Оглавление

[Интро 5](#_Toc72321295)

[Какие темы вы пройдёте: 5](#_Toc72321296)

[Какие навыки получите: 5](#_Toc72321297)

[Введение в серверную разработку 5](#_Toc72321298)

[Что такое сервер? 6](#_Toc72321299)

[На каком языке пишут серверный код? 6](#_Toc72321300)

[Зачем изобрели Node.js? 7](#_Toc72321301)

[За гранью браузера: что такое Node.js? 7](#_Toc72321302)

[Что такое Node.js? 8](#_Toc72321303)

[Как работает Node.js? 8](#_Toc72321304)

[Начинаем кодить 9](#_Toc72321305)

[Нода — не браузер 10](#_Toc72321306)

[Сервер на Node.js 11](#_Toc72321307)

[Подключите API 11](#_Toc72321308)

[Создайте сервер 11](#_Toc72321309)

[Настройте порт 11](#_Toc72321310)

[Установите ответ 12](#_Toc72321311)

[Настройте запрос и ответ 12](#_Toc72321312)

[Объект запроса 13](#_Toc72321313)

[Объект ответа 13](#_Toc72321314)

[Пользуйтесь переменными окружения 15](#_Toc72321315)

[Тело запроса: потоки 15](#_Toc72321316)

[Как обрабатываются данные запроса? 15](#_Toc72321317)

[Как собрать данные запроса? 16](#_Toc72321318)

[Что обрабатывать, когда приходит блок? Событийная модель 16](#_Toc72321319)

[Система модулей Node.js 17](#_Toc72321320)

[Как импортировать модули? 17](#_Toc72321321)

[Как экспортировать модули? 18](#_Toc72321322)

[Как называть модули? 19](#_Toc72321323)

[ES6 модули работают в ноде? 19](#_Toc72321324)

[Работа с файловой системой 20](#_Toc72321325)

[Какой модуль нужен 20](#_Toc72321326)

[Промисы при работе с файлами 21](#_Toc72321327)

[Проблемы с путями 21](#_Toc72321328)

[Что хранится в модуле 23](#_Toc72321329)

[Где находится модуль 23](#_Toc72321330)

[Как преобразовать путь 23](#_Toc72321331)

[Что ещё может модуль fs? 24](#_Toc72321332)

[Читать все файлы директории 24](#_Toc72321333)

[Создавать папки 25](#_Toc72321334)

[Записывать данные в файл 25](#_Toc72321335)

[Удалять файлы 25](#_Toc72321336)

[Много чего ещё 26](#_Toc72321337)

[Потоки для чтения и записи файлов 26](#_Toc72321338)

[Какие бывают потоки и как ими пользоваться? 26](#_Toc72321339)

[Как работать с потоками? 27](#_Toc72321340)

[Почему не воспользоваться методами чтения и записи файлов? 28](#_Toc72321341)

[Метод pipe для работы с потоками 28](#_Toc72321342)

[Отладка Node.js приложения 29](#_Toc72321343)

[Отладка в браузере 29](#_Toc72321344)

[Тестирование сервера: Postman 31](#_Toc72321345)

[Инструмент для тестирования API 31](#_Toc72321346)

[Редактирование запроса 33](#_Toc72321347)

[Компьютерные сети 35](#_Toc72321348)

[Интернет с точки зрения компьютера 36](#_Toc72321349)

[Как найти друг друга в сети? IP-адрес 37](#_Toc72321350)

[Неравенство адресов: публичные и приватные IP 38](#_Toc72321351)

[Где взять IP-адрес? DHCP-службы 38](#_Toc72321352)

[Поиск адреса по имени. DNS 39](#_Toc72321353)

[Новый адрес в интернете: пусть мир узнает обо мне 39](#_Toc72321354)

[Маршрут построен 40](#_Toc72321355)

[Протокол передачи данных TCP 40](#_Toc72321356)

[TCP/IP модель 41](#_Toc72321357)

[Заключение 42](#_Toc72321358)

[Express. Введение 42](#_Toc72321359)

[Приложение на экспресс 43](#_Toc72321360)

[Устанавливаем экспресс 43](#_Toc72321361)

[Настраиваем hot reload 43](#_Toc72321362)

[Устанавливаем точку входа 44](#_Toc72321363)

[Проверяем, всё ли работает 44](#_Toc72321364)

[Как общаются клиент и сервер? Настройка роутинга 45](#_Toc72321365)

[Роутинг GET-запроса 45](#_Toc72321366)

[Объект ответа 45](#_Toc72321367)

[Объект запроса 46](#_Toc72321368)

[Обработка методов в express 46](#_Toc72321369)

[Все говорят о роутинге. Объясняем, что это 47](#_Toc72321370)

[Что такое динамические роуты? 47](#_Toc72321371)

[Параметры роута 47](#_Toc72321372)

[Как организовать роутинг 48](#_Toc72321373)

[Промежуточная обработка. Как улучшить код посредством middlewares 51](#_Toc72321374)

[Создаём мидлвэр 52](#_Toc72321375)

[Подключаем мидлвэр 53](#_Toc72321376)

[Продвинутые мидлвэры: парсер данных 54](#_Toc72321377)

[Сборка пакетов: body-parser 54](#_Toc72321378)

[Подключим к проекту 54](#_Toc72321379)

[Отдача html и статичных файлов в Express 55](#_Toc72321380)

[Кеширование ответа сервера в express 56](#_Toc72321381)

[Кеширующие заголовки 56](#_Toc72321382)

[Кеширующие заголовки в express 57](#_Toc72321383)

[CORS. Обработка ошибок 58](#_Toc72321384)

[Как разрешить запросы с нескольких ресурсов 58](#_Toc72321385)

[Заключение 59](#_Toc72321386)

[Роутеры 59](#_Toc72321387)

[Мидлвэры 60](#_Toc72321388)

[Готовые пакеты 60](#_Toc72321389)

[Дополнительные материалы 61](#_Toc72321390)

[Как решить проблемы совместимости JavaScript 61](#_Toc72321391)

[Особенности асинхронности в Node.js 61](#_Toc72321392)

[Как передавать огромные файлы 61](#_Toc72321393)

[Как выстроить сложную логику роутинга 61](#_Toc72321394)

[Введение. Зачем нужны базы данных? 61](#_Toc72321395)

[Какие бывают базы. SQL и NoSQL 62](#_Toc72321396)

[Реляционные БД 63](#_Toc72321397)

[Нереляционные БД 64](#_Toc72321398)

[Какую базу выбрать? 68](#_Toc72321399)

[MongoDB 68](#_Toc72321400)

[Почему NoSQL? 68](#_Toc72321401)

[Почему MongoDB? 68](#_Toc72321402)

[Установка Mongo 68](#_Toc72321403)

[Windows 68](#_Toc72321404)

[macOS 74](#_Toc72321405)

[Linux 75](#_Toc72321406)

[Взаимодействие с Mongo 75](#_Toc72321407)

[Как взаимодействовать с MongoDB 75](#_Toc72321408)

[Графический интерфейс 75](#_Toc72321409)

[Дополнительные Ссылки 80](#_Toc72321410)

[Подключение к mongo из JavaScript. Mongoose 80](#_Toc72321411)

[Устанавливаем Mongoose 81](#_Toc72321412)

[Подключаемся к серверу mongo 81](#_Toc72321413)

[Запускаем сервер MongoDB 82](#_Toc72321414)

[Схемы и модели 82](#_Toc72321415)

[Создадим схему 82](#_Toc72321416)

[Свойства-объекты 84](#_Toc72321417)

[Свойства-массивы 84](#_Toc72321418)

[Создание модели на основе схемы 85](#_Toc72321419)

[Создание моделей: резюме 86](#_Toc72321420)

[Дополнительные ссылки 86](#_Toc72321421)

[Создание, чтение, обновление и удаление документов 86](#_Toc72321422)

[Создание документов — C 87](#_Toc72321423)

[Чтение документов — R 88](#_Toc72321424)

[Обновление документов — U 88](#_Toc72321425)

[Удаление документов — D 90](#_Toc72321426)

[Ссылки 91](#_Toc72321427)

[Как структурировать код. Контроллеры 91](#_Toc72321428)

[Настраиваем связи 92](#_Toc72321429)

[1. Настраиваем связь двух схем 92](#_Toc72321430)

[2. При создании документа записываем \_id 94](#_Toc72321431)

[3. Получаем всю информацию — метод populate 94](#_Toc72321432)

[Ссылки 95](#_Toc72321433)

**Интро**

За прошедшие полгода вы смогли разобраться во фронтенде и на практике доказали: этого времени и усердной работы достаточно, чтобы понять инженерную специальность. А теперь заглянем в бэкенд. Кто-то находит эту тему даже интереснее, чем фронт. В ближайшие пару месяцев вы определите это самостоятельно.

**Какие темы вы пройдёте:**

* Серверная разработка на Node.js.
* Введение в express.js.
* Работа с базами данных на примере Mongo DB.
* Подготовка к собеседованию. Анализ сложности алгоритмов.

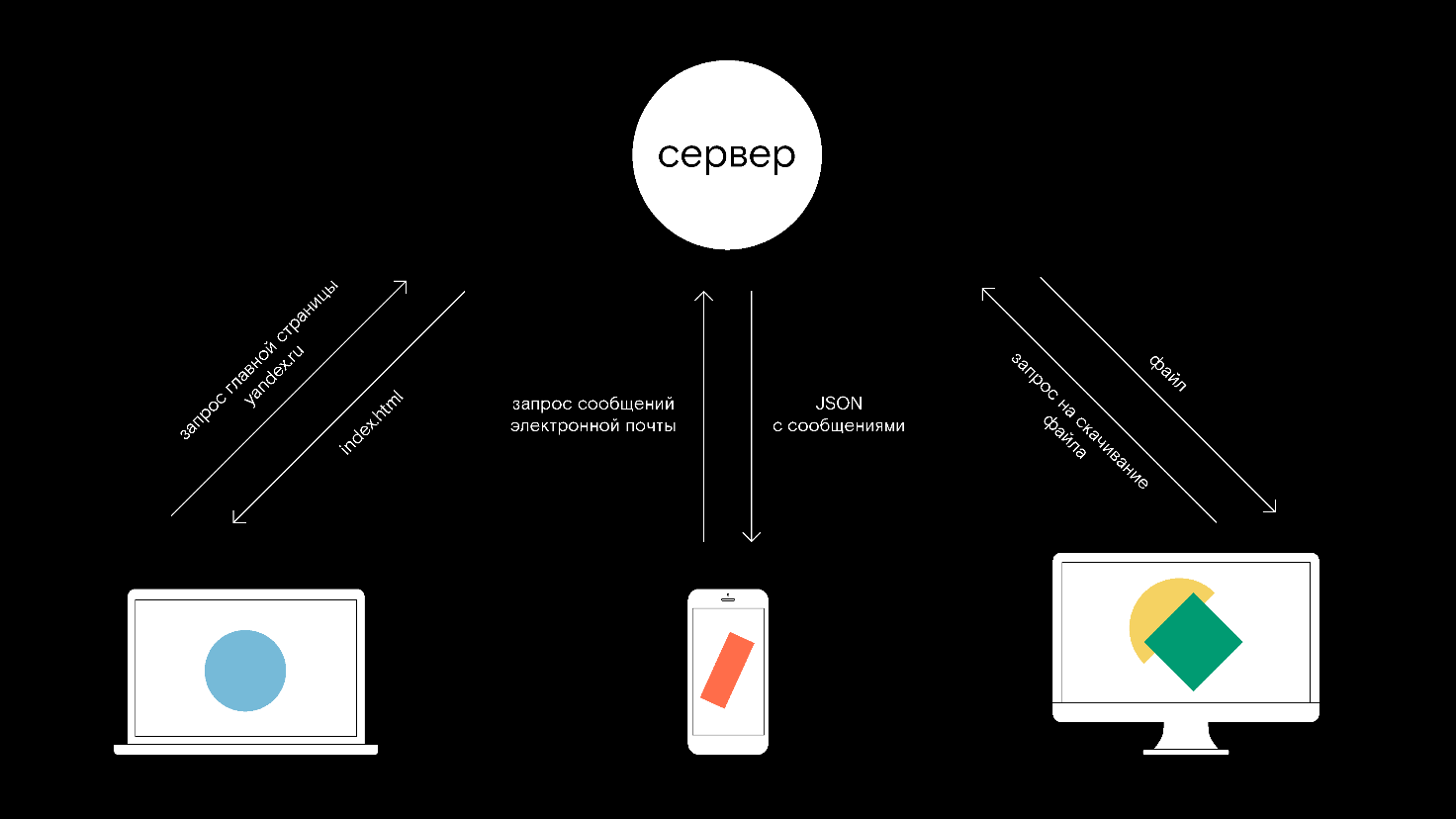
**Какие навыки получите:**

* В первом приближении разберётесь в устройстве Node.js: для чего он изобретён, из чего состоит и как работает.
* Научитесь поднимать простой сервер на «ноде» и быстро разворачивать бэкенд с помощью express.js.
* Разберетесь в API популярной MongoDB и научитесь работать с пользовательскими данными.
* В практической работе научите сервер раздавать статику, начнёте создавать свой API и подготовите структуру проекта.

Ну что, сервер сам себя не поднимет — поехали!

**Введение в серверную разработку**

Всё это время вы писали код для браузера. Но браузер — лишь видимая часть сайта. Теперь переходим к серверу — невидимой части.



**Что такое сервер?**

Сервер — это компьютер, который всегда включён. На нём работают специальные программы: они обрабатывают входящие запросы и дают ответы на них.

Вы уже создавали запросы к серверу в одной из практических работ по JavaScript:

fetch('https://[nomoreparties.co](http://nomoreparties.co)/cohort42/cards', {

headers: {

authorization: 'c56e30dc-2883-4270-a59e-b2f7bae969c6'

}

});

Когда сервер получал такой запрос, он выполнял три шага:

1. Проверял, что запрос корректный: токен и идентификатор группы правильные.
2. Получал карточки от базы данных.
3. Возвращал ответ.

Если на одном из этапов возникала ошибка, сервер сообщал о ней.

У сервера есть логика работы: она не появляется сама по себе — её нужно программировать.

**На каком языке пишут серверный код?**

Долгое время серверный код нельзя было писать на JavaScript. В основном для программирования сервера использовали языки C/C++, Java*,* Ruby*,* Python или PHP. Но в 2009 году американский программист Райан Даль презентовал платформу Node.js.

В следующих уроках мы подробнее разберёмся в основных предпосылках создания Node.js и в том, что включает в себя эта платформа.

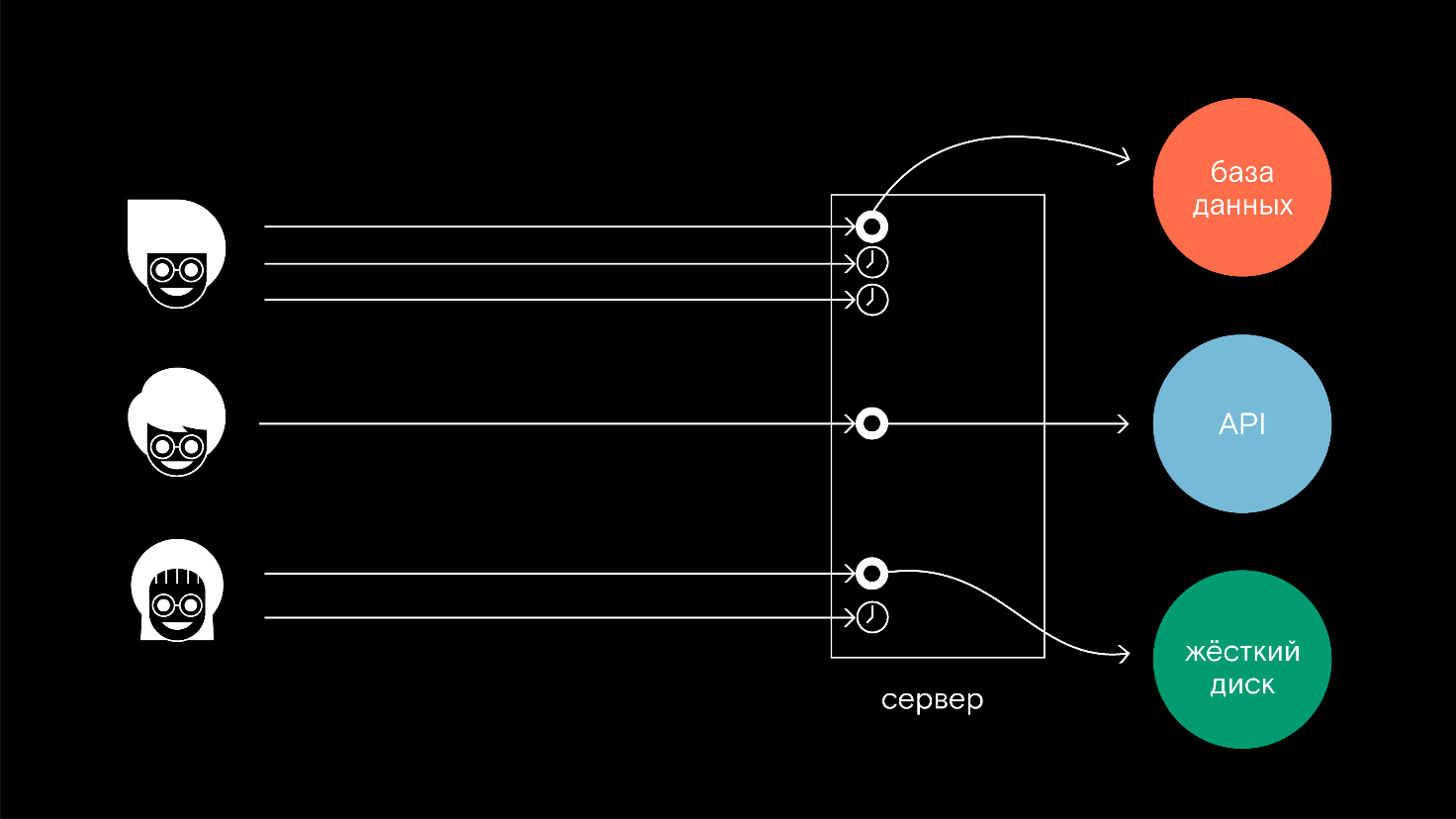
Приятный бонус: изучать новый язык не придётся, всё будет написано на уже знакомом JavaScript.

**Зачем изобрели Node.js?**

Здесь мы говорим о Node.js, как об инструменте для программирования сервера на JavaScript. Но платформа была создана с другой целью.

Раньше большинство веб-серверов работали так: каждый пользователь подключался к серверу и получал часть ресурсов — свой поток. Чем больше потоков, тем больше ресурсов необходимо серверу, чтобы обслужить всех пользователей.

Кроме того, при обработке запросов пользователя применялась блокирующая модель ввода-вывода. То есть все запросы к серверу выполнялись в той последовательности, в которой их отправили, что было неудобно. Один запрос, например, загрузка файла требует много времени на обработку, а другой — вернуть строку с данными — может быть лёгким. Но сервер не примется за вторую задачу, пока не обработает первую. А значит, ответа по простой задаче вы будете ждать очень долго.



