25   response.send("Главная страница");
26 });
27 app.listen(3000);

```

Здесь определен объект `productRouter`, который обрабатывает все запросы по маршруту `"/products"`. Это главный маршрут. Однако в рамках этого маршрута может быть подмаршрут `"/"` со своим обработчиком, а также подмаршруты `"/:id"` и `"/create"`, которые также имеют свои обработчики.



Начало работы с MongoDB



Наиболее популярной системой управления базами данных для Node.js на данный момент является MongoDB. Для работы с этой платформой прежде всего необходимо установить сам сервер MongoDB.

При подключении и взаимодействии с бд в MongoDB можно выделить следующие этапы:

- * Подключение к серверу
- * Получение объекта базы данных на сервере
- * Получение объекта коллекции в базе данных
- * Взаимодействие с коллекцией (добавление, удаление, получение, изменение данных)

Ключевым классом для работы с MongoDB является класс **MongoClient**, и через него будет идти все взаимодействие с хранилищем данных. Соответственно вначале мы должны получить MongoClient:

```
const MongoClient = require("mongodb").MongoClient;
```

Для подключения к серверу mongodb применяется метод **connect()**:

```
1  const MongoClient = require("mongodb").MongoClient;
2
3  // создаем объект MongoClient и передаем ему строку подключения
4  const mongoClient = new MongoClient("mongodb://localhost:27017/", { useNewUrlParser: true });
5  mongoClient.connect(function(err, client){
6
7      if(err){
8          return console.log(err);
9      }
10     // взаимодействие с базой данных
11     client.close();
12 });
```

Получив объект подключенного клиента, мы можем обращаться к базе данных на сервере. Для этого используется метод **client.db(назв бд)**

База данных в MongoDB не имеет таблиц.

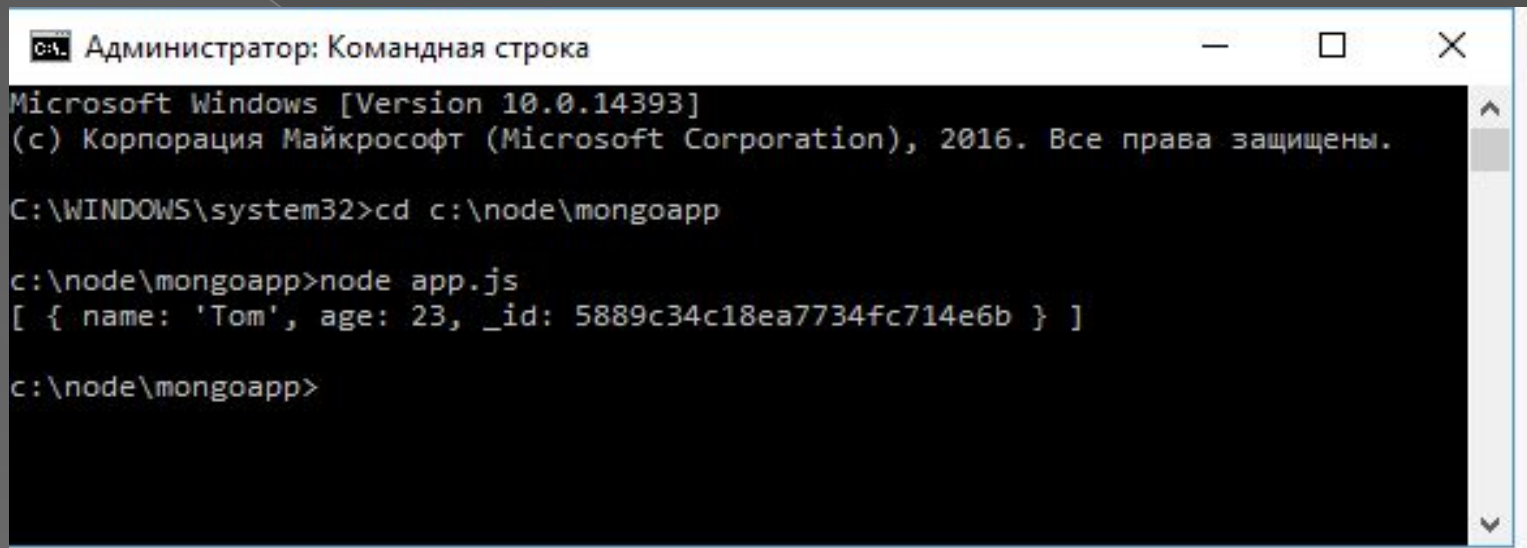
* Вместо этого все данные попадают в коллекции.

* В рамках node.js для взаимодействия с базой данных (добавления, удаления, чтения данных) нам потребуется получить объект коллекции. Для этого применяется метод **db.collection("название_коллекции")**, в который передается название коллекции.

* В отличие от таблиц в реляционных системах, где все данные хранятся в виде строк, в коллекциях в MongoDB данные хранятся в виде документов. Например, добавим в базу данных один документ. Для этого определим в каталоге проекта следующий файл **app.js**:

