Node JS & Express Framework

Что такое среда разработки Express?

Среда разработки Express включает в себя установку Nodejs, менеджера пакетов NPM и (опционально) Express 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.

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):

```
> express --help

  Usage: express [options] [dir]

  Options:

    -h, --help          output usage information
        --version       output the version number
    -e, --ejs           add ejs engine support
        --pug           add pug engine support
        --hbs           add handlebars engine support
    -H, --hogan         add hogan.js engine support
    -v, --view <engine> add view <engine> support (ejs|hbs|hjs|jade|pug|twig|vash) (defaults to jade)
    -c, --css <engine>  add stylesheet <engine> support (less|stylus|compass|sass) (defaults to plain css)
        --git           add .gitignore
    -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:

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

Генератор создаст (и перечислит) файлы проекта

```
create : express-locallibrary-tutorial
   create : express-locallibrary-tutorial/package.json
   create : express-locallibrary-tutorial/app.js
   create : express-locallibrary-tutorial/public/images
   create : express-locallibrary-tutorial/public
   create : express-locallibrary-tutorial/public/stylesheets
   create : express-locallibrary-tutorial/public/stylesheets/style.css
   create : express-locallibrary-tutorial/public/javascripts
   create : express-locallibrary-tutorial/routes
   create : express-locallibrary-tutorial/routes/index.js
   create : express-locallibrary-tutorial/routes/users.js
   create : express-locallibrary-tutorial/views
   create : express-locallibrary-tutorial/views/index.pug
   create : express-locallibrary-tutorial/views/layout.pug
   create : express-locallibrary-tutorial/views/error.pug
   create : express-locallibrary-tutorial/bin
   create : express-locallibrary-tutorial/bin/www

   install dependencies:
     > cd express-locallibrary-tutorial && npm install

   run the app:
     > 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)

```
cd express-locallibrary-tutorial
npm install
```

- Затем запустите приложение.

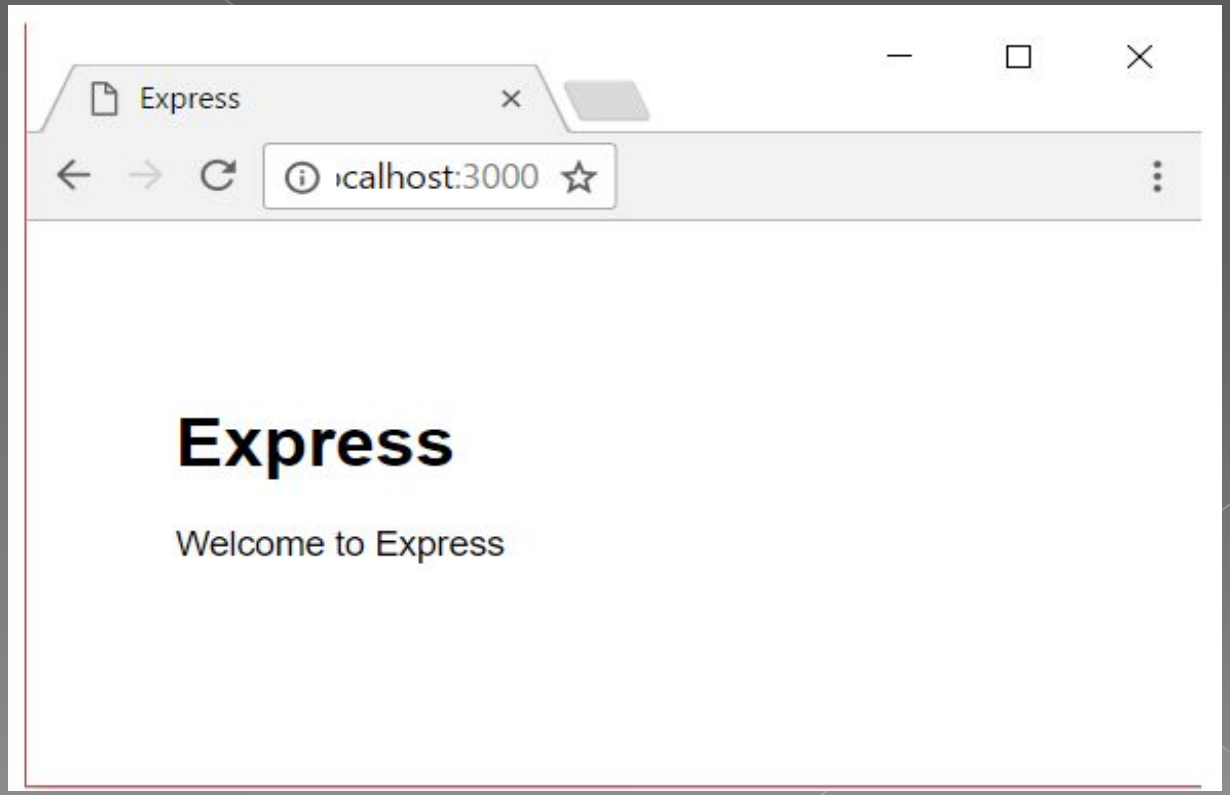
В Windows используйте следующую команду:

```
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