Если на сервер с блокирующим вводом/выводом приходило миллион запросов, то создавалось миллион потоков, многие из которых просто ждали, пока задачи выполнялись по порядку и больше ничего не делали.

Райан Даль, создатель Node.js, полагал, что ввод и вывод должны быть неблокирующими. Реализовать это позволяет асинхронность: если от пользователя приходит запрос, который требует обращения в базу данных, можно зарегистрировать колбэк, поставить его в очередь и пойти выполнять другие запросы.

В JavaScript мы уже привыкли к асинхронности, нас не удивишь колбэками и промисами. Именно поэтому JavaScript и стал основным языком Node.js.

# За гранью браузера: что такое Node.js?

Языки программирования создают для решения определённых задач. JavaScript придумали, чтобы писать код для браузера, Objective-C и SWIFT — для приложений под macOS и iOS, PHP — для программирования серверов. Если язык используют только для работы в определённой среде, его называют **языком конкретной предметной области**.

Но языки программирования развиваются, обрастают новой функциональностью и инструментами — и иногда выходят за пределы изначальных целей. В результате язык становится универсальным: на нём можно писать код для совершенно разных платформ. Такие языки называют **языками общего назначения**. Например, на C++ можно запрограммировать и сервер, и холодильник.

Когда вышла Node.js, или на сленге — «нода», JavaScript эволюционировал из языка конкретной предметной области в язык общего назначения.

### Что такое Node.js?

Node.js. собран из двух основных составляющих:

* **движок JavaScript**. Он нужен для компиляции и исполнения кода — так же, как в браузере. В Node.js используется движок V8, который разработали Google для браузеров Chrome.
* **набор модулей** — утилиты, которые позволяют JavaScript взаимодействовать с файловой системой, сетью и другими системами компьютера. Именно модули высвобождают JS из браузера и позволяют ему работать в более широких рамках. С помощью утилит на JavaScript теперь можно писать логику работы сервера, программировать роботов и создавать десктопные приложения.

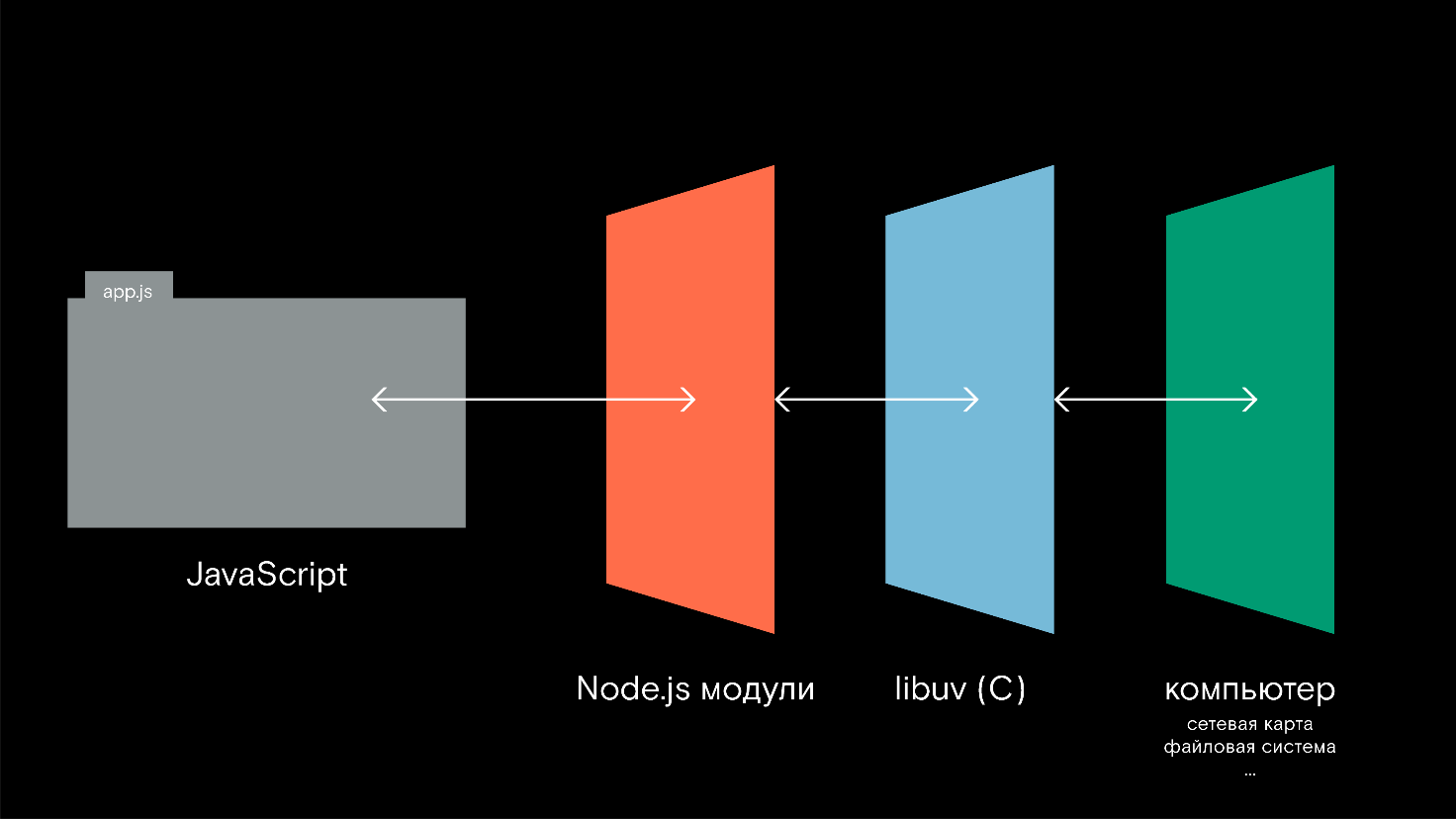
### Как работает Node.js?

Сервер — это удалённый компьютер, который умеет принимать запросы и возвращать ответы. Но как именно он это делает?

У сервера есть сетевая карта, которая может принимать запросы, поступающие из сети. Чтобы обрабатывать запросы, нужно получить доступ к сетевой карте — научиться получать и отправлять через неё данные.

В JavaScript нет функциональности для взаимодействия с «железом», поэтому он не может получить данные от сетевой карты. Чтобы решить эту проблему, пишут библиотеку команд на языке, который способен обмениваться данными с картой (например, на C).

Получается, каждый модуль — своего рода интерфейс для работы с C посредством JavaScript-команд. JavaScript-код взаимодействует с библиотекой libuv, написанной на C, а она обращается к файловой системе, сетевой карте и другим системам компьютера.



### Начинаем кодить

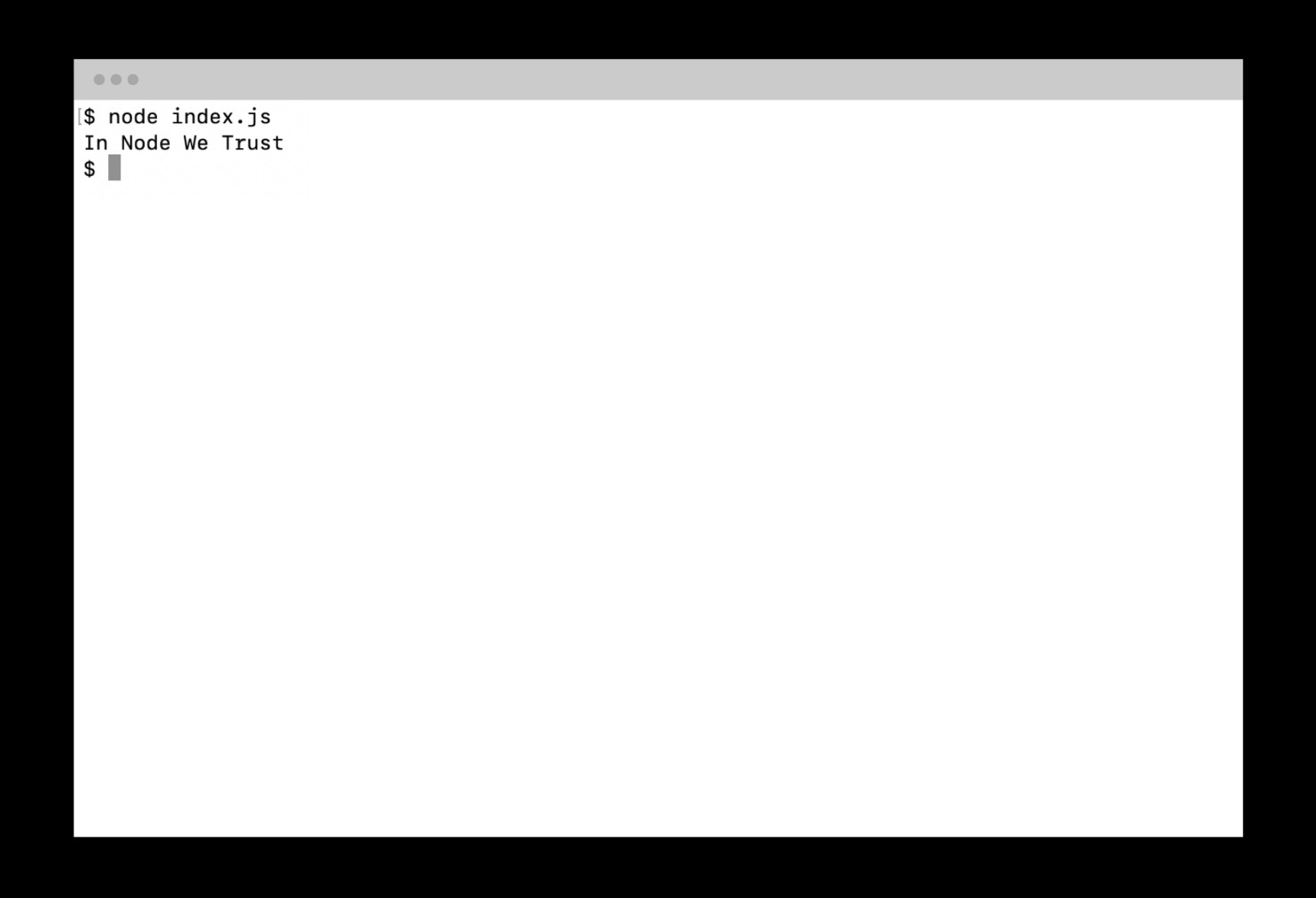
Вы уже установили ноду в предыдущих уроках курса. Но если начали работать на другом компьютере или ноду съела собака, можно снова скачать её тут:

[Библиотека пакетов NPM](https://praktikum.yandex.ru/learn/web/courses/370a2c73-45bf-439f-a747-ef4e3c0db48f/sprints/1703/topics/af1c30a0-05e5-46b5-8570-6742270bc6d8/lessons/548a7bd4-8c6e-4846-8bdb-7ffe699ab718/)

Пора начинать кодить. Локально создайте файл index.js с кодом:

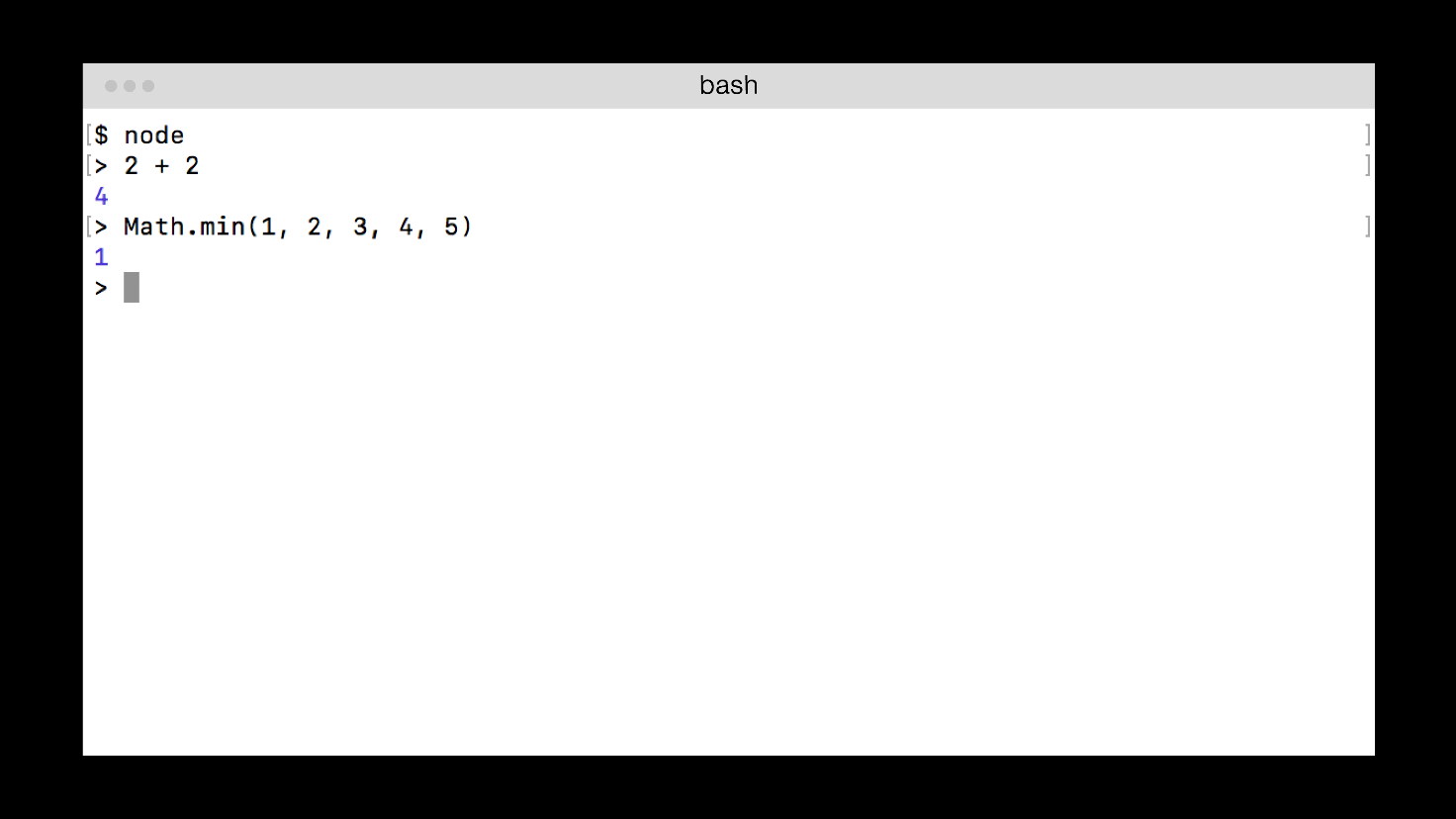
console.log('In Node We Trust');

Сохраните файл и запустите в терминале командой node index.js (для начала с помощью команды cdимя\_папки нужно попасть в папку с файлом):



Код, который написан на JavaScript, исполнился вне браузера. Node.js позволяет это делать.

Также можно запускать JavaScript-команды прямо в bash. Терминал будет работать примерно как браузерная консоль. Для этого запустите команду node, а дальше пишите JS-код:



Чтобы выйти из этого режима, нажмите Ctrl+D.

### Нода — не браузер

Знакомые вам API вроде DOM и fetch есть только в браузере. Из ноды к ним обратиться не выйдет. Проверьте это сами — отредактируйте файл index.js:

console.log(window);

Если запустите файл из терминала, увидите такую ошибку: ReferenceError: window is not defined.

В Node.js нет объекта window, потому что нет и браузерного окна. Вместо window в Node.js есть глобальный объект global: его свойства и методы доступны глобально, но на этом их сходство с window заканчиваются. Global содержит собственные методы и свойства, их мы будем изучать по ходу курса.

Ну, хватит теории — пора переходить к практике. В следующем уроке создадим сервер на Node.js.

**Сервер на Node.js**

Пришло время создать простейший сервер. Для этого придётся сделать несколько шагов (довольно много, на самом деле). Так что не будем долго тянуть и приступим.

**Подключите API**

Серверную программу нужно научить принимать сообщения, которые поступают на сетевую карту. В JavaScript нет такой встроенной возможности, зато она есть в специальном API из Node.js.

Имя этого API — http. Импортируйте его в код файла index.js командой require:

const http = require('http');

**Создайте сервер**

Для этого достаточно строчки кода:

const http = require('http');

const server = http.createServer(); *// создаём сервер*

Так мы даём библиотеке на C команду подключиться к сетевой карте и принимать сообщения. Теперь мы получаем к ним доступ из JavaScript-кода.

**Настройте порт**

На компьютере может работать несколько программ, которые принимают сообщения, приходящие на сетевую карту. Чтобы это было возможно, у сетевой карты есть точки входа — порты. Всего их 65536.

Node.js надо знать, с какого порта принимать сообщения: для этого используют метод listen.