```
1  const MongoClient = require("mongodb").MongoClient;
2
3  const url = "mongodb://localhost:27017/";
4  const mongoClient = new MongoClient(url, { useNewUrlParser: true });
5
6  mongoClient.connect(function(err, client){
7
8      const db = client.db("usersdb");
9      const collection = db.collection("users");
10     let user = {name: "Tom", age: 23};
11     collection.insertOne(user, function(err, result){
12
13         if(err){
14             return console.log(err);
15         }
16         console.log(result.ops);
17         client.close();
18     });
19 });
```

В качестве базы данных здесь используется "usersdb". При этом не важно, что по умолчанию на сервере MongoDB нет подобной базы данных. При первом к ней обращении сервер автоматически ее создаст.



```
Администратор: Командная строка
Microsoft Windows [Version 10.0.14393]
(c) Корпорация Майкрософт (Microsoft Corporation), 2016. Все права защищены.

C:\WINDOWS\system32>cd c:\node\mongoapp

c:\node\mongoapp>node app.js
[ { name: 'Tom', age: 23, _id: 5889c34c18ea7734fc714e6b } ]

c:\node\mongoapp>
```


Добавление и получение данных в MongoDB

Для добавления мы можем использовать различные методы. Если нужно добавить один объект, то применяется метод **insertOne()**. При добавлении набора объектов можно использовать метод **insertMany()**.

```
1 const MongoClient = require("mongodb").MongoClient;
2
3 const url = "mongodb://localhost:27017/";
4 const mongoClient = new MongoClient(url, { useNewUrlParser: true });
5
6 let users = [{name: "Bob", age: 34}, {name: "Alice", age: 21}, {name: "Tom", age: 45}];
7
8 mongoClient.connect(function(err, client){
9
10     const db = client.db("usersdb");
11     const collection = db.collection("users");
12
13     collection.insertMany(users, function(err, results){
14
15         console.log(results);
16         client.close();
```

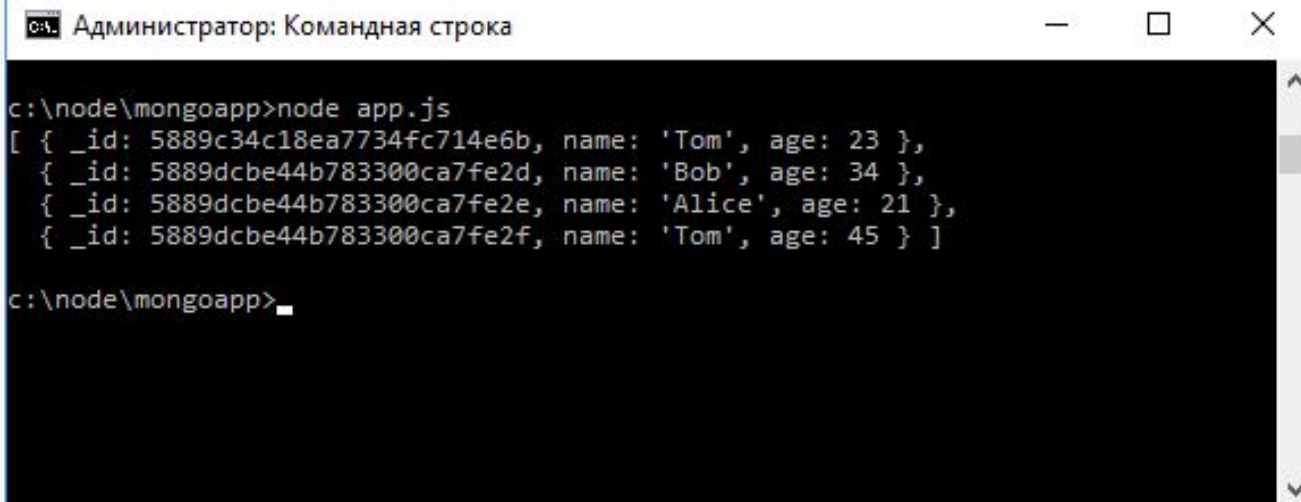
Администратор: Командная строка