```
const express = require("express");

const app = express();
app.get("/", function(request, response){

    response.send("<h1>Главная страница</h1>");
});
app.get("/about", function(request, response){

    response.send("<h1>О сайте</h1>");
});
app.get("/contact", function(request, response){

    response.send("<h1>Контакты</h1>");
});
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().

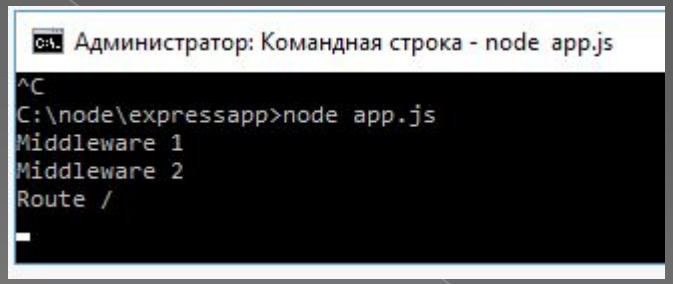
При необходимости мы можем встроить в конвейер обработки запроса на любом этапе любую функцию middleware. Для этого применяется метод **app.use()**. Так, изменим файл app.js следующим образом:

```
const express = require("express");

const app = express();
app.use(function(request, response, next){

    console.log("Middleware 1");
    next();
});
app.use(function(request, response, next){

    console.log("Middleware 2");
    next();
});

app.get("/", function(request, response){

    console.log("Route /");
    response.send("Hello");
});
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().

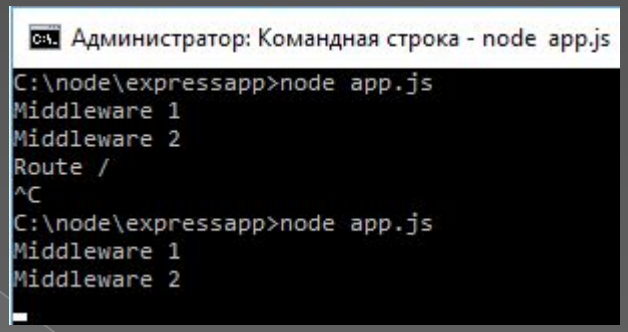
При запуске приложения после обращения по адресу "http://localhost:3000/" последовательно отработают все три middleware:



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

```
const express = require("express");

const app = express();
app.use(function(request, response, next){

    console.log("Middleware 1");
    next();
});
app.use(function(request, response, next){

    console.log("Middleware 2");
    response.send("Middleware 2");
});

app.get("/", function(request, response){
    console.log("Route /");
    response.send("Hello");
});
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:

```
const express = require("express");
const fs = require("fs");

const app = express();
app.use(function(request, response, next){

    let now = new Date();
    let hour = now.getHours();
    let minutes = now.getMinutes();
    let seconds = now.getSeconds();
    let data = `${hour}:${minutes}:${seconds} ${request.method} ${request.url} ${request.get("user-agent")}`;
    console.log(data);
    fs.appendFile("server.log", data + "\n", function(){});
    next();
});

app.get("/", function(request, response){
    response.send("Hello");
});
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.