Укажите ноде принимать сообщения с 3000 порта:

const http = require('http');

const server = http.createServer(); *// создаём сервер*

server.listen(3000); *// будем принимать сообщения с 3000 порта*

**Установите ответ**

Теперь мы можем принимать входящие сообщения. Но на них ещё нужно реагировать. Для этого функция createServer принимает колбэк — в нём и описывают код, который нужно запустить при получении запроса:

const http = require('http');

*// передадим обработчик*

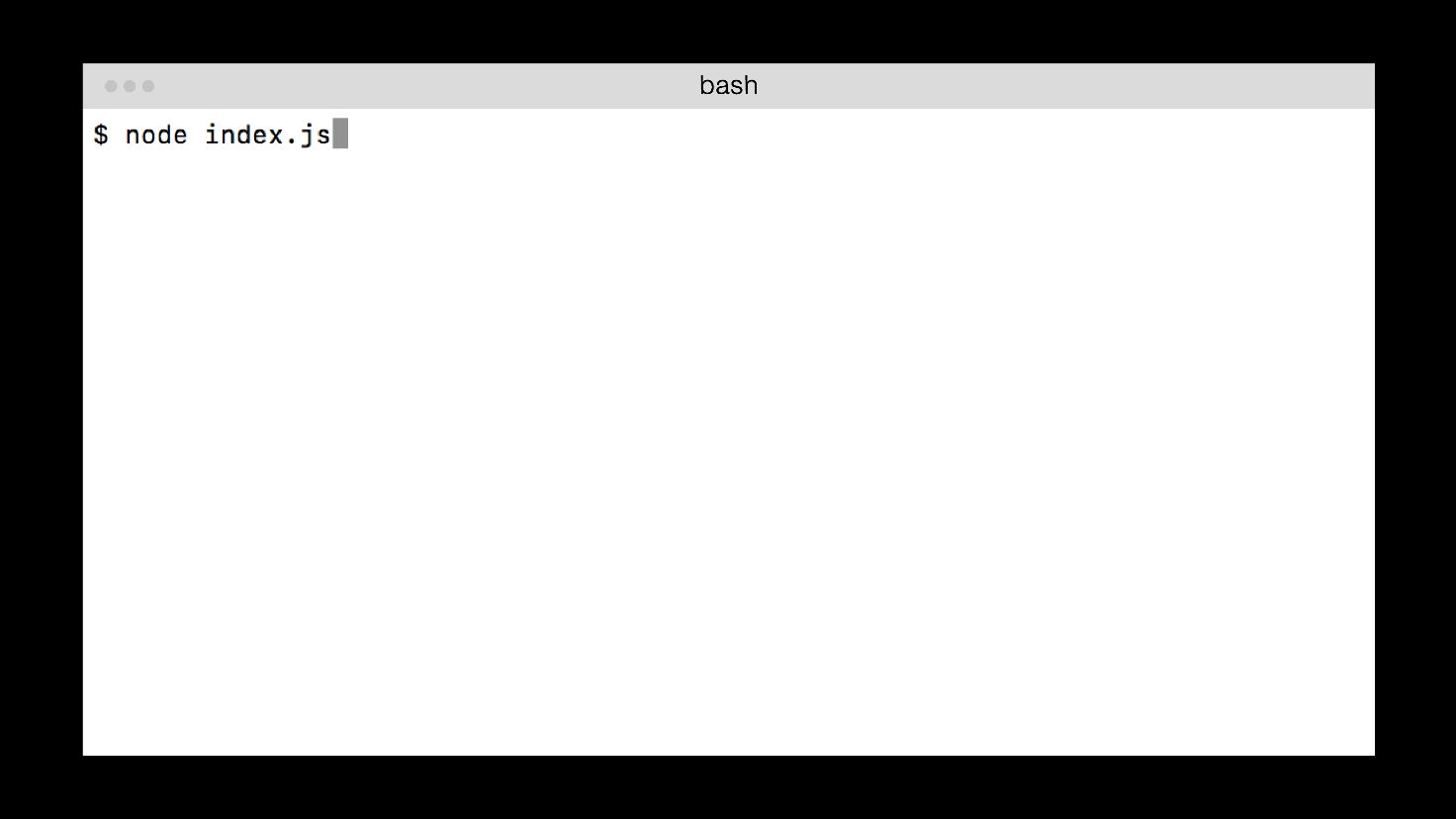
const server = http.createServer(() => {

console.log('Пришёл запрос!');

});

server.listen(3000);

Функция-обработчик, которую передал createServer, будет вызываться при каждом входящем запросе, пришедшем на 3000 порт. Сохраните этот код в файле index.js и запустите его в терминале:



**Настройте запрос и ответ**

Откройте в браузере адрес http://localhost:3000. В терминале появится сообщение — значит, наш сервер принимает запросы и реагирует на них. Но пока его реакция — всего лишь сообщение в консоли. Нам же нужно отправить пользователю ответ. Чтобы это сделать, сначала разберёмся с параметрами колбэка. Первый параметр — объект запроса, второй — объект ответа:

const http = require('http');

const server = http.createServer((request, response) => {

console.log('Пришёл запрос!');

console.log(request);

console.log(response);

});

server.listen(3000);

**Объект запроса**

Обычно его называют request, или сокращённо req. Вся информация о запросе содержится в свойствах объекта:

*// запустите этот файл и перейдите*

*// в браузере по адресу: http://localhost:3000/hello*

const http = require('http');

const server = http.createServer((req, res) => {

console.log(req.url); *// /hello*

console.log(req.method); *// GET*

console.log(req.headers); *// здесь будут заголовки запроса*

console.log(req.body); *// а здесь тело запроса, но у GET запроса его нет*

});

server.listen(3000);

**Объект ответа**

Второй параметр обработчика называют response, или res. Он содержит свойства и методы для работы с ответом:

*// запустите этот файл и перейдите*

*// в браузере по адресу: http://localhost:3000*

const http = require('http');

const server = http.createServer((req, res) => {

res.statusCode = 200; *// статус ответа*

res.statusMessage = 'OK'; *// сообщение ответа*

res.setHeader('Content-Type', 'text/plain'); *// добавить ответу заголовок*

res.write('Hello, '); *// отправить часть ответа — строку "Hello, "*

res.write('world!'); *// отправить часть ответа — строку "world!"*

res.end(); *// закончить отправку ответа*

});

server.listen(3000);

Ответ от сервера получен! Мы отправляем его частями методом res.write.

Когда все данные получены, вызываем метод res.end. Так мы обозначаем, что ответ пришёл полностью. Обратите внимание: отправку всех частей ответа нужно прописать до вызова res.end.

А вот такой код приведёт к ошибке:

const http = require('http');

const server = http.createServer((req, res) => {

res.write('Hello, ');

res.end();

res.write('world!'); *// вызовет ошибку*

});

server.listen(3000);

Код статуса ответа и заголовки можно передать одним методом — writeHead:

const http = require('http');

const server = http.createServer((req, res) => {

res.writeHead(200, {

'Content-Type': 'text/html; charset=utf8'

});

*// в методе end тоже можно передать данные*

res.end('<h1>Привет, мир!</h1>', 'utf8');

});

server.listen(3000);

Не всегда кодировка по умолчанию — UTF-8, поэтому, чтобы не было проблем с отображением, нужно дописывать её вручную.

Вторым аргументом end принимает кодировку отправляемых данных. Каждый запрос должен заканчиваться вызовом этого метода.

**Пользуйтесь переменными окружения**

Сервер готов, но у него есть одна проблема. Сейчас мы захардкодили (то есть явно указали в коде) входящий порт. Это не лучшая практика: входящие данные лучше передавать параметрами.

В Node.js это позволяют сделать переменные окружения. Они доступны из любой части программы, поэтому мы можем передать их при запуске сервера из терминала. Для этого перед командой запуска прописывают имена переменных и их значения:

NODE\_ENV=production node index.js

*# NODE\_ENV - имя переменной окружения, а production — её значение*

Внутри скриптов переменные окружения хранятся в объекте process.env:

if (process.env.NODE\_ENV !== 'production') {

console.log('Код запущен в режиме разработки');

}

Все переменные окружения принято называть заглавными буквами в snake case (слова разделены нижним подчёркиванием). В переменных окружения обычно передают конфигурационную информацию, необходимую приложению. Именно так мы укажем порт — при запуске сервера:

PORT=3000 node app.js

# Тело запроса: потоки

В теме [«Работа с API»](https://praktikum.yandex.ru/trainer/web/lesson/e591cdbc-f411-4447-85a7-00d745529f2b) мы говорили, что получить ответ от сервера — не значит получить данные. Так работает протокол передачи данных: информация бьётся на небольшие пакеты, которые собираются воедино на стороне клиента.

Мы вызывали метод json асинхронно: нужно было дождаться события загрузки данных, а затем превратить ответ в объект JSON:

fetch('https://api.kanye.rest')

.then(res => res.json())

.then((result) => {

console.log(result);

});

Сначала обрабатываем приход ответа, а потом — данных. Отсюда и асинхронный код.

С запросом ситуация такая же: если он дошёл до сервера, это ещё не значит, что пользовательские данные пришли полностью.

### Как обрабатываются данные запроса?

Нам неизвестно заранее, сколько данных отправит пользователь. При этом информация не приходит на сервер просто так — с ней нужно что-то сделать, например, записать на диск.

Если бы мы ждали, пока вся информация дойдёт до сервера, и после этого начинали её обрабатывать, то потеряли бы уйму времени. А если пользователей тысячи, сайт бы безнадёжно тормозил. Чтобы этого избежать, обработку запроса реализовали так: поступление каждого пакета данных — отдельное событие, которое нужно обработать асинхронно. Получается, мы устанавливаем слушатель, который отслеживает пакеты, и добавляем очередной пакет ко всем остальным. Так мы собираем запрос целиком «попакетно».

Обратите внимание: пока мы ждём очередной пакет, можно работать с другими, чтобы ускорить процесс и обойтись без зависаний.

### Как собрать данные запроса?

Для работы с телом запроса есть специальная структура — поток (англ. stream). Чтобы полностью получить тело запроса, нужно принять все блоки и склеить их.

### Что обрабатывать, когда приходит блок? Событийная модель

Приход каждого блока — отдельное событие. Чтобы его обработать, пользуйтесь методом on объекта запроса:

const http = require('http');

const server = http.createServer((req, res) => {

req.on('data', (chunk) => {

console.log(chunk); *// <Buffer 66 69 65 6c 64 3d 76 61 6c 75 65>*

});

});

server.listen(3000);

Событие data, которое мы обрабатываем, происходит при получении очередного пакета данных. Когда на сервер придёт запрос, у которого есть тело, в консоли окажутся вот такие сообщения: <Buffer 66 69 65 6c 64 3d 76 61 6c 75 65>. Количество сообщений зависит от размера запроса: больше данных — больше сообщений.

Данные из сети приходят в бинарном виде. Получить их в JavaScript мы можем благодаря специальному объекту Buffer.

Каждый поступивший блок нужно добавлять в одну переменную. После того как придёт последний пакет, мы целиком получим тело ответа:

const http = require('http');

const server = http.createServer((req, res) => {

let data = '';

req.on('data', (chunk) => {

data += chunk.toString();

});

});

server.listen(3000);

Когда приходит последний пакет, происходит событие end. Его и нужно обработать, чтобы превратить запрос в JSON-объект.

const http = require('http');

const server = http.createServer((req, res) => {

let data = '';

req.on('data', (chunk) => {

data += chunk.toString();

});

req.on('end', () => {

console.log(JSON.parse(data));

});

});

server.listen(3000);

Ответ от сервера обрабатывается по той же самой схеме. Вы не писали код для склеивания блоков в единый ответ только потому, что эта логика скрыта под капотом браузера. Так обрабатывать входные данные значительно проще.

Такой же способ есть и для обработки запроса. В следующим уроке о нём и поговорим.

**Система модулей Node.js**

Модули позволяют работать с системами компьютера из JavaScript. В предыдущих уроках вы создали сервер, импортировав модуль http:

const http = require('http');

Http — встроенный модуль Node.js. То, как именно импортировать модуль, прописывают в документации, но обычно достаточно передать имя модуля функции require:

const md5 = require('md5');

Так устанавливаются пакеты из npm. Но вы можете создавать собственные модули и импортировать в проект. Любой файл c JS-кодом может стать модулем — нужно лишь оформить его по определённым правилам. О них расскажем в этом уроке.

**Как импортировать модули?**

Код модуля описывают в отдельном файле, а затем импортируют в точку входа — файл index.js. Создадим файл utils.js в папке проекта и импортируем его в index.js:

*// index.js*

const utils = require('./utils'); *// Поскольку все модули имеют расширение .js, при импорте его можно опустить и ограничиться названием модуля — utils.*

Точка в начале названия модуля означает, что файл нужно искать в той же директории, в которой лежит index.js. Если в папке с проектом нет файла utils.js, движок будет искать папку с именем utils, а в ней — файл index.js.

Также мы можем импортировать модуль из других папок, задав относительный путь:

const utils = require('../utils'); *// две точки означают переход в директорию на уровень выше*

const helpers = require('../../helpers');

**Как экспортировать модули?**

Чтобы модули работали, их нужно правильно оформить: показать, какие переменные и функции мы хотим получить снаружи. Для этого важно явно обозначить, что именно экспортируется из модуля.

Это можно сделать с помощью объекта module.exports, доступного внутри файла. Всё, что нужно экспортировать, записывают в свойства этого модуля:

*// utils.js*

module.exports.someFunction = () => {

console.log('Меня экспортировали');

};

module.exports.someValue = 42;

Не важно, как именно вы записываете значения в свойства module.exports. Такой код тоже сработает:

*// utils.js*

const someFunction = () => {

console.log('Меня экспортировали');

};

const someValue = 42;

module.exports = {

someFunction,

someValue

};

При импорте мы получим объект module.exports. Останется только достать из него свойства:

*// index.js*

const utils = require('./utils');

const someFunction = utils.someFunction;

const someValue = utils.someValue;

Проще сделать это деструктуризацией:

*// index.js*

const { someFunction, someValue } = require('./utils');

**Как называть модули?**

Только строчными буквами. У Node.js есть свой алгоритм кеширования файлов. При этом заглавные и строчные буквы — разные символы для ноды: прописные буквы часто ломают кеширование. Для именования модулей пользуйтесь маленькими буквами.

**ES6 модули работают в ноде?**

Да, но только начиная с 8 версии Node.js. Кроме того, чтобы ES6 модули заработали:

* ноду нужно запустить со специальным флагом -experimental-modules;
* все файлы должны иметь расширение .mjs.

*# вот так ES6 модули работают*

node -experimental-modules index.mjs

Экспорт ES6 модулей выполняется иначе. Необходимо экспортировать каждое нужное значение директивой export:

*// test.mjs*

export const theme = 'ES6-module';

export const method = (name) => { return `Здравствуйте, ${name}!` };

При импорте такого модуля мы получим объект с ключами, как у экспортированных переменных.

*// index.js*

import { theme, method } from './test'; *// из test вернулся объект*

console.log(method('Стас Басов')); *// "Здравствуйте, Стас Басов!"*

ES6-модули по умолчанию работают в строгом режиме, поэтому вам не нужно прописывать 'use strict'.

Но это не касается node-модулей в целом — они работают в обычном режиме. Так что, если хотите писать код в строгом режиме, не забывайте явно прописывать это директивой 'use strict' внутри каждого модуля.

С настройкой мы закончили. В следующем уроке опустимся на уровень глубже и разберёмся в устройстве модулей.

# Работа с файловой системой

Node.js позволяет взаимодействовать с системами компьютера. И одна из них — файловая. Серверу приходится постоянно работать с файловой системой: когда мы отправляем пост в Instagram, сервер должен получить фотографию и записать на диск; когда просматриваем ленту, запрашиваем фото с сервера, он находит нужный файл и отправляет нам данные.

В этом уроке расскажем, как работать с файлами на сервере: научим читать данные из файлов и папок, записывать информацию в файл, создавать директории и удалять файлы. Поехали.

### Какой модуль нужен

Для работы с файлами в ноде используют модуль fs. У него есть по методу для каждой операции.

Начнём с чтения данных из файла. Это делают функцией readFile (англ. «прочитать файл»). Она работает асинхронно и принимает два аргумента: имя файла, из которого нужно прочитать данные, и колбэк. В колбэке описывают, что нужно сделать с прочитанными данными.

На вход колбэк принимает два аргумента: ошибку и данные файла.

const fs = require('fs');

fs.readFile('data.json', (err, data) => {

if (err) {

console.log(err);

return;

}

console.log('data: ', data.toString('utf8'));

});

В первый параметр колбэка может попасть одно из двух значений:

* объект ошибки — в случае ошибки при чтении файла;
* null, если ответ пришёл и всё хорошо.

Второй параметр — данные файла — приходят в виде буффера (двоичных данных). Прежде чем работать с данными, их нужно преобразовать в строку одним из двух способов:

* методом toString, как в примере: data.toString('utf8'). Он принимает на вход строку — кодировку читаемого файла.
* передать кодировку внутри свойства encoding объекта опций. В таком случае объект опций становится вторым параметром метода readFile. Данные при этом приходят сразу в виде строки, так что преобразовывать их не нужно:

const fs = require('fs');

fs.readFile('data.json', { encoding: 'utf8' }, (err, data) => { *// передали вторым аргументом объект опций. Он содержит свойство encoding, куда записывают кодировку данных*

if (err) {

console.log(err);

return;

}

console.log('data: ', data); *// здесь мы не вызываем метод toString, поскольку данные уже пришли в виде строки*

});

### Промисы при работе с файлами

В 10 версии Node.js в модуле fs появилась поддержка промисов. Если вы пользуетесь промисами, колбэки передавать не нужно. Промис разрешается с прочитанными данными или отклоняется в случае ошибки, поэтому просто добавляйте асинхронные обработчики then:

const fsPromises = require('fs').promises;

fsPromises.readFile('data.json', { encoding: 'utf8' })

.then((data) => {

console.log(data);

})

.catch(err => {

console.log(err);

});

[Ссылка на документацию к API для поддержки промисов](https://nodejs.org/api/fs.html#fs_fs_promises_api)

## Проблемы с путями

При работе с файловой системой мы встречаемся с проблемой: как задавать пути к файлам? Относительно входного модуля или того, в котором мы пишем код? Разберёмся на примере.

Есть точка входа app.js с таким кодом:

*// app.js*

const fs = require('fs');

const readFile = () => {

const file = fs.readFile('file.txt', { encoding: 'utf8' }, (err, data) => {

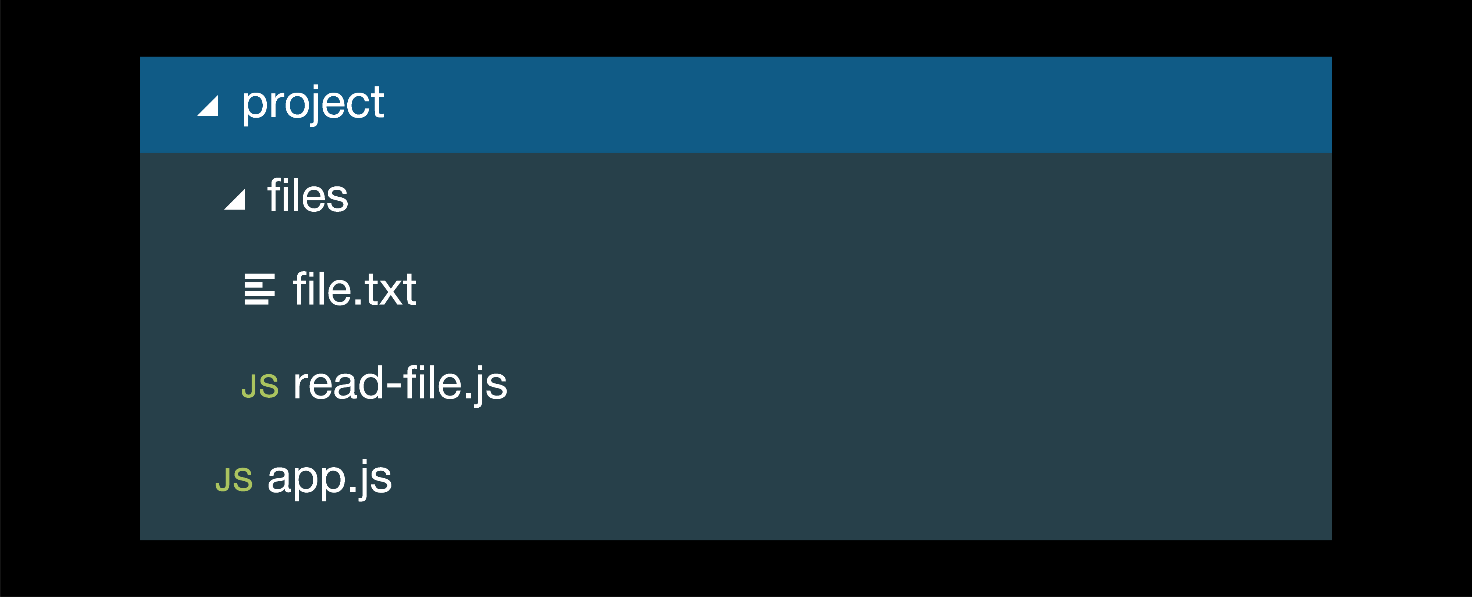
console.log(data); *// Выводим его содержимое в консоль*

}); *// Здесь читаем файл file.txt по относительному пути*

};

readFile();

Позже мы решили убрать файл и всю логику работы с ним в отдельную папку — и получили такую файловую структуру:



Поскольку логика работы с файлом теперь находится в отдельном модуле, его нужно подключить к точке входа — файлу app.js. Импортируем функцию readFile:

*// app.js*

const fs = require('fs');

const { readFile } = require('./files/read-file');

readFile();

И экспортируем эту же функцию из read-file.js:

*// read-file.js*

const fs = require('fs');

module.exports.readFile = () => {

const file = fs.readFile('file.txt', { encoding: 'utf8' }, (err, data) => {

console.log(data);

});

};

Такой код приведёт к ошибке: файл file.txt не найден. Проблема в относительном пути: он отсчитывается от файла, где исполняется код, а не где он написан. Мы могли бы изменить путь к файлу в коде: вместо "file.txt" написать "/files/file.txt", но такой подход не очень удобен. Если в проекте много файлов, менять файловую структуру становится очень сложно: тяжело отследить, где в коде и к каким файлам мы обращаемся.