```
c:\node\mongoapp>node app.js
{ result: { ok: 1, n: 3 },
  ops:
    [ { name: 'Bob', age: 34, _id: 5889dcbe44b783300ca7fe2d },
      { name: 'Alice', age: 21, _id: 5889dcbe44b783300ca7fe2e },
      { name: 'Tom', age: 45, _id: 5889dcbe44b783300ca7fe2f } ],
  insertedCount: 3,
  insertedIds:
    [ 5889dcbe44b783300ca7fe2d,
      5889dcbe44b783300ca7fe2e,
      5889dcbe44b783300ca7fe2f ] }

c:\node\mongoapp>
```

Получение данных

```
1 const MongoClient = require("mongodb").MongoClient;
2
3 const url = "mongodb://localhost:27017/";
4 const mongoClient = new MongoClient(url, { useNewUrlParser: true });
5
6 mongoClient.connect(function(err, client){
7
8     const db = client.db("usersdb");
9     const collection = db.collection("users");
10
11     if(err) return console.log(err);
12
13     collection.find().toArray(function(err, results){
14
15         console.log(results);
16         client.close();
17     });
```



Администратор: Командная строка

```
c:\node\mongoapp>node app.js
[ { _id: 5889c34c18ea7734fc714e6b, name: 'Tom', age: 23 },
  { _id: 5889dcbe44b783300ca7fe2d, name: 'Bob', age: 34 },
  { _id: 5889dcbe44b783300ca7fe2e, name: 'Alice', age: 21 },
  { _id: 5889dcbe44b783300ca7fe2f, name: 'Tom', age: 45 } ]

c:\node\mongoapp>
```

Удаление документов в MongoDB

Удалять документы в MongoDB можно различными способами. Здесь надо отметить следующие методы коллекции:

deleteMany(): удаляет все документы, которые соответствуют определенному критерию

deleteOne(): удаляет один документ, который соответствует определенному критерию

findOneAndDelete(): получает и удаляет один документ, который соответствует определенному критерию

drop(): удаляет всю коллекцию

Удалим всех пользователей, у которых имя "Tom":

Удалим всех пользователей, у которых имя "Tom":

```
1  const MongoClient = require("mongodb").MongoClient;
2
3  const url = "mongodb://localhost:27017/";
4  const mongoClient = new MongoClient(url, { useNewUrlParser: true });
5
6  mongoClient.connect(function(err, client){
7
8      if(err) return console.log(err);
9
10     const db = client.db("usersdb");
11     db.collection("users").deleteMany({name: "Tom"}, function(err, result){
12
13         console.log(result);
14         client.close();
15     });
16 });
```

Маршрутизация

При обработке запросов фреймворк Express опирается на систему маршрутизации. В приложении определяются маршруты, а также обработчики этих маршрутов. Если запрос соответствует определенному маршруту, то вызывается для обработки запроса соответствующий обработчик. Для обработки данных по определенному маршруту можно использовать ряд функций, в частности:

- use
- get
- post
- put
- delete

В качестве первого параметра эти функции могут принимать шаблон адреса, запрос по которому будет обрабатываться. Вторым параметром функций представляет функцию, которая будет обрабатывать запрос по совпавшему с шаблоном адресу. Например:

```
1  const express = require("express");
2  const app = express();
3
4  // обработка запроса по адресу /about
5  app.get("/about", function(request, response){
6
7      response.send("<h1>О сайте</h1>");
8  });
9
10 // обработка запроса по адресу /contact
11 app.use("/contact", function(request, response){
12
13     response.send("<h1>Контакты</h1>");
14 });
15
16 // обработка запроса к корню веб-сайта
17 app.get("/", function(request, response){
18
19     response.send("<h1>Главная страница</h1>");
20 });
21 app.listen(3000);
```

Обновление документов в MongoDB

Для обновления элементов в MongoDB есть несколько методов:

- * updateOne
- * updateMany
- * findOneAndUpdate()

Метод **findOneAndUpdate()** обновляет один элемент. Он принимает следующие параметры:

Критерий фильтрации документа, который надо обновить

Параметр обновления

Дополнительные опции обновления, которые по умолчанию имеют значение null

Функция обратного вызова, которая выполняется при обновлении