```
const express = require("express");
const app = express();

app.use(function (request, response) {
    response.send("<h2>Hello</h2>");
});

app.listen(3000);
```

Отправка объекта:

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

Отправка массива:

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

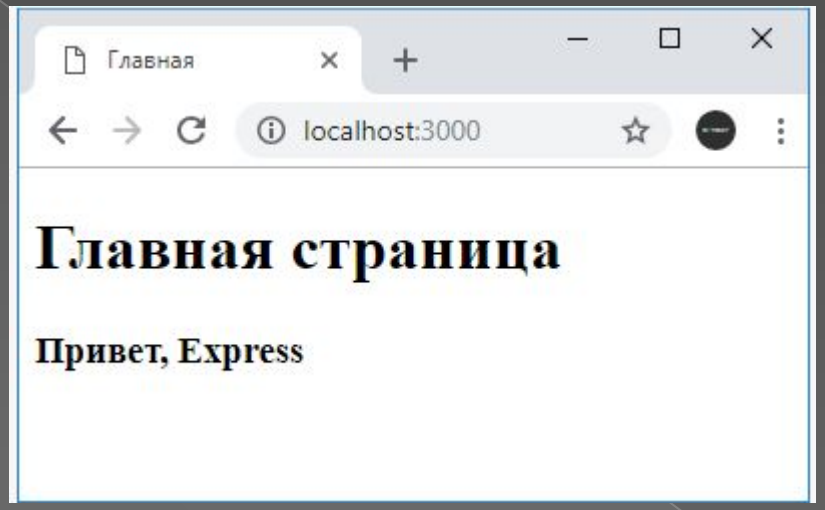
Отправка объекта:

```
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 есть отправляемый код html довольно большой, то соответственно код приложения тоже становится громоздким. Например, мы можем написать так:




```
const express = require("express");
const app = express();

app.use(function (request, response) {
  response.send(`<!DOCTYPE html>
  <html>
  <head>
      <title>Главная</title>
      <meta charset="utf-8" />
  </head>
  <body>
      <h1>Главная страница</h1>
      <h3>Привет, Express</h3>
  </body>
  <html>`);
});

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**:

```html
<!DOCTYPE html>
<html>
  <head>
      <title>Главная</title>
      <meta charset="utf-8" />
  </head>
  <body>
      <h1>Главная страница</h1>
      <h3>Привет, Express</h3>
  </body>
<html>
```

Отправим этот файл с помощью функции sendFile:

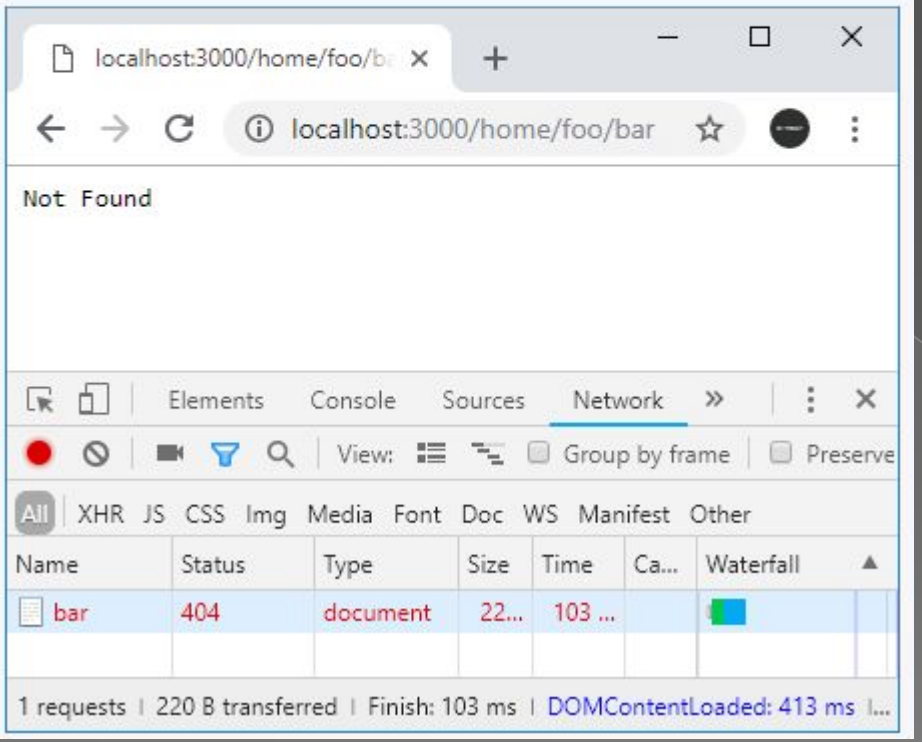
```
const express = require("express");
const app = express();

app.use(function (request, response) {
  response.sendFile(__dirname + "/index.html");
});

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:

```
const express = require("express");
const app = express();

app.use("/home/foo/bar",function (request, response) {
  response.sendStatus(404)
});

app.listen(3000);
```



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()**:

```
const express = require("express");
const app = express();

app.use("/home/foo/bar",function (request, response) {
  response.status(404).send(`Ресурс не найден`);
});

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. Второй параметр функций представляет функцию, которая будет обрабатывать запрос по совпавшему с шаблоном адресу. Например:

```
const express = require("express");
const app = express();

// обработка запроса по адресу /about
app.get("/about", function(request, response){

    response.send("<h1>О сайте</h1>");
});

// обработка запроса по адресу /contact
app.use("/contact", function(request, response){

    response.send("<h1>Контакты</h1>");
});

// обработка запроса к корню веб-сайта
app.get("/", function(request, response){

    response.send("<h1>Главная страница</h1>");
});
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:

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

Такой маршрут будет соответствовать строке запроса "/bk" или "/bok".
Символ + указывает, что предыдущий символ может встречаться 1 и более раз:

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

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

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

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

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

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

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

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":

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

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 оносительным 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:

```
const express = require("express");
const app = express();

app.use("/home",function (request, response) {
  response.redirect("about")
});
app.use("/about", function (request, response) {
  response.send("<h1>About</h1>");
});

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:

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

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

Переадресация на адрес, который располагается уровнем выше:

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

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:

```
const express = require("express");

const app = express();
app.get("/", function(request, response){

    response.send("<h1>Главная страница</h1>");
});
app.use("/about", function(request, response){

    let id = request.query.id;
    let userName = request.query.name;
    response.send("<h1>Информация</h1><p>id=" + id +"</p><p>name=" + userName + "</p>");
});

app.listen(3000);
```

POST-запросы и отправка форм

При оправке 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:

```
const express = require("express");
const bodyParser = require("body-parser");

const app = express();

// создаем парсер для данных application/x-www-form-urlencoded
const urlencodedParser = bodyParser.urlencoded({extended: false});

app.get("/register", urlencodedParser, function (request, response) {
    response.sendFile(__dirname + "/register.html");
});
app.post("/register", urlencodedParser, function (request, response) {
    if(!request.body) return response.sendStatus(400);
    console.log(request.body);
    response.send(`${request.body.userName} - ${request.body.userAge}`);
});