Выход есть: сделать пути динамическими. То есть не прописывать путь явно, а получать его из информации о самом модуле. Для этого нужно:

* знать, где находится модуль, из которого мы обращаемся к файлу;
* создать дополнительный модуль «собирания» пути к файлу.

Обо всём по порядку.

### Что хранится в модуле

Любой модуль Node.js содержит информацию о самом себе и окружении. Мы всегда можем посмотреть, где лежит модуль, и входной ли он для приложения.

### Где находится модуль

Внутри любого модуля есть переменные \_\_filename и \_\_dirname. Они хранят имя файла модуля и путь к папке, где лежит модуль, соответственно.

*// app.js*

console.log(\_\_filename); *// /usr/local/project/app.js*

console.log(\_\_dirname); *// /usr/local/project*

Мы могли бы сделать путь динамическим шаблонной строкой или конкатенацией:

const file = fs.readFile(`${\_\_dirname}/file.txt`, { encoding: 'utf8' }, (err, data) => {});

Но так поступать не стоит: на разных операционных системах бывают различные разделители. На macOS — слеш, а на MS Windows — обратный слеш. Чтобы избежать проблем с разделителями, преобразовывать путь лучше специальным модулем path.

### Как преобразовать путь

Модуль path хранит множество полезных методов для работы с путём к файлу. Задачу по преобразованию выполняет метод join. Он учитывает контекст, а потому не возникнет проблемы с разделителями, о которой мы говорили:

*// read-file.js*

const fs = require('fs');

const path = require('path');

module.exports.readFile = () => {

const filepath = path.join(\_\_dirname, 'file.txt'); *// собрали абсолютный путь к файлу*

const file = fs.readFile('file.txt', { encoding: 'utf8' }, (err, data) => {

console.log(data);

});

};

Вот ещё несколько полезных методов модуля path:

const fs = require('fs');

const path = require('path');

*// path.normalize убирает из пути лишние*

*// разделители и разрешает символы .. и .*

path.normalize('/foo/bar//baz/asdf/quux/..'); *// /foo/bar/baz/asdf*

*// path.dirname извлекает путь к папке из переданного пути*

path.dirname(process.mainModule.filename); *// /usr/local/my-project*

*// path.extname извлекает расширение файла*

path.extname('app.js'); *// .js*

[О других методах можете почитать в документации Node.js](https://nodejs.org/api/path.html)

Резюмируем: для работы с файловой системой нужны два модуля: fs и path. Первый содержит методы для работы с файлами, второй — для преобразования путей. Эти модули позволяют выстроить работу с файловой системой и не навредить при этом гибкости проекта. Мы можем менять структуру файлов и не затрагивать функциональность.

# Что ещё может модуль fs?

### Читать все файлы директории

Для этого есть метод fs.readdir. Он читает все файлы внутри директории. Первый аргумент метода — путь к директории. Второй — колбэк, в котором описано, что делать с полученными данными.

У колбэка также два параметра. Первый — ошибка, второй — массив имён найденных файлов:

const fs = require('fs');

fs.readdir('.', (err, files) => {

if (err) {

console.log(err);

return;

}

console.log('data: ', files);

});

### Создавать папки

С этим поможет метод fs.mkdir. Он принимает два параметра: имя новой папки и колбэк с единственным аргументом — ошибкой. Первым параметром можно передать имя папки вместе с путём, где нужно создать папку.

const fs = require('fs');

fs.mkdir('incomingData/data', (err) => {

if (err) console.log(err);

});

### Записывать данные в файл

Для этого есть метод fs.writeFile. Принимает три параметра:

* файл, куда нужно записать данные;
* сами данные, записанные строкой;
* колбэк для обработки ошибки.

const fs = require('fs');

fs.writeFile('data.json', JSON.stringify([1, 2, 3]), (err) => {

if (err) console.log(err);

});

### Удалять файлы

Это делает метод fs.unlink. У него два параметра: имя файла и колбэк для обработки ошибок.

const fs = require('fs');

fs.unlink('data.json', (err) => {

if (err) {

console.log(err);

return;

}

console.log('Файл удалён!');

});

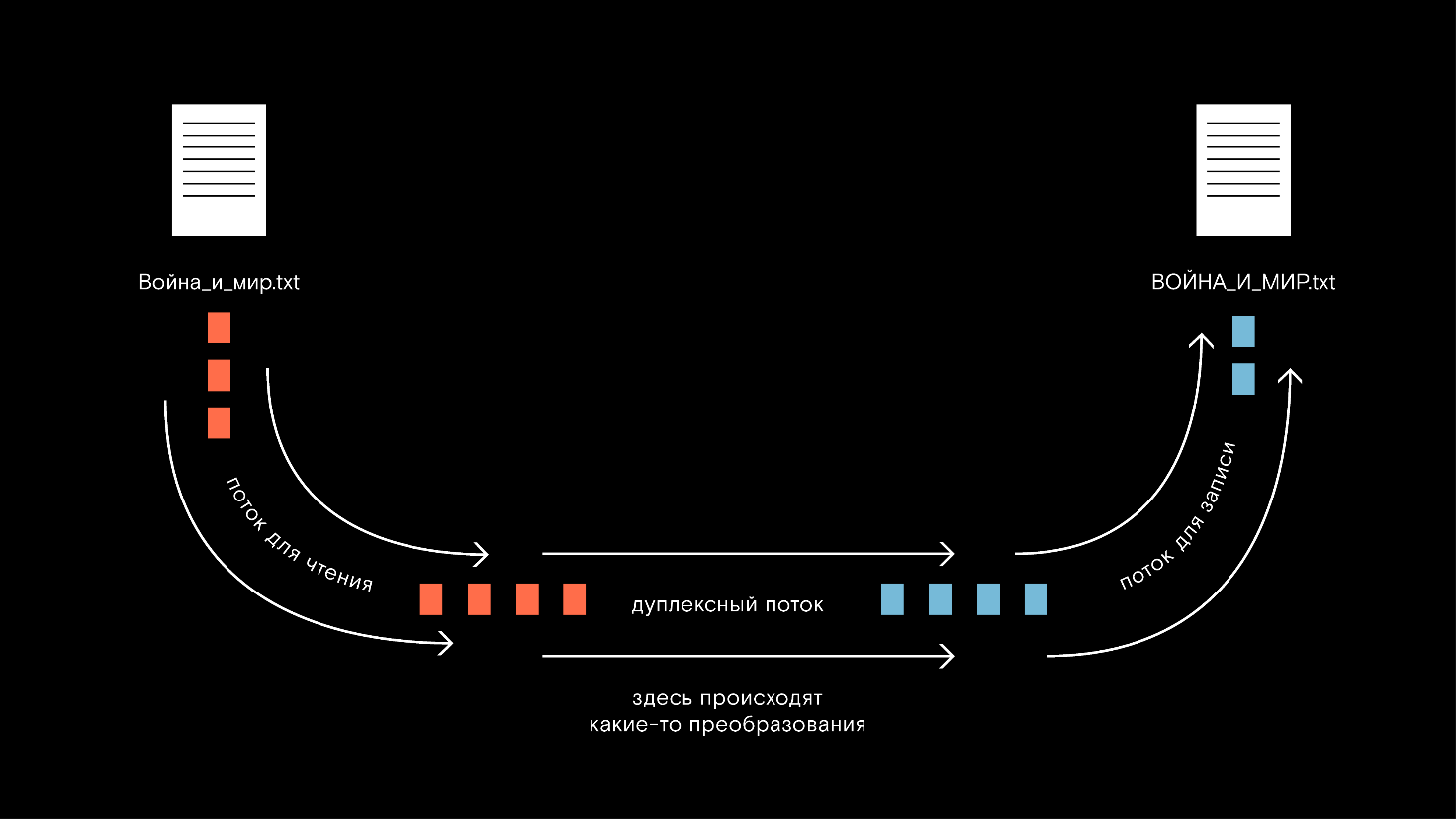
### Много чего ещё

Остальные методы модуля fs по большей части работают аналогично. Так что, если захотите сделать с файлом что-то, чего мы не объясняли — почитайте документацию: вы без труда разберётесь.

[Ссылка на документацию модуля fs.](https://nodejs.org/api/fs.html)

**Потоки для чтения и записи файлов**

Для чтения запроса можно использовать поток (англ. stream) — вы уже делали это в четвёртом уроке темы. Но это не единственное назначение потоков: они используются в Node.js, в том числе для работы с файлами. Поток — последовательность данных, которые поступают от какого-то источника. Файл вполне может быть таким источником.



**Какие бывают потоки и как ими пользоваться?**

Потоки бывают трёх видов:

* для чтения данных из тела запроса или из файла;
* для записи данных в файл;
* дуплексные потоки — одновременно для чтения и записи.

Дуплексные потоки часто используют для трансформации данных, которые в них поступают. Их можно комбинировать с потоками чтения и записи.

К примеру, на сайте есть видео с разрешением 1920 на 1080 точек. Пользователь подключается к общественному вайфаю в аэропорту и пытается его посмотреть. Скорость оставляет желать лучшего, и видео тормозит. Тут на помощь вам, вайфаю и пользователю приходят потоки:

* сначала открывается поток чтения из видеофайла;
* данные попадают в дуплексный поток: он одновременно считывает данные и преобразует их — сжимает;
* затем данные попадают в поток для записи — на устройство пользователя.

В результате видео хоть и потеряло в качестве, зато не тормозит.

**Как работать с потоками?**

В модуле fs есть методы createReadStream и createWriteStream. Они создают потоки для чтения и записи соответственно. Первым аргументом методы принимают путь к файлу, а вторым — объект опций, где можно передать, например, кодировку:

const fs = require('fs');

*// создаём поток для чтения из файла in.txt*

const reader = fs.createReadStream('./in.txt', { encoding: 'utf8' });

*// создаём поток для записи в файл out.txt*

const writer = fs.createWriteStream('./out.txt', { encoding: 'utf8' });

Чтобы скомбинировать потоки чтения и записи, то есть превратить их в дуплексный поток, нужно при каждом событии data потока чтения вносить порцию данных в поток записи:

const fs = require('fs');

const reader = fs.createReadStream('./in.txt', { encoding: 'utf8' });

const writer = fs.createWriteStream('./out.txt', { encoding: 'utf8' });

reader.on('data', (data) => { *// отслеживаем событие data потока чтения*

writer.write(data); *// записываем порцию данных в поток записи*

});

Когда данные полностью прочитаны, поток для чтения сгенерирует событие end. Его нужно обработать и сообщить потоку записи, что все данные пришли и ничего больше записывать не нужно. Для этого у потока записи есть метод end:

const fs = require('fs');

const reader = fs.createReadStream('./in.txt', { encoding: 'utf8' });

const writer = fs.createWriteStream('./out.txt', { encoding: 'utf8' });

reader.on('data', (data) => {

writer.write(data);

});

*// когда все данные прочитаны, даём Node сигнал,*

*// что поток для записи больше нам не нужен*

reader.on('end', () => {

writer.end();

});

Последнее, что нужно сделать — обработать ошибку. В случае ошибки поток для чтения сгенерирует событие error. Его нужно обработать:

const fs = require('fs');

const reader = fs.createReadStream('./in.txt', { encoding: 'utf8' });

const writer = fs.createWriteStream('./out.txt', { encoding: 'utf8' });

reader.on('data', (data) => {

writer.write(data);

});

reader.on('end', (data) => {

writer.end();

});

*// повесим обработчик события error*

reader.on('error', (err) => {

console.log(err);

});

**Почему не воспользоваться методами чтения и записи файлов?**

Конечно, эту логику можно реализовать методами readFile и writeFile. Но при таком подходе программа работала бы медленнее, потому что серверу пришлось бы сначала полностью прочитать файл, записать его в оперативную память и только потом — на диск. Получается, для обработки каждого файла нужно заметно больше памяти. Причём больше на размер файла. Если вы загружаете видео, программа в целом будет работать ещё медленнее.

Потоки позволяют избежать таких проблем: данные читаются небольшими порциями и тут же записываются на диск. В оперативке при этом сохраняется только очередная порция данных, и алгоритм менее «прожорливо» потребляет память.

**Метод pipe для работы с потоками**

Чтобы организовать простейшую работу с двумя потоками, мы добавили три обработчика событий. Но код можно сделать заметно короче, если воспользоваться специальным методом для комбинирования потоков — pipe. Вам даже не нужно закрывать потоки и обрабатывать ошибки — вся эта логика уже описана внутри метода pipe:

const fs = require('fs');

const reader = fs.createReadStream('./in.txt', { encoding: 'utf8' });

const writer = fs.createWriteStream('./out.txt', { encoding: 'utf8' });

reader.pipe(writer);

Метод pipe есть только у потоков для чтения — поток для записи передают как аргумент.

Напоследок разберём простой пример. Обращение пользователя к серверу — тоже поток для чтения, так что мы можем обработать этот запрос методом pipe:

const http = require('http');

const fs = require('fs');

const server = http.createServer(function (req, res) {

req.pipe(fs.createWriteStream(`./out-${Math.random()}.txt`));

});

server.listen(3000);

Данные, которые пользователь отправил в запросе, мы записываем этим кодом в текстовый файл со случайным именем.

Потоки — большая и сложная тема. Мы рассказали лишь об основах, которые необходимы на практике. Тем не менее, если вы хотите освоить «серьёзный» бэкенд, в потоках нужно основательно разобраться.

# Отладка Node.js приложения

Код без ошибок пишет только тот, кто не пишет код. Вы наверняка убедились в этом с первого же месяца обучения.

В курсах по фронтенду вы делали отладку кода в браузере: для этого там есть инструменты разработчика. Нода существует вне браузера, но это не мешает использовать браузерные инструменты для отладки. Просто нужен предварительный танец с бубном. В этом уроке расскажем, как его танцевать.

### Отладка в браузере

Чтобы отлаживать код Node.js в браузере, выполните два шага:

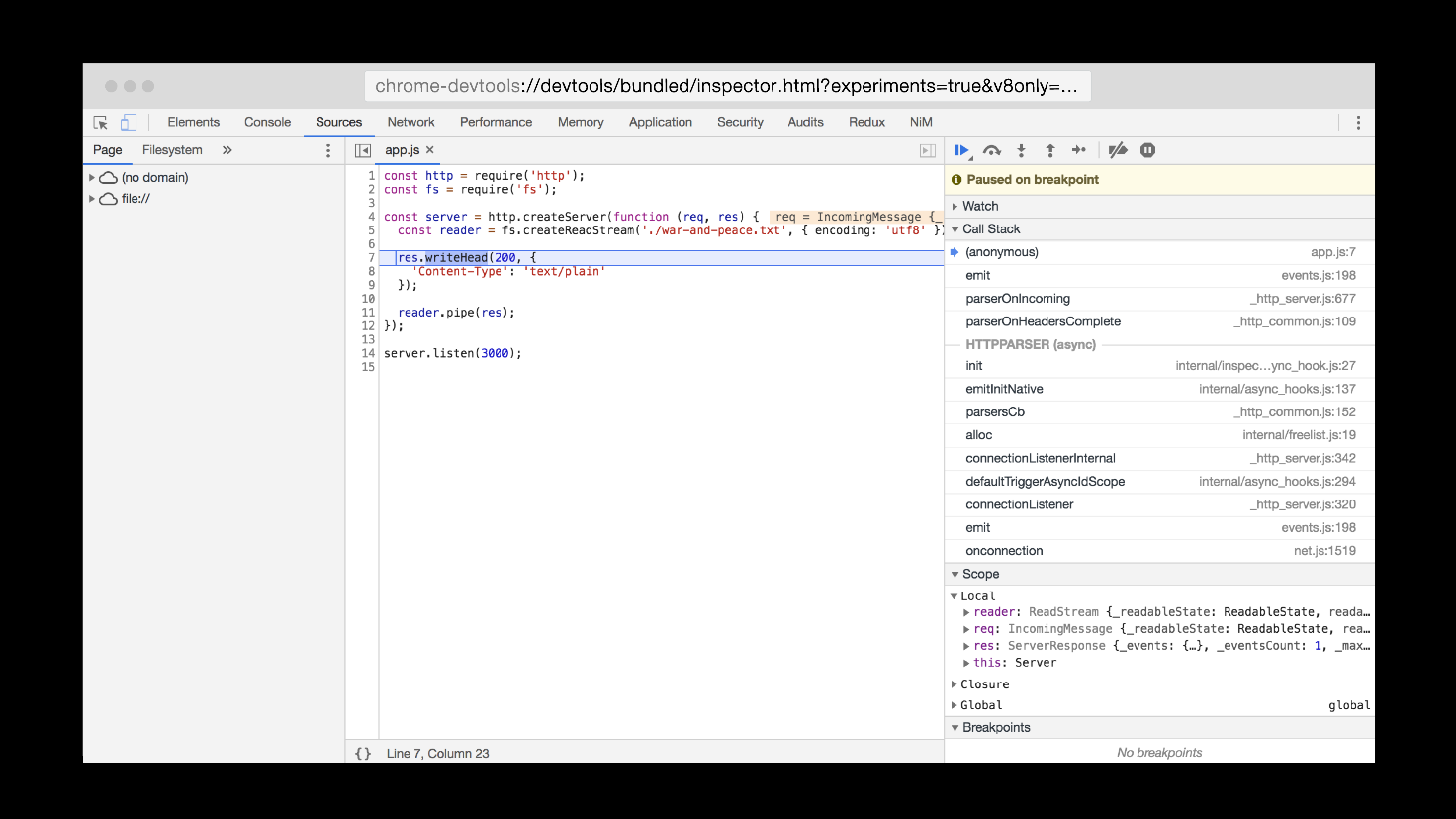
1. установите в Яндекс.Браузер или в Google Chrome расширение [Node.js Inspector Manager](https://chrome.google.com/webstore/detail/nodejs-v8-inspector-manag/gnhhdgbaldcilmgcpfddgdbkhjohddkj);
2. запустите приложение с флагом --inspect:

Скопировать кодBASH

node --inspect index.js

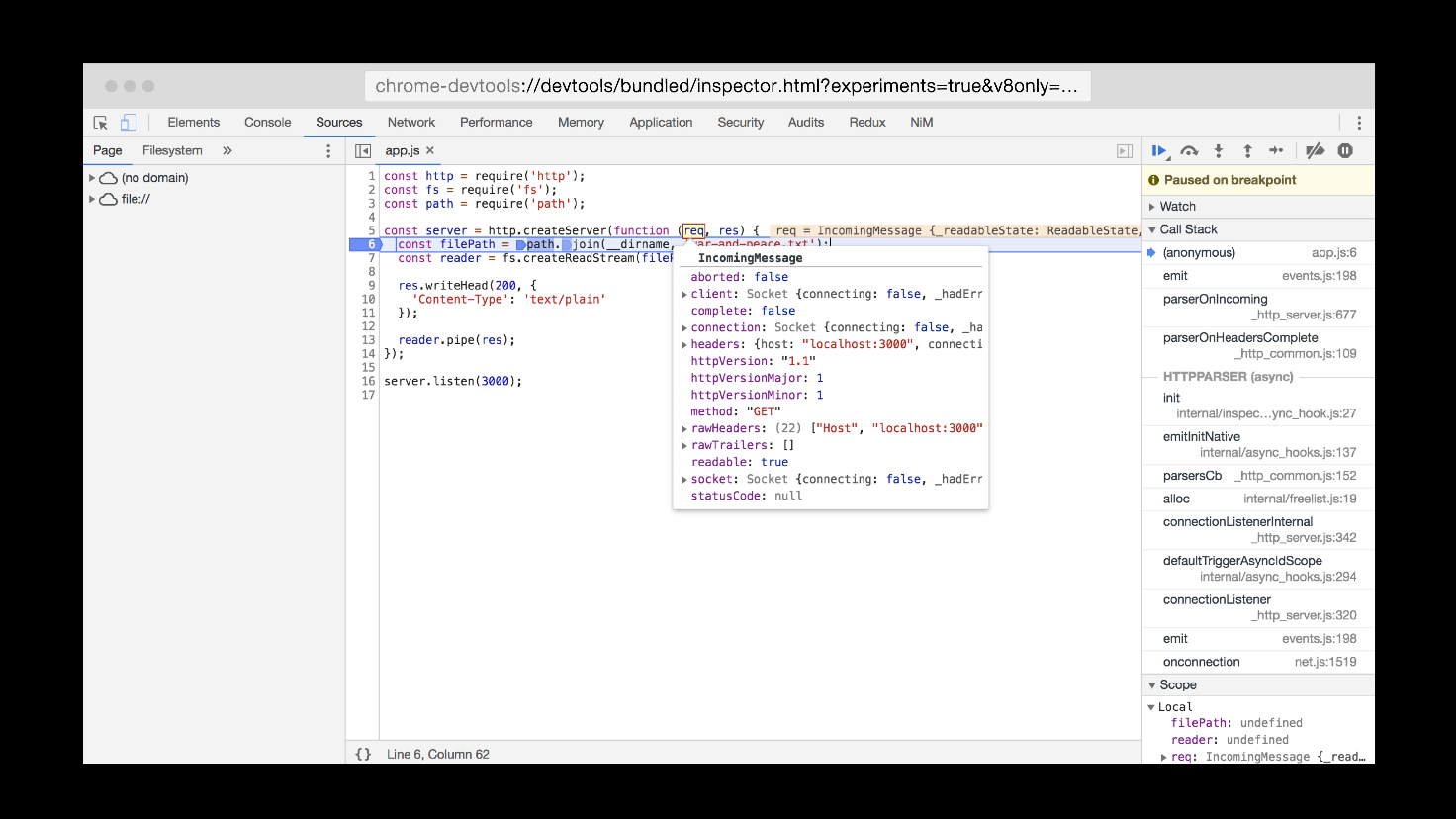


После этой команды в браузере откроется вкладка с инструментами разработчика и вашим кодом:



Здесь можно отлаживать код так же, как в браузере: например, устаналивать debugger в коде или создавать брейкпоинты.

Затем откройте другую вкладку и перейдите на http://localhost:3000. Выполнение кода прервётся в точках останова, и вы сможете узнать значения переменных, наведя на них курсор.



Если захотите вспомнить, как отлаживать JS-код, вот ссылка:

[Дебаггинг JavaScript](https://praktikum.yandex.ru/learn/web/courses/35d951a1-b62c-4a96-96ac-a8118657fad0/sprints/1702/topics/249891dc-7cdb-44b9-b0d4-3762501e42da/lessons/d3e1663f-1cd4-4144-a8b2-a1d33d3b2fcc/)

# Тестирование сервера: Postman

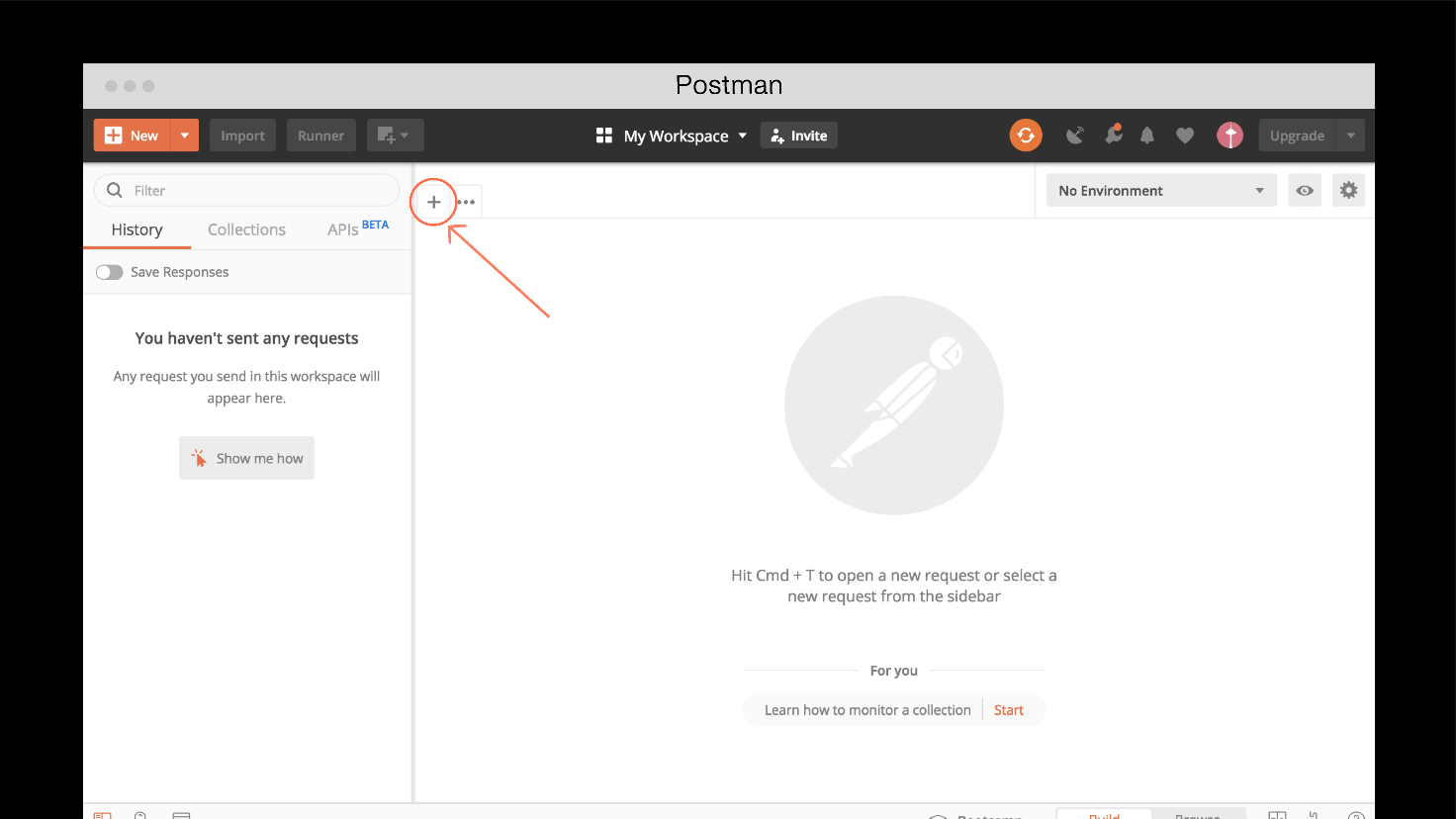
В этом уроке поговорим о тестировании: отправке запросов и проверке реакции сервера на них.

Для этой задачи подходит и браузер — вы можете вписывать GET-запросы в адресную строку, POST-запросы отправлять через форму, а методом fetch делать запросы любого типа. Но это неудобно. Чтобы узнать о запросе, например о заголовках, придётся лезть во вкладку Network в инструментах разработчика. Чтобы отправить PATCH-запрос, нужно и вовсе редактировать файл с кодом, сохранять и открывать в браузере. В общем, для тестирования нужен более гибкий инструмент.

### Инструмент для тестирования API

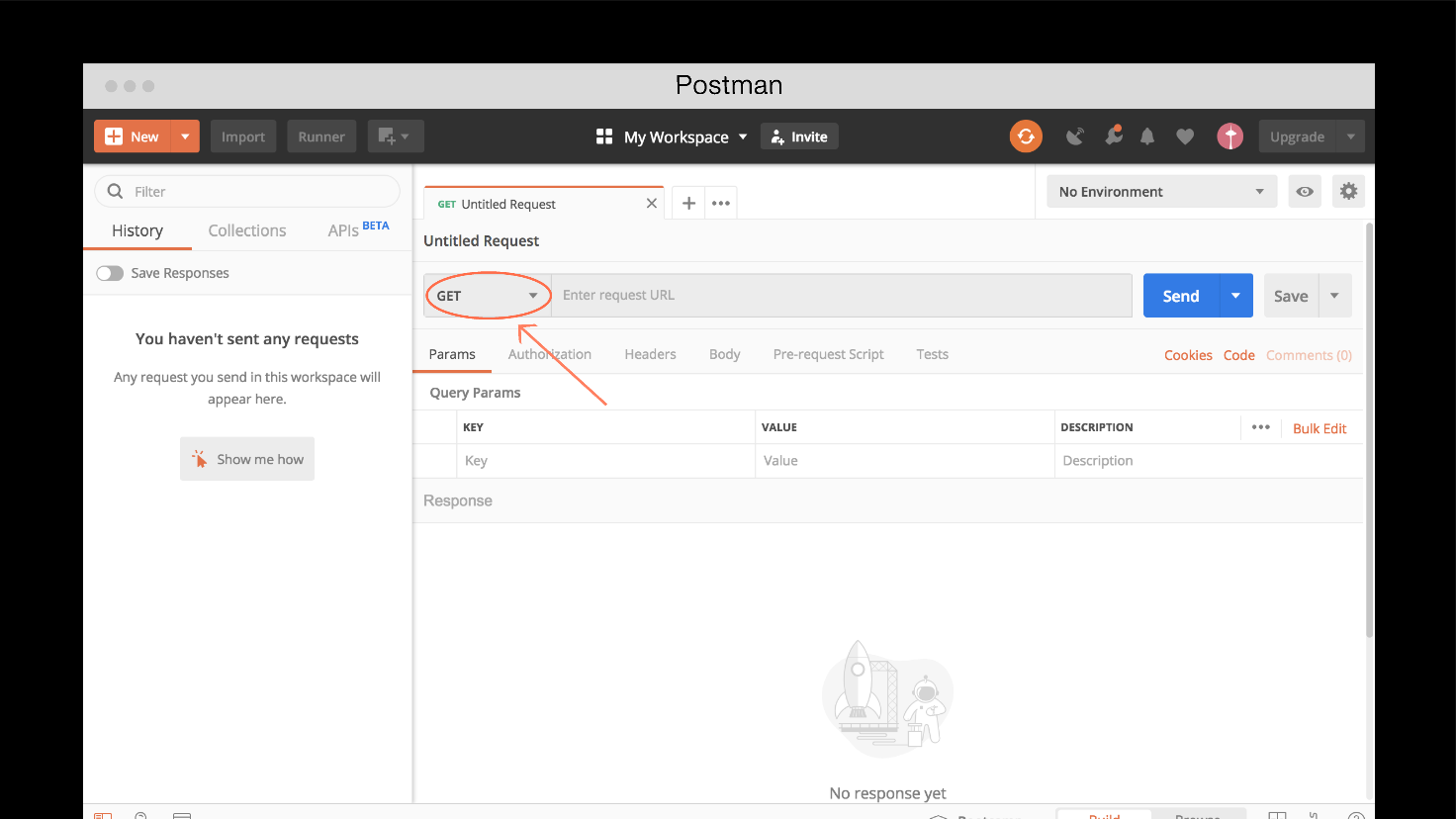
Чтобы отправлять запросы на сервер и получать ответы, есть специальные приложения. Мы расскажем об одном из самых популярных — Postman. Скачать и установить его можно по ссылке: <https://www.getpostman.com/downloads/>.

Когда установите приложение и зайдёте в него, перед вами появится такой экран:

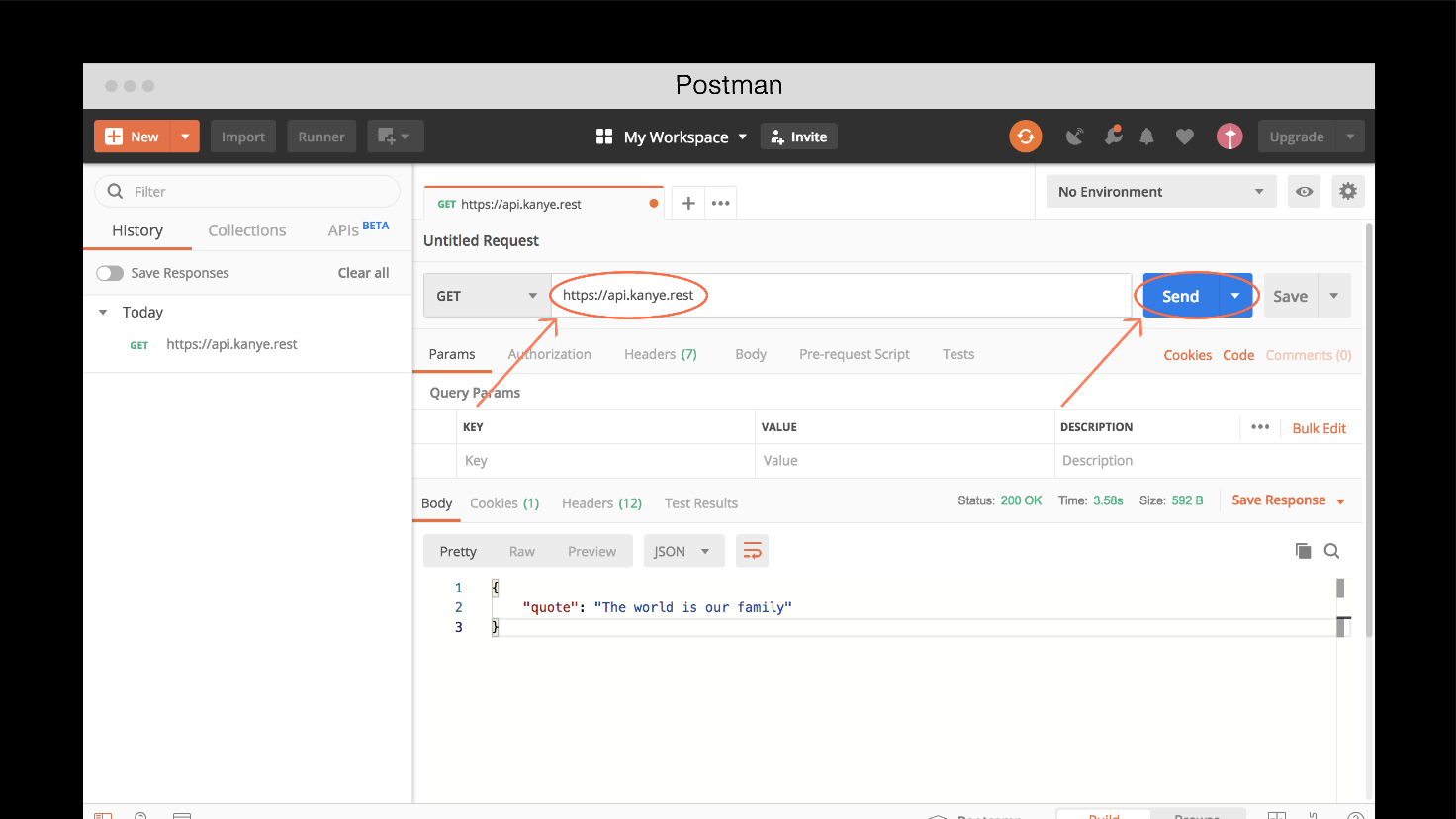


Нажмите на плюс в левом верхнем углу.