```
1  const MongoClient = require("mongodb").MongoClient;
2
3  const url = "mongodb://localhost:27017/";
4  const mongoClient = new MongoClient(url, { useNewUrlParser: true });
5
6  let users = [{name: "Bob", age: 34} , {name: "Alice", age: 21}, {name: "Tom", age: 45}];
7  mongoClient.connect("mongodb://localhost:27017/", function(err, client){
8
9      if(err) return console.log(err);
10
11     const db = client.db("usersdb");
12     const col = db.collection("users");
13     col.insertMany(users, function(err, results){
14
15         col.findOneAndUpdate(
16             {age: 21}, // критерий выборки
17             { $set: {age: 25}}, // параметр обновления
18             function(err, result){
19
20                 console.log(result);
21                 client.close();
```

Mongoose

Mongoose представляет специальную ODM-библиотеку (Object Data Modelling) для работы с MongoDB, которая позволяет сопоставлять объекты классов и документы коллекций из базы данных. Грубо говоря, Mongoose работает подобно инструментам ORM



```
const User = mongoose.model("User", userScheme);
```

```
1  const mongoose = require("mongoose");
2  const Schema = mongoose.Schema;
3
4  // установка схемы
5  const userScheme = new Schema({
6    name: String,
7    age: Number
8  });
9
10 // подключение
11 mongoose.connect("mongodb://localhost:27017/usersdb", { useNewUrlParser: true });
12
13 const User = mongoose.model("User", userScheme);
14 const user = new User({
15   name: "Bill",
16   age: 41
17 });
18
19 user.save(function(err){
20   mongoose.disconnect(); // отключение от базы данных
21
```

Первый параметр в методе **mongoose.model** указывает на название модели. Mongoose затем будет автоматически искать в базе данных коллекцию, название которой соответствует названию модели во множественном числе.

Второй параметр функции **mongoose.model** - собственно схема.

Определение схемы в Mongoose

Схема в Mongoose определяет метаданные модели - ее свойства, типы данных и ряд другой информации.

В прошлой теме схема определялась следующим образом:

```
1 const userScheme = new Schema({
2   name: String,
3   age: Number
4 });
```

Если свойство представляет сложный объект, то в качестве типа указываем определение этого объекта:

```
1 const userScheme = new Schema({
2   name: String,
3   age: Number,
4   company: {
5     name: String,
6     employee: [String], // тип - массив строк
7     date: Date
8   }
9 });
```


Валидация

- **required:** требует обязательного наличия значения для свойства
- **min** и **max:** задают минимальное и максимальное значения для числовых данных
- **minlength** и **maxlength:** задают минимальную и максимальную длину для строк
- **enum:** строка должна представлять одно из значений в указанном массиве строк
- **match:** строка должна соответствовать регулярному выражению

```
// установка схемы
const userScheme = new Schema({
  name: {
    type: String,
    required: true,
    minlength: 3,
    maxlength: 20
  },
  age: {
    type: Number,
    required: true,
    min: 1,
    max: 100
  }
});
```

Итоги

Здесь мы рассмотрели лишь основные механизмы библиотеки `express.js`, те, которые ответственны за работу веб-сервера, но эта библиотека обладает и многими другими возможностями. Мы не останавливались на проверках, которые проходят запросы до поступления их в обработчики, мы не говорили о вспомогательных методах, которые доступны при работе с переменными `res` и `req`. И, наконец, мы не затрагивали одну из наиболее мощных возможностей `express`. Она заключается в использовании промежуточного программного обеспечения, которое может быть направлено на решение практически любых задач — от разбора запросов до реализации полноценной системы аутентификации.

Надеемся, этот материал помог вам разобраться в основных особенностях устройства `express`, и теперь вы, при необходимости, сможете понять всё остальное, самостоятельно проанализировав интересующие вас части исходного кода этой библиотеки.