app.get("/", function(request, response){
    response.send("Главная страница");
});
```

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 одинм адресом "/register" связаны две функции, только одна обрабатывает запросы get, а другая - запросы post. Второй параметр - выше созданный парсер. Третий параметр - обработчик:

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

Параметры маршрута

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

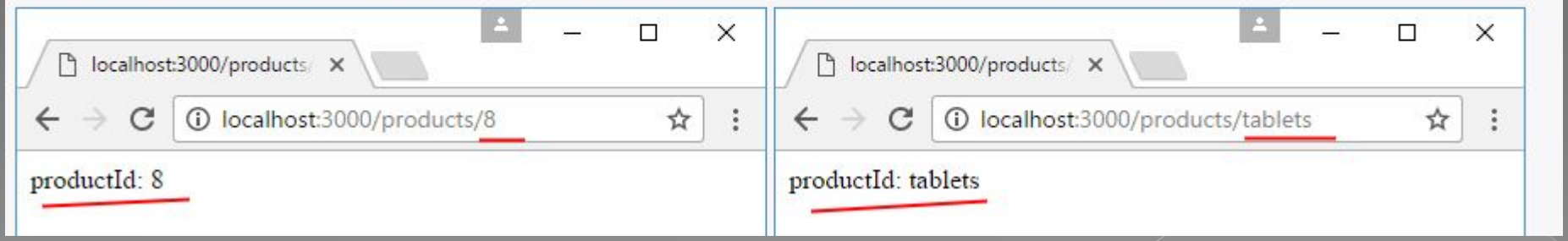
```
localhost:3000/about/user?id=3&name=Tome
```

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:

```
const express = require("express");
const app = express();

app.get("/products/:productId", function (request, response) {
  response.send("productId: " + request.params["productId"])
});

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

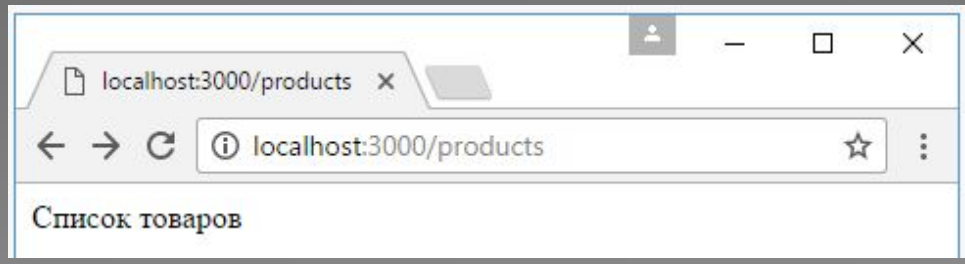
Router позволяет определить дочерние подмаршруты со своими обработчиками относительно некоторого главного маршрута. Например, определим следующее приложение:

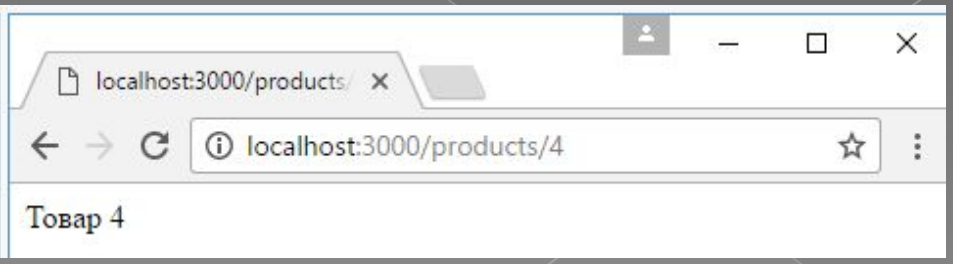
```
const express = require("express");
const app = express();

app.use("/about", function (request, response) {
  response.send("О сайте");
});

app.use("/products/create",function (request, response) {
  response.send("Добавление товара");
});
app.use("/products/:id",function (request, response) {
  response.send(`Товар ${request.params.id}`);
});
app.use("/products/",function (request, response) {
  response.send("Список товаров");
});

app.use("/", function (request, response) {
  response.send("Главная страница");
});
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:

```
const express = require("express");
const app = express();

// определяем Router
const productRouter = express.Router();

// определяем маршруты и их обработчики внутри роутера
productRouter.use("/create", function(request, response){
  response.send("Добавление товара");
});
productRouter.use("/:id", function(request, response){
  response.send(`Товар ${request.params.id}`);
});
productRouter.use("/", function(request, response){
  response.send("Список товаров");
});
// сопоставляем роутер с конечной точкой "/products"
app.use("/products", productRouter);

app.use("/about", function (request, response) {
  response.send("О сайте");

app.use("/", function (request, response) {
  response.send("Главная страница");
});
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 взаимодействия с хранилищем данных. Соответственно вначале мы должны получить MongoClient:

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

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

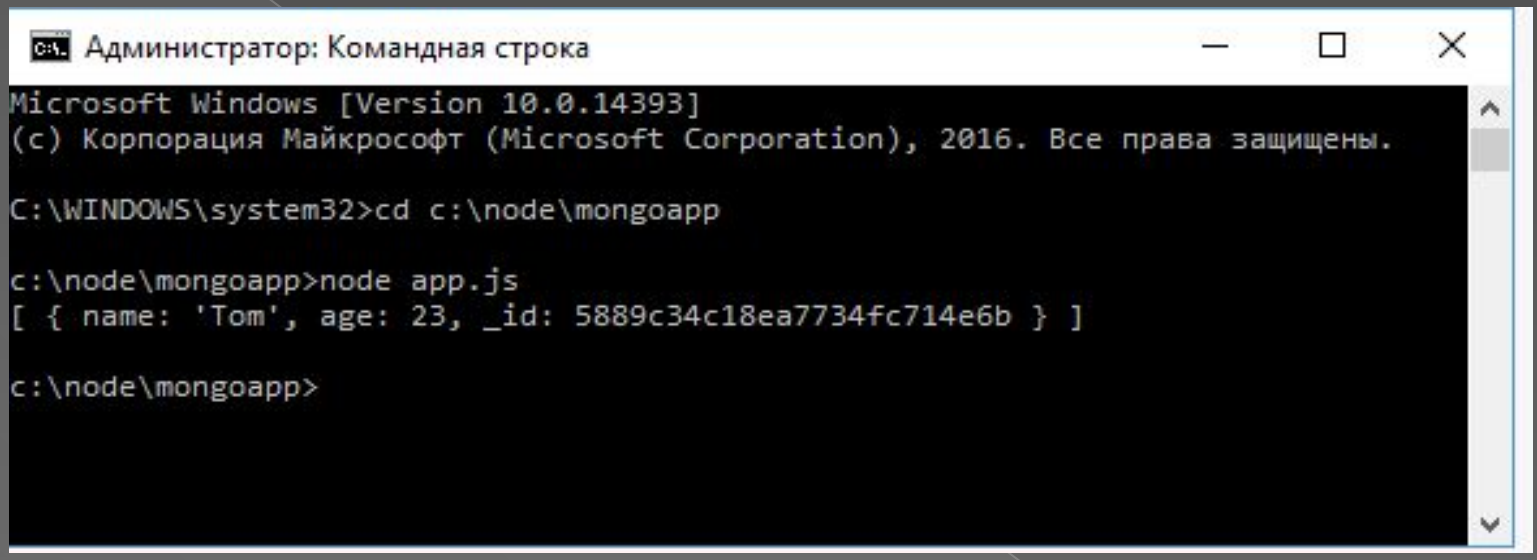
```
1  const MongoClient = require("mongodb").MongoClient;
2
3  // создаем объект MongoClient и передаем ему строку подключения
4  const mongoClient = new MongoClient("mongodb://localhost:27017/", { useNewUrlParser: true });
5  mongoClient.connect(function(err, client){
6
7      if(err){
8          return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

Получив объект подлюченного 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**:

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

const url = "mongodb://localhost:27017/";
const mongoClient = new MongoClient(url, { useNewUrlParser: true });

mongoClient.connect(function(err, client){

    const db = client.db("usersdb");
    const collection = db.collection("users");
    let user = {name: "Tom", age: 23};
    collection.insertOne(user, function(err, result){

        if(err){
            return console.log(err);
        }
        console.log(result.ops);
        client.close();
    });
});
```