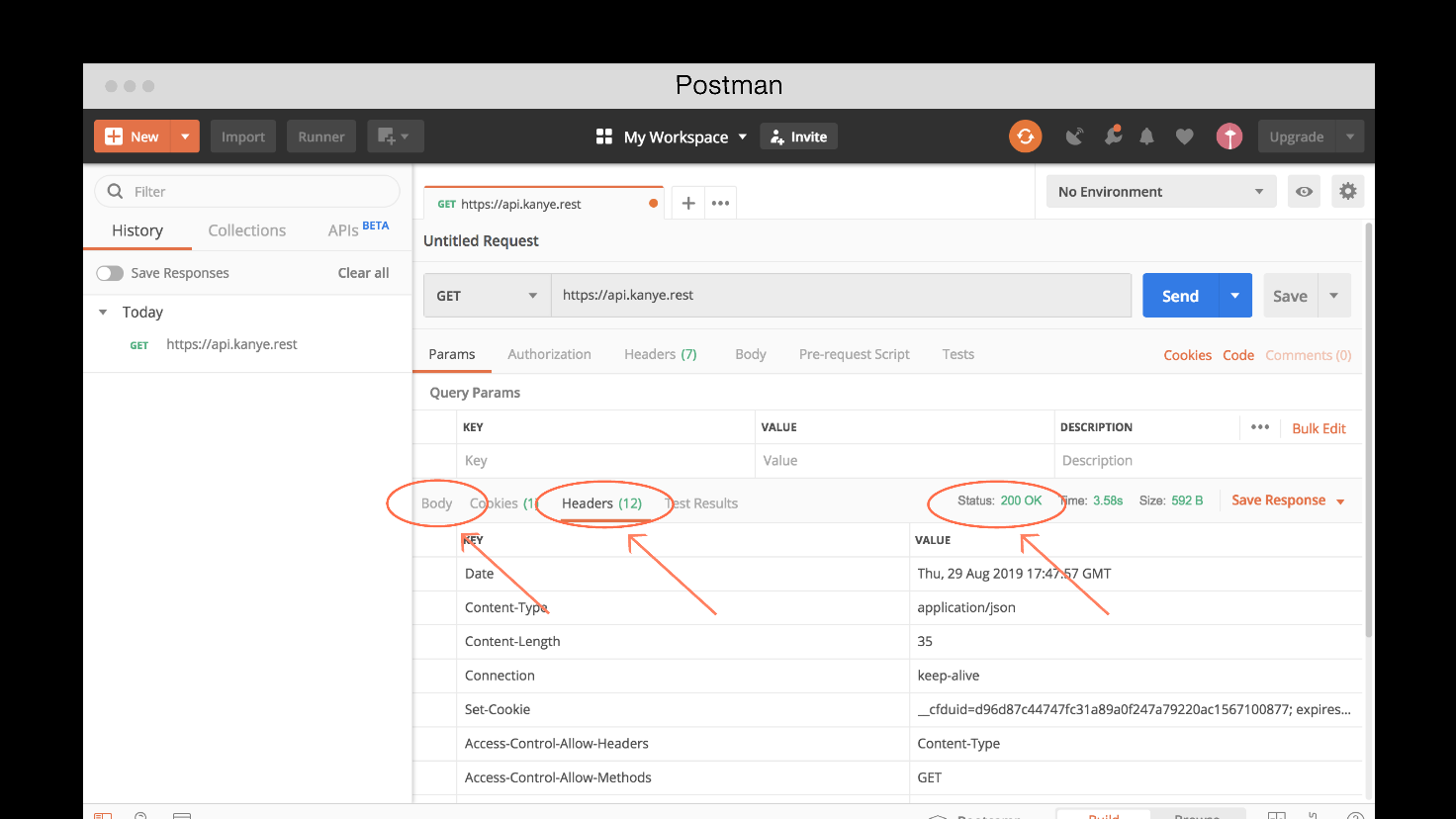
Появится окно для создания запроса. В выпадающем списке слева от адресной строки можно указать тип запроса. Пока оставим GET:



В адресной строке нужно указать URL, куда мы хотим отправить запрос. Протестируем на уже знакомом [https://api.kanye.rest](https://api.kanye.rest/). Вбейте адрес и нажмите Send:



Kanye REST вернёт ответ. В нижней панели отобразятся статус ответа, его тело и заголовки:



### Редактирование запроса

В Postman можно редактировать тело и заголовки запроса. Для примера создадим такой запрос:

Скопировать код

POST https://jsonplaceholder.typicode.com/posts

Content-Type: application/json

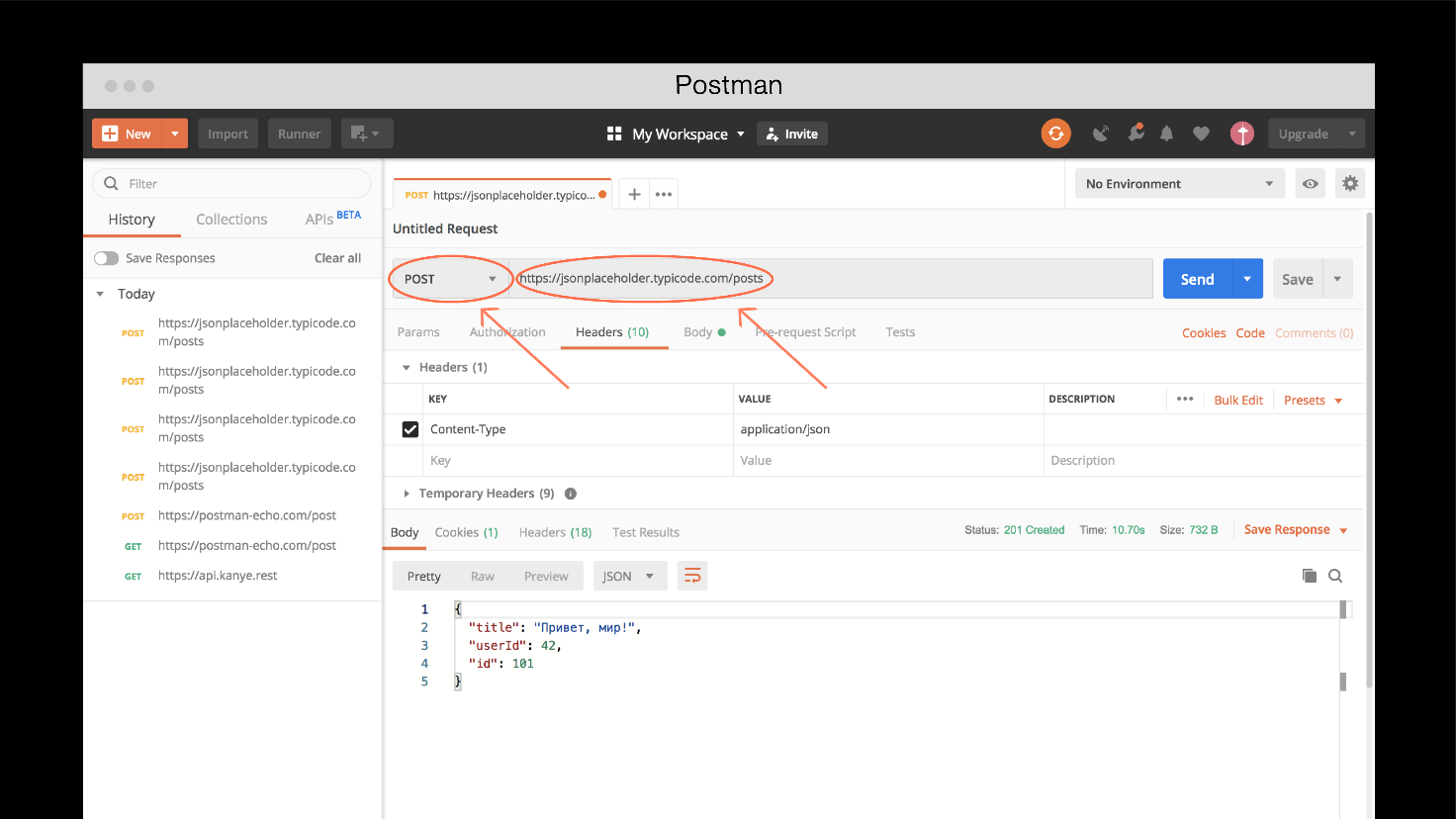
{

"title": "Привет, мир!",

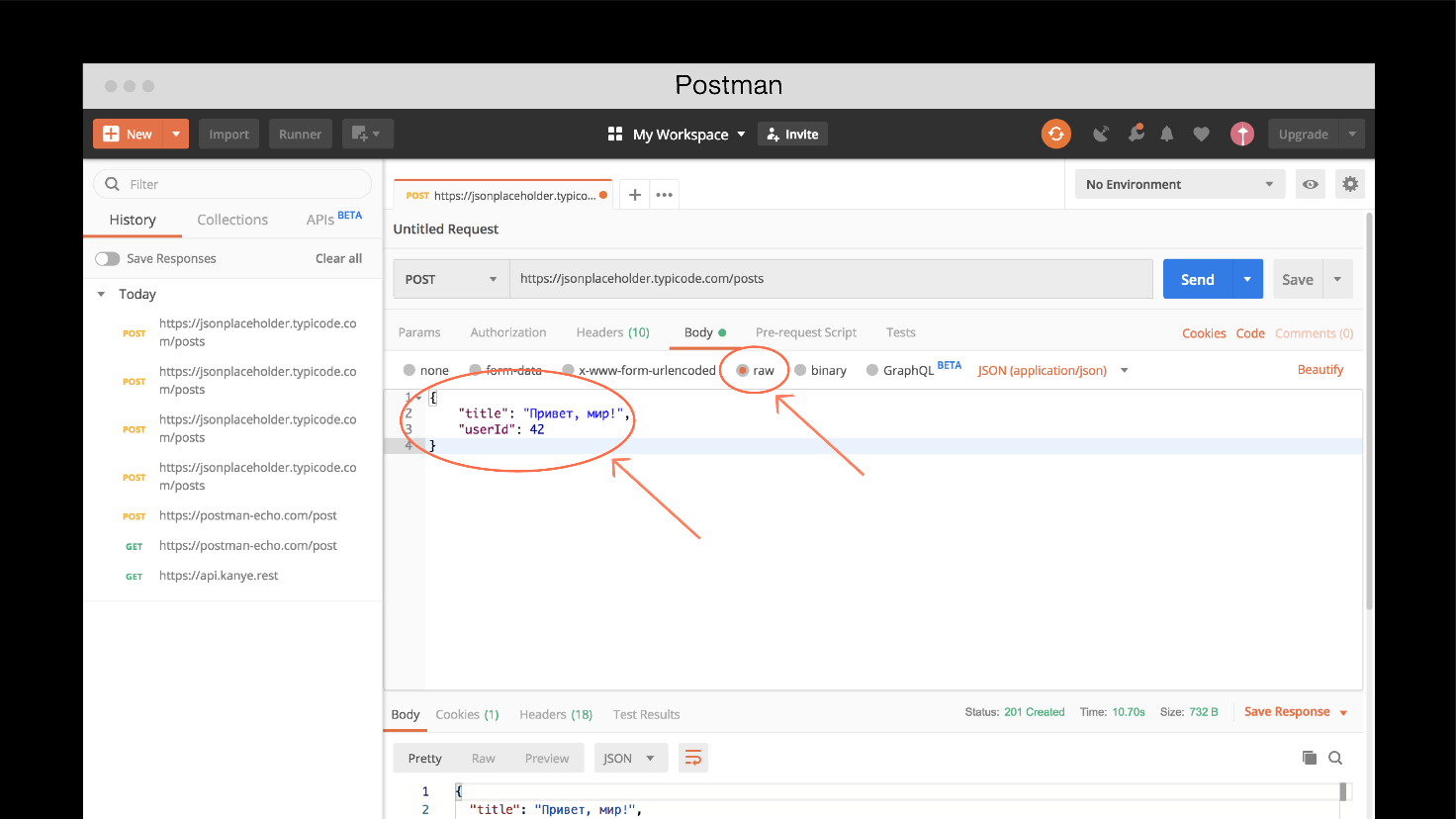
"userId": 42

}

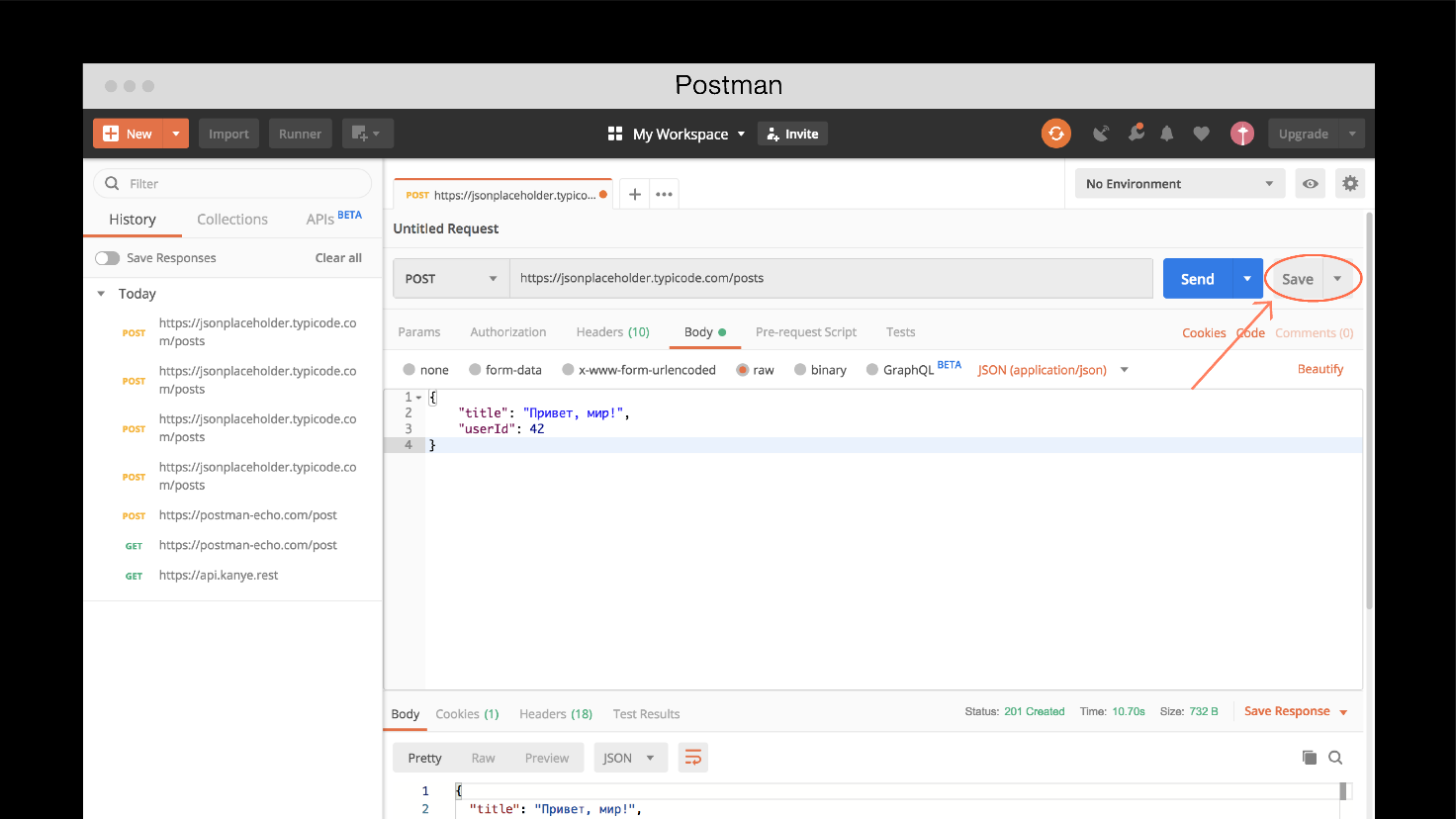
Сначала сделайте POST-запрос с соответствующим URL и заголовками:



Настройте тело запроса: перейдите во вкладку Body, выберите радио-кнопку raw и скопируйте в текстовую область ниже необходимый json-объект:



Чтобы сохранить запрос, нажмите Save. Это полезно, если вы собираетесь неоднократно тестировать сервер этим запросом:



Postman — простой и удобный инструмент для тестирования запросов. Он сэкономит много времени при разработке сервера или API.

# Компьютерные сети

Бэкенд-разработчику важно понимать, как устроены компьютерные сети.

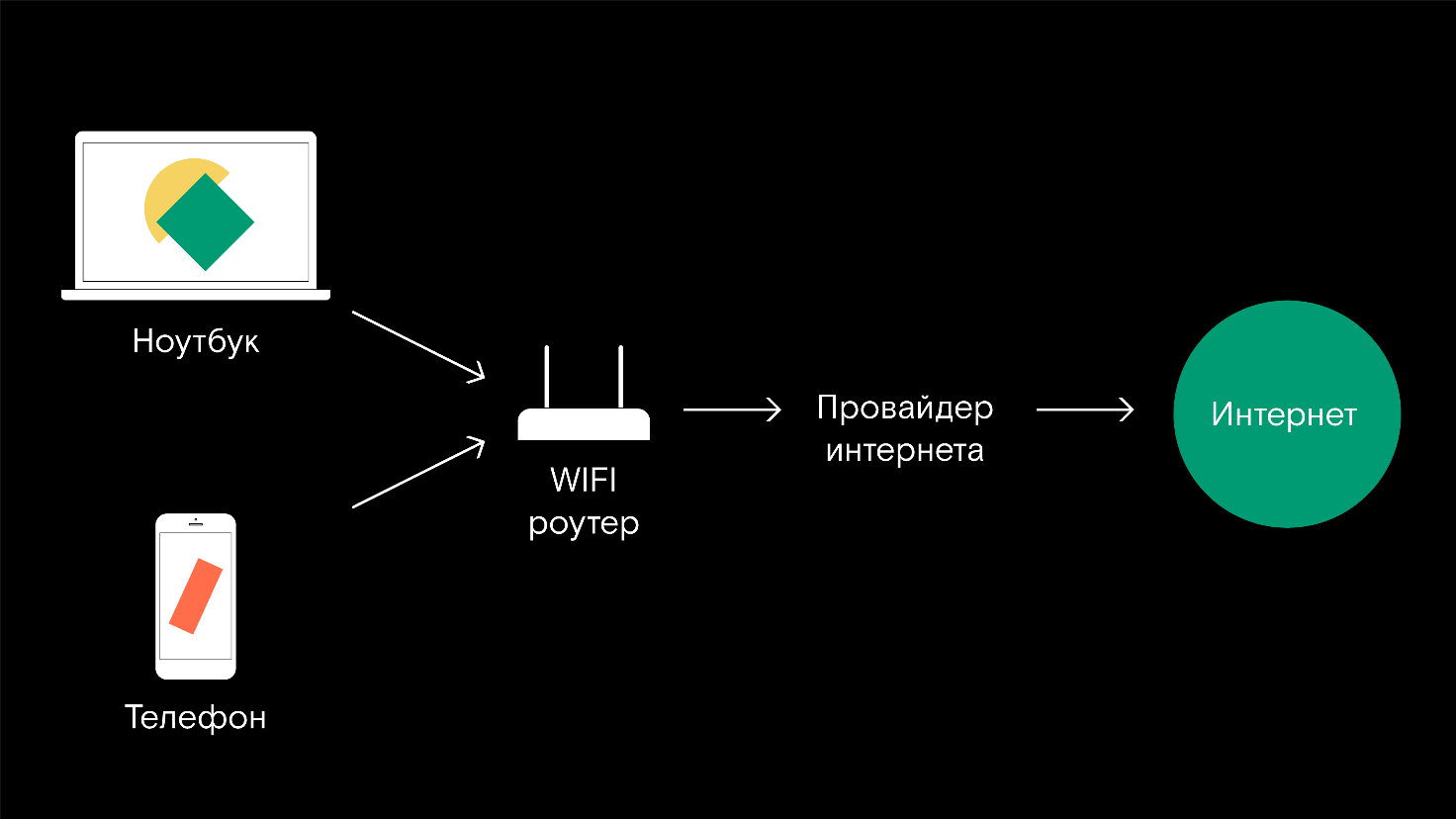
Наступит момент, когда вам потребуется защищать свой сервер от атак, настраивать шифрование и соединяться с серверами, используя различные протоколы — и всё это непременно с непроницаемым выражением лица, как у Джонни Ли Миллера в «Хакерах». Для этих задач (и чтобы добиться такой же мимики) надо понимать, какие агенты есть в интернете, как они взаимодействуют и по каким протоколам работают.

В этом уроке расскажем об общей структуре и технических принципах глобальной сети: никакой магии — только чёткий набор правил и комбинаций систем.

## Интернет с точки зрения компьютера

Вы сидите за компьютером и ищете решение очередной кодерской задачи. На столе лежит смартфон, на экране периодически всплывают уведомления в «Вотсапе» и «Телеграме». Открыл сайт, прочёл сообщение — так воспринимаем ситуацию мы. Но что происходит в это время с точки зрения устройств?

Поставим себя на их место.



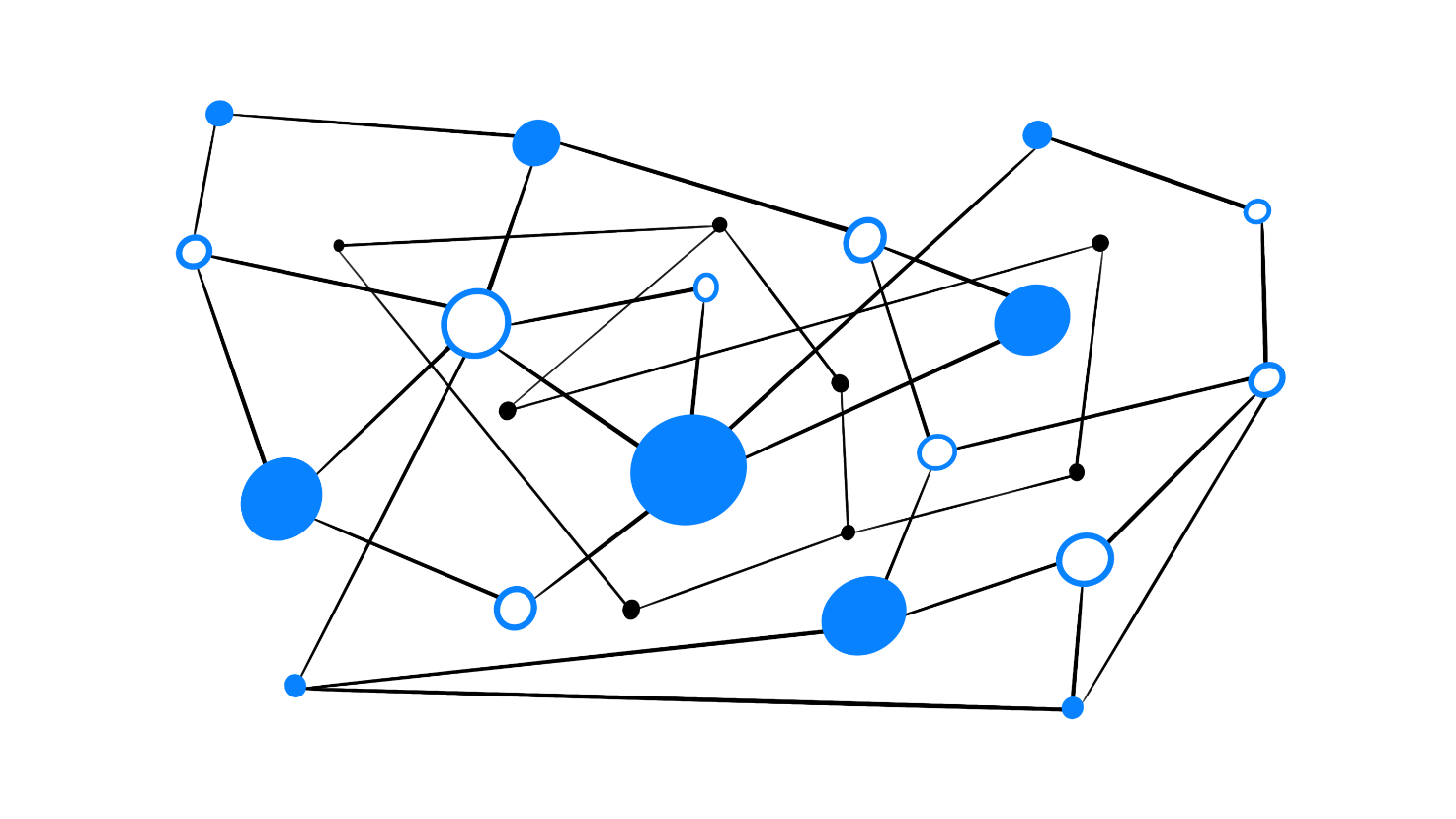
В вашей квартире наверняка есть роутер. Он обеспечивает телефону и компьютеру доступ в интернет, передавая им данные по кабелю или радиоволнам (если это Wi-Fi роутер). Роутер, в свою очередь, подключен к интернет-провайдеру, с помощью которого можно войти в глобальную сеть интернет.

У вас дома есть маленькая «локальная» сеть из роутера, компьютера, телефона и других устройств, которым вы дали доступ к сети (смарт-ТВ или смарт-утюг). Если не будет связи с провайдером — вы всё равно сможете работать внутри этой сети: при посредничестве роутера передавать файлы с компьютера на телефон, смотреть по ТВ фильмы с компьютерного жесткого диска или гладить бельё через веб-интерфейс.

Ваша домашняя сеть связана с сетью побольше: с сетью провайдера. Если провайдер вдруг утратит связь с глобальной сетью — вы (если это позволит провайдер) всё равно сможете обмениваться данными с теми компьютерами и серверами, которые подключены к провайдеру.

А пока у провайдера есть связь с внешним миром, он может передавать ваши запросы в другие компьютерные сети к разным серверам.

Получается, что интернет — децентрализованная совокупность связанных между собой больших и малых сетей.

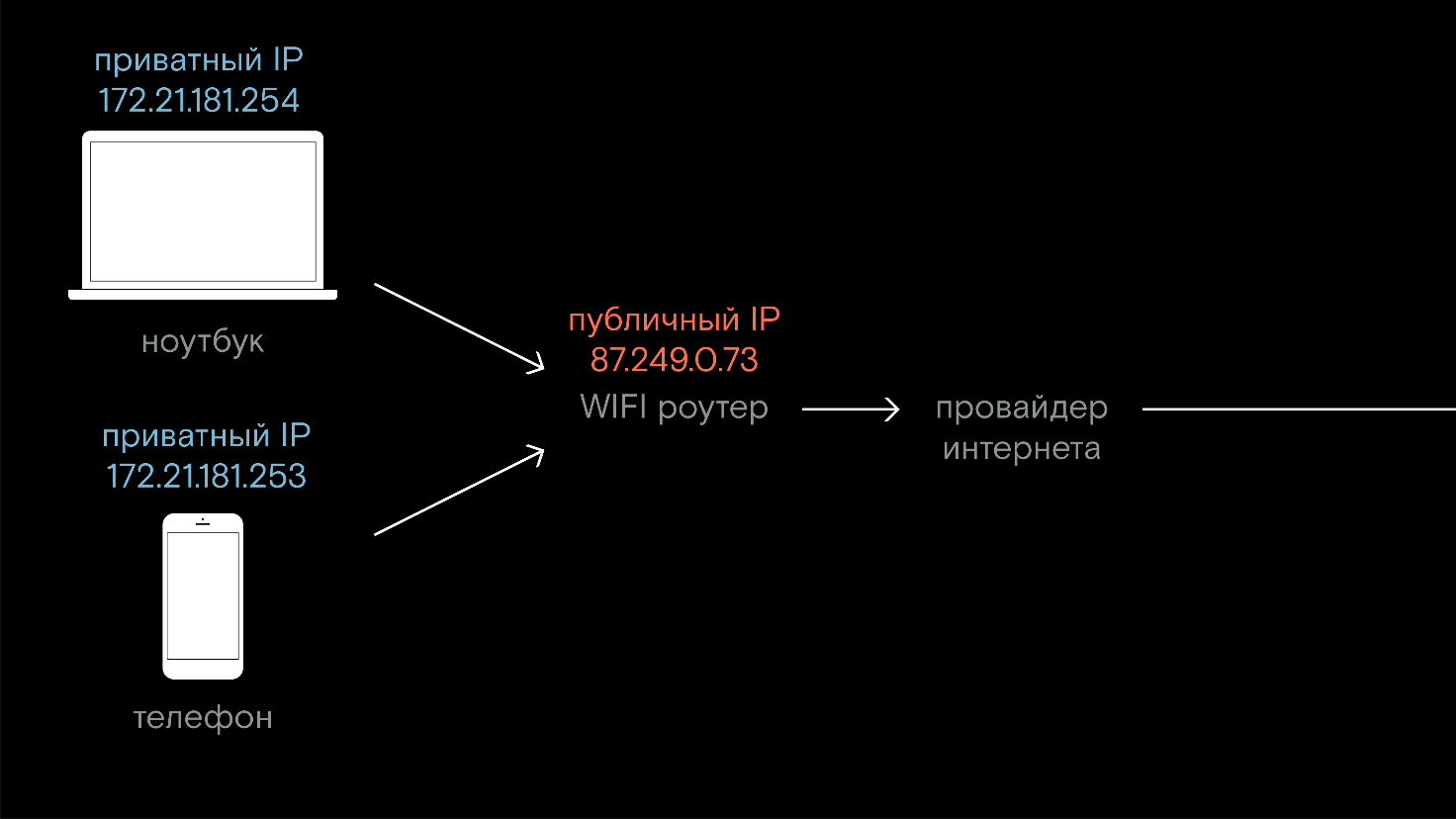


## Как найти друг друга в сети? IP-адрес

Что мы подразумеваем, когда говорим «подключиться к сети»?

Когда пользователь ищет что-то в интернете, устройство, будь то телефон или компьютер, отправляет запросы к различным серверам и получает от них ответы.

Интернет — огромная незримая «почтовая сеть», где по проводам ходят посылки с данными. Но чтобы доставить посылку, нужно знать «почтовый адрес» получателя — его IP-адрес, уникальный идентификатор устройства в сети. С его помощью вы сможете обмениваться данными с сервером.



## Неравенство адресов: публичные и приватные IP

IP-адрес есть у каждого устройства в каждой сети. В зависимости от того, в сети какого уровня они используются, IP-адреса бывают публичными и приватными.

**Публичные («белые») IP-адреса** предназначены для адресации в глобальной сети, они уникальны.

При подключении к интернету роутер получает публичный IP-адрес от провайдера. Он уникален и не может совпадать с публичным IP-адресом другого устройства — как не может быть двух домов с абсолютно одинаковыми адресами (вы тоже подумали про фильм «Ирония судьбы или С лёгким паром»?) Ведь как иначе понять, куда отправлять данные?

**Приватные («серые») IP-адреса** уникальны только в пределах локальной сети (например, в пределах вашей квартиры). Телефон и компьютер, подключённые к одному роутеру, не могут иметь одинаковые IP-адреса. Но устройства, который подключены к другому роутеру, (например, вашего соседа) могут быть с такими же IP-адресами, как и у вас.

Все пакеты данных из интернета приходят на роутер, а тот уже разбирается, какому устройству эти пакеты отдавать — компьютеру или телефону. Роутер запоминает, какое устройство из локальной сети отправило запрос в интернет и возвращает ответ именно ему. Программная служба, которая занимается этим процессом, называется NAT (Network Address Translation) и запущена на вашем роутере.

В актуальном стандарте адресации IPv4 может существовать небольшое число адресов: около четырёх миллиардов. Это адресное пространство сейчас почти исчерпано.

Чтобы решить эту проблему, придумали новый формат — IPv6. Комбинаций адресов этого формата значительно больше: 340 ундециллионов (число с 36 нулями). Сам адрес выглядит так: 2001:0db8:11a3:09d7:1f34:8a2e:07a0:765d.

## Где взять IP-адрес? DHCP-службы

При подключении к сети каждое устройство получает IP-адрес. За процедуру выдачи IP-адреса отвечает служба **DHCP** (Dynamic Host Configuration Protocol — «протокол динамической настройки узла»).

В домашней сети эта служба запущена на роутере. Если к вам в гости зайдёт знакомый и подключится к сети Wi-Fi, DHCP выдаст его устройству приватный IP-адрес.

Публичные адреса выдаёт специальный DHCP-сервер провайдера. Если вы перезагрузите домашний роутер, доступ в интернет появится не сразу. Эта задержка связана с обращением к DHCP-серверу — приходится ждать, пока роутер получит публичный IP-адрес.

IP-адрес устройства может меняться. Каждый раз, когда устройство подключается к сети, DHCP-служба роутера или провайдера выдаёт свободный адрес, который зарезервирован для этой сети. За каждым провайдером закреплён диапазон публичных IP-адресов, которые можно выдавать клиентам при подключении.

Если бы IP-адреса выдавали случайно, это привело бы к проблеме: вчера мы подключились к серверу [yandex.ru](http://yandex.ru/) и успешно искали ответы на вопросы Вселенной, но после перезагрузки сервера IP-адрес Яндекса поменялся и, выходит, мы не можем к нему подключиться. Что, если вчерашний IP Яндекса сегодня выдан онлайн-казино или порносайту?

Эта проблема решена. IP-адреса могут быть как динамическими, так и статическими. Поведение динамических адресов мы уже описали: вчера у вашего роутера был один IP-адрес, а сегодня, после перезагрузки, уже другой.

А вот статические IP-адреса навсегда закреплены за сервером и не меняются: IP-адрес можно «застолбить».

Многие провайдеры тоже предлагают клиентам услугу «статический IP»: купив такой адрес, вы сможете обращаться по IP к домашнему компьютеру.

А для приватных сетей зарезервированы несколько блоков адресов, которые не используются в пространстве публичных IP и распределением которых занимается администратор или DHCP-служба локальной сети.

## Поиск адреса по имени. DNS

Для доступа на сайты мы не вводим IP-адреса в браузер: как их вообще можно запомнить? Чтобы сделать адреса сетевых серверов более удобными придумали систему доменных имён — буквенных адресов, которыми мы и пользуемся. Мы пишем имя сайта — и специальная система DNS преобразует его в IP-адрес, понятный компьютерным системам.

Запрошенный адрес сайта передаётся провайдеру на специальный сервер — DNS (domain name system, «система доменных имён»). На нём хранится что-то вроде таблицы соответствия: какой IP-адрес за каким доменным именем закреплён.

Но провайдер не может собрать список таких соответствий для всего интернета, поэтому, если IP-адрес не удаётся найти на DNS-сервере провайдера, запрос передается «старшему товарищу» — корневому DNS-серверу или его зеркалу.

**Корневые DNS-серверы** — 13 «главных» DNS-серверов и их зеркала (копии), на которых хранится полный список соответствий «доменное имя — IP-адрес». Там обязательно найдётся нужное соответствие, результат вернётся на DNS провайдера и сохранится: в следующий раз уже не придётся обращаться на корневой сервер с тем же запросом. А запрос отправится по полученному IP-адресу.

## Новый адрес в интернете: пусть мир узнает обо мне

Возможно, вам уже сейчас необходимо доменное имя, чтобы разместить на нём «тот самый сайт, которого не хватает в интернете». Нужно обзавестись собственным доменом.

С точки зрения пользователя вы, возможно, уже знакомы с этой процедурой.

Ищете регистратора доменных имен, регистрируетесь, выбираете подходящее свободное доменное имя (что-нибудь простое и запоминающееся, например, poijafasdfhwed.io), заказываете, оплачиваете. Готово. Теперь вы владеете набором букв, которые никуда не ведут.

Чтобы разместить сайт (файлы и базу данных) в интернете, нужен **хостинг** — место на сервере. Вы покупаете и оплачиваете его отдельно — не обязательно там, где купили домен.

Для примера возьмём хостинг-сервис hosting.fake. Регистрируетесь на сервисе, выбираете тариф, оплачиваете — готово. После этого вы должны известить всемирную систему адресации (ту самую DNS, Domain Name System), что домен poijafasdfhwed.io размещён на серверах компании hosting.fake.

Для этого на hosting.fake **вы**берёте **NS-адреса** (name servers) вашего хостинга — обычно их можно найти в разделе справки или настройки — и указываете их (например, ns1.hosting.fake) в настройках домена в личном кабинете (там, где покупали доменное имя). Так вы связываете ваше доменное имя с хостингом. Это называется «делегировать домен на name servers хостинга».

Регистратор доменного имени передаёт эту информацию в систему DNS, и через некоторое время (обычно — от трёх часов до суток) данные, размещённые на хостинге, становятся доступны по доменному имени. Задержка связана с тем, что для передачи новой информации в DNS и распространения её по «зеркалам» DNS требуется время.

Интересно наблюдать, как в один и тот же момент домен откроется через одного провайдера (например, через вашего мобильного оператора), но не будет доступен через другого (например, домашнего провайдера). Информация распространяется не мгновенно и не равномерно: на всё требуется время, а магия вне Хогвартса запрещена.

После успешного добавления этой информации в DNS, пользователь набирает адрес poijafasdfhwed.io в браузере и запрос идёт к DNS провайдера:

— На каком IP этот домен? — Не знаю, спрошу у корневого, — отвечает DNS провайдера и отправляет запрос к географически ближнему корневому серверу или его «зеркалу». Тот отвечает: — Не знаю, но знаю того, кто знает, — и запрос отправляется дальше по цепочке. Путь приводит к нейм-серверу вашего хостинга, ns1.hosting.fake, а тот уже указывает на IP того сервера, где лежит ваш сайт. Готово, нашли.

Ответ сохраняется в DNS провайдера, и в следующий раз он уже ни у кого не будет спрашивать.

## Маршрут построен

Если бы ваш компьютер и сервер напрямую соединялись проводом, можно было бы отправлять данные прямо по нему. Но интернет — сложная сеть, где невозможно соединить каждый компьютер с каждым сервером. Данные приходится передавать через множество узлов, и каждый раз определять, какому именно узлу их отправить. Чтобы доставить данные, IP-адреса недостаточно — нужно определить маршрут.

Запрос от вашего роутера поступает на роутер провайдера, тот по адресу назначения запроса определяет оптимальный промежуточный узел, куда его нужно передать. Тот определяет следующий. И так — пока запрос не дойдёт до получателя: пакет данных может пройти через десятки серверов, прежде чем попадёт в цель.

Определение оптимального пути называется **маршрутизацией**. Решение «куда дальше отправить пакет» принимается на каждом промежуточном узле (включая домашний роутер).

Определить, через какие узлы прошёл запрос, можно с помощью терминала. Откройте его и введите команду:

*# Mac OS и Linux*

traceroute yandex.ru

Скопировать кодBASH

*# Windows*

tracert yandex.ru

В командной строке вы увидите адреса серверов, через которые прошёл запрос, прежде чем достичь цели — сайта [yandex.ru](http://yandex.ru/).

Мы выяснили, как запрос идёт от нашего устройства до сервера назначения. Теперь разберёмся, что представляет собой этот запрос.

## Протокол передачи данных TCP

За передачу данных отвечает TCP (Transmission Control Protocol — «протокол управления передачей»).

**Протокол TCP** устанавливает связь с сервером по процедуре трёхэтапного согласования ("three way handshake"):

* клиент посылает на сервер специальный запрос на синхронизацию;
* сервер получает запрос и отвечает;
* клиент посылает еще один сигнал, чтобы подтвердить, что получил ответ сервера.

Связь считается установленной, если все три этапа прошли успешно. Если возникает ошибка, всё начинается заново.