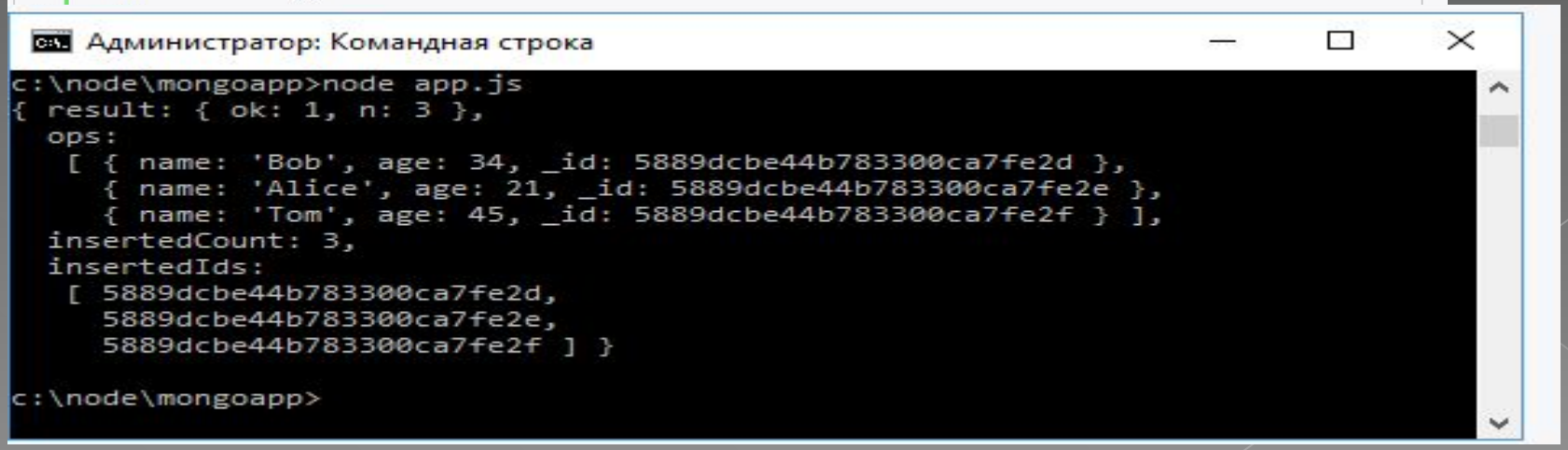
В качестве базы данных здесь используется "usersdb". При этом не важно, что по умолчанию на сервере MongoDB нет подобной базы данных. При первом к ней обращении сервер автоматически ее создаст.



Добавление и получение данных в MongoDB

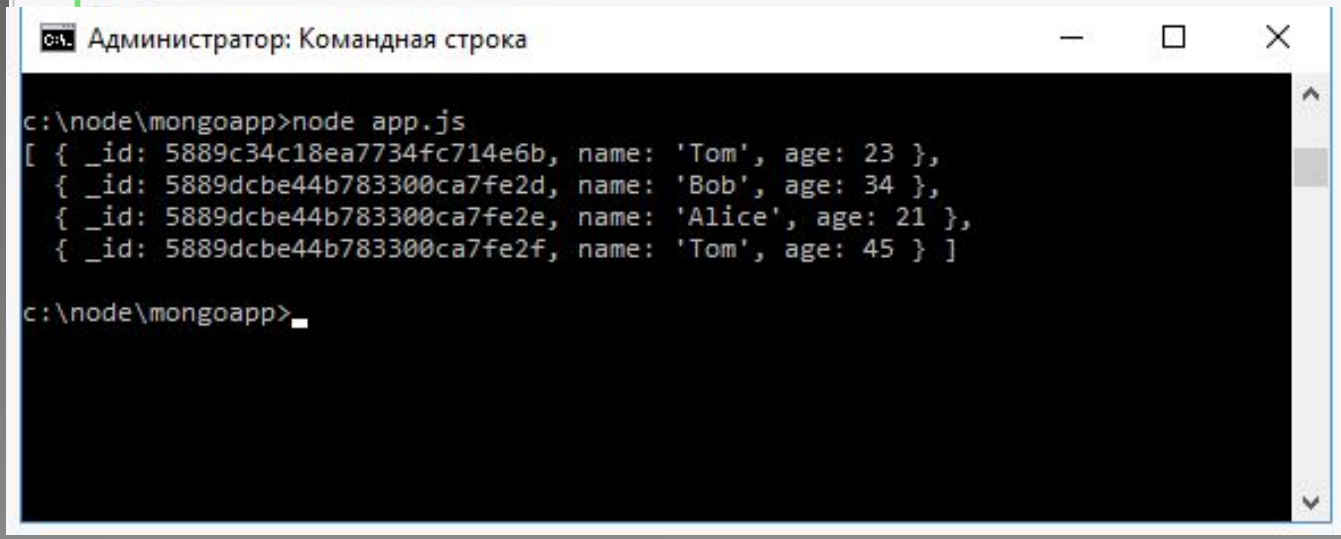
Для добавления мы можем использовать различные методы. Если нужно добавить один объект, то применяется метод **insertOne()**. При добавлении набора объектов можно использовать метод **insertMany()**.

```
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();
```



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     });
```



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":

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

const url = "mongodb://localhost:27017/";
const mongoClient = new MongoClient(url, { useNewUrlParser: true });

mongoClient.connect(function(err, client){

    if(err) return console.log(err);

    const db = client.db("usersdb");
    db.collection("users").deleteMany({name: "Tom"}, function(err, result){

        console.log(result);
        client.close();
    });
});
```

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. Второй параметр функций представляет функцию, которая будет обрабатывать запрос по совпавшему с шаблоном адресу. Например:

```
const express = require("express");
const app = express();

// обработка запроса по адресу /about
app.get("/about", function(request, response){

    response.send("<h1>О сайте</h1>");
});

// обработка запроса по адресу /contact
app.use("/contact", function(request, response){

    response.send("<h1>Контакты</h1>");
});

// обработка запроса к корню веб-сайта
app.get("/", function(request, response){

    response.send("<h1>Главная страница</h1>");
});
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

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

const url = "mongodb://localhost:27017/";
const mongoClient = new MongoClient(url, { useNewUrlParser: true });

let users = [{name: "Bob", age: 34} , {name: "Alice", age: 21}, {name: "Tom", age: 45}];
mongoClient.connect("mongodb://localhost:27017/", function(err, client){

    if(err) return console.log(err);

    const db = client.db("usersdb");
    const col = db.collection("users");
    col.insertMany(users, function(err, results){

        col.findOneAndUpdate(
            {age: 21}, // критерий выборки
            { $set: {age: 25}}, // параметр обновления
            function(err, result){

                console.log(result);
                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);
```

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

// установка схемы
const userScheme = new Schema({
    name: String,
    age: Number
});

// подключение
mongoose.connect("mongodb://localhost:27017/usersdb", { useNewUrlParser: true });

const User = mongoose.model("User", userScheme);
const user = new User({
    name: "Bill",
    age: 41
});

user.save(function(err){
    mongoose.disconnect();  // отключение от базы данных

```

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:

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

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

```
const userScheme = new Schema({
    name: String,
    age: Number,
    company: {
        name: String,
        employee: [String], // тип - массив строк
        date: Date
    }
});
```

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 задача — 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.