После того как связь установлена, начинается передача данных. Сначала их разбивают на сегменты. Эти сегменты нужно доставить в целости и в нужном порядке. Если какие-то пакеты потерялись по пути к серверу, их отправляют заново. Если данные пришли не в том порядке, их упорядочивает сервер. Если бы этого механизма не было, в сообщениях в «Вотсапе» или «Телеграме» все буквы были бы перепутаны.

Из-за того, что данные приходят в том виде, в каком их отправили, TCP называют надёжным протоколом.

За надёжность приходится платить скоростью. Иногда это критично: например, при видеозвонках и трансляциях. Важно, чтобы данные приходили в режиме реального времени. И ничего страшного, если часть данных пропадёт: мы просто не увидим один из кадров, и звук исказится на долю секунды. Для такой коммуникации обычно применяют другой протокол передачи данных — **UDP.** Он куда быстрее, но менее надёжен.

## TCP/IP модель

Мы выяснили, что представляют собой IP и TCP по отдельности. На базе этих протоколов построили модель передачи данных через интернет — TCP/IP модель. Она включает в себя 4 уровня:



Каждый уровень — это набор протоколов. Разберём их все:

* **Канальный.** Этот уровень ближе всего к железу. Он определяет, как разделить данные на пакеты и закодировать для передачи по сети. Он помогает упаковать информацию так, чтобы промежуточные узлы понимали, куда её следует отправить, а принимающая сторона смогла правильно её интерпретировать. Протоколы этого уровня: WLAN (для подключения по Wi-Fi) и Ethernet (по проводу).
* **Сетевой.** Данные закодированы, теперь необходимо выстроить логистику передачи пакетов, логику самой сети. Она строится на протоколе IP (Internet Protocol). Этот протокол определяет, каким образом компьютеры объединяются в сеть и как осуществляется маршрутизация пакетов в сети. IP-адрес — часть протокола IP.
* **Транспортный.** Когда сеть готова, в ней можно обмениваться информацией. Для этого и нужен транспортный протокол. Он отвечает на вопрос: «Как передавать сообщения?», и включает в себя протоколы TCP (надежно, но неторопливо) и UDP (быстро, но с возможными потерями).
* **Прикладной.** Это уровень, с которым взаимодействует пользователь. Он включает в себя протоколы HTTP, FTP и SSH.

Данные, которые вы отправляете в запросе к поисковику или в фейсбучном сообщении, обрабатываются всеми уровнями: от канального до прикладного. Работа каждого уровня регулируется единым набором правил. Именно благодаря этим правилам мы можем пользоваться интернетом.

**Заключение**

Мы понимаем, почему после этой темы желание заниматься бэкендом могло пропасть. Наверняка сейчас вы думаете примерно вот о чём: «Неужели всё и дальше будет так сложно? Неужели, чтобы поработать с телом запроса, нужно будет каждый раз собирать его по частям? А при работе с файлами всё время думать, как комбинировать потоки друг с другом?»

Иногда вам действительно придётся это делать. Но на этом плохие новости заканчиваются, теперь о хороших:

* Вы уже умеете нативно реализовывать эти операции и понимаете, как они работают. Без этого профессионалом не стать.
* На чистом Node.js серверный код пишут редко. Обычно задействуют фреймворк — готовый код для типовых задач, он сильно упрощает разработку. Многое из того, что вы реализовали в этой теме, есть внутри фреймворков «из коробки» — бери и пользуйся.

Позади сложная часть: вы поняли, как всё работает. А впереди приятная часть: вы осознаете, как просто «сколотить» всё на фреймворке. В следующей теме мы познакомим вас с Express *—* самым популярным фреймворком для разработки серверов на Node.

**Express. Введение**

В предыдущей теме вы разобрались с Node.js: научились поднимать сервер и работать с файловой системой. Нода даёт широкие возможности, поэтому может показаться, что её достаточно для решения любых задач и другие инструменты не нужны.

Но это не так. Программирование сервера во многом сводится к решению типовых задач: настроить авторизацию или реализовать логирование запросов (запись в базу данных). Поскольку разработка — дело творческое, программисты решают типовые задачи по-разному.

Такой подход может привести к проблеме с поддержкой. Например, разработчик написал код для авторизации пользователей, а потом ушёл в другую компанию. Вы приходите на его место и не понимаете, какая логика заложена в коде. Придётся либо всё переделывать, либо очень долго вникать в чужую работу, а это — головная боль для вас и уйма пущенных на ветер денег — для работодателя.

К тому же разрабатывать сервер придётся намного дольше: все детали каждый раз нужно прописывать вручную. Программистам, может, и нравится этим заниматься, но вот компаниям точно не нравится за это платить.

В таких ситуациях помогают фреймворки — наборы стандартных решений. С ними вы можете перейти из одного проекта в другой и сразу «въехать» в логику, заложенную в программу. Кроме того, выстроить логику также становится проще и быстрее — она уже расписана и реализована, а вам нужно лишь правильно воспользоваться синтаксисом.

Но есть две крайности. Одна — писать весь код на чистой ноде. Вторая — бездумно обращаться к методам фреймворка, не понимая, как они устроены. Такой подход ещё называют monkey coding («мартышкино программирование»). Лучше не впадать ни в одну из крайностей: разработчику следует пользоваться современными инструментами и понимать, как именно они работают и что за задачи решают.

В этой теме мы научим вас пользоваться самым популярным фреймворком для разработки серверного кода — Express. С помощью него вы сможете быстрее разворачивать сервер и лучше понимать код других разработчиков — на «экспрессе» написаны тысячи серверов.

**Приложение на экспресс**

Начнём с практики. В этом уроке развернём сервер с помощью экспресса.

**Устанавливаем экспресс**

Экспресс — пакет ноды, поэтому его устанавливают из NPM:

*# Инициализируйте package.json, чтобы нода могла устанавливать зависимости*

npm init -y *# флаг -y ответит «Да» на все вопросы по настройке package.json*

*# Установите экспресс*

npm install express

**Настраиваем hot reload**

Когда вы работали с проектами на «Реакте», то пользовались так называемым dev-режимом сборки: запускали команду npm start в Create React App. Вы могли заметить, что в этом режиме сайт автоматически обновлялся при сохранении любого файла. Такое поведение называют "hot reload" (от англ. «горячая перезагрузка»). В Create React App это делал Webpack, встроенный в сборку Create React App. В проектах на Express мы не используем Webpack — hot reload будет выполнять специальный пакет nodemon.

Hot reaload нужен только для разработки, потому установите nodemon как dev-зависимость:

*# флаг -D установит nodemon как dev-зависимость*

npm install nodemon -D

Осталось, чтобы nodemon и программа запускались вместе. Откройте package.json и отредактируйте раздел "scripts":

{

...

"scripts": {

"start": "nodemon index.js"

},

...

}

**Устанавливаем точку входа**

Создадим файл index.js и опишем в нём приложение. Для этого подключите express, а затем создайте приложение методом express:

const express = require('express');

const app = express();

Готово! Вы только что создали express-приложение.

**Проверяем, всё ли работает**

Настроим порт, который должен слушать приложение. Возьмём его из переменной окружения:

const express = require('express');

*// Слушаем 3000 порт*

const { PORT = 3000 } = process.env;

const app = express();

app.listen(PORT, () => {

*// Если всё работает, консоль покажет, какой порт приложение слушает*

console.log(`App listening on port ${PORT}`)

})

Запустим приложение:

node index.js

*# В терминале окажется сообщение "App listening on Port 3000"*

Перейдём в браузере по адресу http://localhost:3000. Мы видим сообщение "Cannot GET /*"*. Это означает, что экспресс запущен и работает, просто пока ему нечего нам отдать.

В следующем уроке исправим это — разберёмся, как настраивать роутинг.

# Как общаются клиент и сервер? Настройка роутинга

В предыдущем уроке вы создали сервер фреймворком express. Это круто, только пока он не работает: при запуске всплывает сообщение "Cannot GET /".

В этом уроке разберём настройки роутинга. Так сервер сможет отдавать веб-страницы и принимать данные от пользователя. В общем, делаем всё то же, что в первой теме, но вместо чистой ноды воспользуемся экспрессом.

Начнём с обработки GET-запроса, а запросы других типов рассмотрим чуть позже.

### Роутинг GET-запроса

Для обработки GET-запроса в экспрессе есть метод get. Он принимает на вход два аргумента:

* строку с запросом;
* колбэк, предписывающий, что нужно делать, если такой запрос пришёл на сервер.

app.get('/', (req, res) => {

*// логика обработки запроса*

});

Параметры колбэка — объекты запроса и ответа.

### Объект ответа

У объекта ответа есть метод send — он отправляет ответ. Параметром ему передают информацию, которую нужно отправить пользователю, например, веб-страницу.

app.get('/', (req, res) => {

res.send(

`<html>

<body>

<p>Ответ на сигнал из далёкого космоса</p>

</body>

</html>`

);

});

Метод send может принимать разные данные в качестве аргумента:

res.send('немного текста');

res.send('<p>немного html</p>');

res.send({ some: 'json' });

Кроме отправки ответа метод автоматически выставит заголовок Content-Type в соответствии с аргументом, который ему передали:

res.send('немного текста'); *// Content-Type: text/plain*

res.send('<p>немного html</p>'); *// Content-Type: text/plain*

res.send({ some: 'json' }); *// Content-Type: application/json*

По умолчанию метод send отправляет ответ со статусом 200. Его можно изменить методом status:

app.get('/', (req, res) => {

res.status(404);

res.send('<h1>Страница не найдена</h1>');

});

Метод status устроен так, что его и send можно использовать по цепочке. Обычно так и делают:

app.get('/', (req, res) => {

res.status(404).send('<h1>Страница не найдена</h1>');

});

Со статус-кодами вы познакомились [в этой теме](https://praktikum.yandex.ru/learn/web/courses/4da123a7-d4aa-4bc2-b299-b48f921da09c/sprints/2798/topics/5dc7307b-8486-40a5-b418-0888ac73337e/lessons/c0b00d3c-ac98-439a-b724-45efecfc896a/). Важно присваивать разным ошибкам соответствующие статус-коды. Это нужно для правильной работы любого приложения.

У объекта ответа есть и другие методы. Будем знакомиться с ними постепенно, но если что — [вот документация](http://expressjs.com/ru/api.html#res).

### Объект запроса

Запрос содержит информацию о том, что нужно пользователю. Она может быть организована по-разному, например, в GET-запросе данные пользователя «упакованы» в строку, которая хранится в свойстве query объекта запроса.

app.get('/', (req, res) => {

res.send(req.query);

});

В примере мы можем передавать любые данные в формате query string, сервер обработает их и пришлёт JSON-файл в виде строки.

### Обработка методов в express

В экспрессе есть методы для обработки любого запроса. Если хотите освежить их в памяти — [просмотрите этот урок](https://praktikum.yandex.ru/learn/web/courses/4da123a7-d4aa-4bc2-b299-b48f921da09c/sprints/2798/topics/5dc7307b-8486-40a5-b418-0888ac73337e/lessons/3164b264-9a7e-4100-838c-62480f73fd2d/). Они называются так же, как соответствующие HTTP-методы, и могут быть вызваны на приложении, где им в качестве параметров будут переданы эндпоинты и функции-обработчики:

app.get('/books', getBooks);

app.post('/books', createBook);

app.put('/books', replaceBook);

app.patch('/books', updateBookInfo);

app.delete('/books', deleteBook);

**Все говорят о роутинге. Объясняем, что это**

В предыдущем уроке вы настроили роутинг на экспрессе. Сейчас для каждого запроса есть свой обработчик: когда клиент запрашивает страницу, срабатывает соответствующий обработчик.

Такая логика подходит не всегда. Невозможно написать по обработчику для каждой страницы пользователя в соцсети или к каждому плейлисту в музыкальном сервисе. Вместо этого используют динамические роуты.

**Что такое динамические роуты?**

Раньше вы писали статические роуты, они обрабатывали запросы на конкретные адреса: /, /animals, /search.

Динамический роут — это обработчики шаблона адреса. То есть сразу всех адресов, подходящих под определённое правило.

Чтобы создать динамический роут, в URL укажите двоеточие и имя свойства, которое может меняться:

app.get('/users/:id', (req, res) => {

*// Этот код обработает все запросы, чей путь начинается с /users/111111*

});

Обратите внимание: значение переменной в адресе не может быть пустым. Вот такой запрос не будет обработан:

http://localhost:3000/users

**Параметры роута**

Обработать запрос мало — нужно получить его параметры, чтобы отдавать пользователю нужные данные. В экспрессе это просто: все параметры запроса становятся свойствами JSON-объекта req.params:

app.get('/users/:id', (req, res) => {

res.send(req.params);

*/\* Если отправить запрос на адрес "http://localhost:3000/users/123",*

*внутри объекта req.params окажется JSON-объект:*

*{ "id": "123" } \*/*

});

Формат JSON оказывается тут как никогда кстати. Особенно при обработке адресов со множественной вложенностью:

app.get('/users/:id/albums/:album/:photo', (req, res) => {

const { id, album, photo } = req.params;

*/\* При обращению к адресу "http://localhost:3000/users/123/albums/333/2"*

*параметры запроса будут записаны в таком виде:*

*{"id":"123","album":"333","photo":"2"}*

*Мы записали их в переменные id, album и photo \*/*

res.send(`Мы на странице пользователя с id ${id}, смотрим альбом №${album} и фотографию №${photo}`);

});

**Как организовать роутинг**

Мы уже не раз говорили о преимуществах модульного кода. Логику роутинга тоже следует разнести по разным файлам. Пусть один отвечает за основную часть сайта, второй — за админку, третий — за мобильное приложение.

Разберёмся, как разбить этот код на модули:

*// Здесь функциональность точки входа*

const express = require('express');

const { PORT = 3000 } = process.env;

const app = express();

*// Здесь данные*

const users = [

{ name: 'Мария', age: 22 },

{ name: 'Виктор', age: 30 },

{ name: 'Анастасия', age: 48 },

{ name: 'Алексей', age: 51 }

];

*// Здесь роутинг*

app.get('/users/:id', (req, res) => {

if (!users[req.params.id]) {

res.send(`Такого пользователя не существует`);

*// не забудем выйти из функции*

return;

}

const { name, age } = users[req.params.id];

res.send(`Пользователь ${name}, ${age} лет`);

});

app.listen(PORT, () => {

console.log(`App listening on port ${PORT}`);

});

Первым делом **перенесём данные в отдельный файл** db.js:

*// db.js*

module.exports = {

users: [

{ name: 'Мария', age: 22 },

{ name: 'Виктор', age: 30 },

{ name: 'Анастасия', age: 48 },

{ name: 'Алексей', age: 51 }

]

};

Даже четыре лишние строчки засоряют код. А пользователей на сайте обычно больше четырёх (даже если это сайт администрации малообитаемого острова в Тихом океане). Так что всегда храните данные отдельно.

**Настроим роутер.** Код роутинга тоже перенесём в отдельный файл.

Для этого код придётся дописать. Логика ответа на запросы описана в обработчиках get. Мы вызывали get как метод самого приложения — переменной app. В отдельном модуле с роутингом этой переменной нет. И создать её не получится — может быть только одно приложение. Поэтому в экспрессе есть метод Router. Он создаёт объект, на который мы и повесим обработчики:

*// router.js*

const router = require('express').Router(); *// создали роутер*

const { users } = require('./db.js'); *// данные нужны для роутинга, поэтому импортируем их*

router.get('/users/:id', (req, res) => {

if (!users[req.params.id]) {

res.send(`Такого пользователя не существует`);

return;

}

const { name, age } = users[req.params.id];

res.send(`Пользователь ${name}, ${age} лет`);

});

module.exports = router; *// экспортировали роутер*

К слову, как и в предыдущем уроке, не только GET-метод может быть добавлен к объекту Router:

router.get('/books', getBooks);

router.post('/books', createBook);

router.put('/books/:id', replaceBook);

router.patch('/books/:id', updateBookInfo);

router.delete('/books/:id', deleteBook);

**Настроим точку входа.** Для этого импортируем роутер в index.js и запустим его методом use. Он принимает на вход два параметра:

* начало URL — роутер будет запускаться только для запросов, начинающихся с этой строки;
* сам роутер:

*// index.js*

const express = require('express');

const router = require('./router.js'); *// импортируем роутер*

const { PORT = 3000 } = process.env;

const app = express();

app.use('/', router); *// запускаем*

app.listen(PORT, () => {

console.log(`App listening on port ${PORT}`);

});

Первый параметр метода use позволяет сделать код модульным. Вы можете написать разные роутеры для разной функциональности:

const express = require('express');

const router = require('./router.js');

const api = require('./api.js');

const backoffice = require('./backoffice.js');

const { PORT = 3000 } = process.env;

const app = express();

*// Для разных запросов разные роутеры.*

*// Выглядит просто отлично!*

app.use('/', router);

app.use('/api', api);

app.use('/admin', backoffice);

app.listen(PORT, () => {

console.log(`App listening on port ${PORT}`);

});

На этом всё — роутинг готов. Теперь наше приложение можно гордо называть сервером, ведь оно выполняет его главную функцию: обработку запросов.

В следующем уроке разберём, чем занят сервер между получением запроса и выдачей ответа.

**Промежуточная обработка. Как улучшить код посредством middlewares**

В заданиях предыдущего урока вы описали логику роутинга:

router.get('/users/:id', (req, res) => {

*// логика обработки начиналась*

const { id } = req.params;

if (!users[id]) {

res.send({ error: 'Такого пользователя нет' });

return;

}

*// логика обработки заканчивалась, отправлялся ответ*

res.send(users[id]);

});

Это отличный код для поставленной задачи. Но часто придётся реализовывать что-то более глобальное, например, логирование или аутентификацию пользователя. Если прописывать эту логику внутри роутеров, кода станет так много, что в нём будет сложно разобраться.

Эту проблему решают мидлвэры (middleware) — функции промежуточной обработки. Они позволяют описать логику обработки запроса в отдельном модуле, что сокращает код роутера.

В мидлвэр удобно выносить и типовые задачи. Например, для любого запроса сервер проверяет наличие записи в базе данных. Благодаря мидлвэру эту логику не нужно прописывать в каждом роутере.

**Создаём мидлвэр**

Разобьём код роутера на отдельные функции: одна служит для проверки, есть ли пользователь в нашей базе, другая — для отправки ответа с нужным пользователем.

Начнём с ответа:

*// Вынесём отправку ответа в отдельную функцию*

const sendUser = (req, res) => {

const { name, age } = users[req.params.id];

res.send(`Пользователь ${name}, ${age} лет`);

};

Переходим ко второй функции — мидлвэру. Она проверяет, существует ли пользователь:

* если нет, отправляет в ответе ошибку;
* если да, вызывает функцию отправки ответа с конкретным пользователем — уже написанную sendUser.

Переведём это на джаваскриптовый:

*// Проверим, существует ли пользователь:*

const doesUserExist = (req, res, next) => {

if (!users[req.params.id]) {

res.send(`Такого пользователя не существует`);

return;

}

next(); *// вызываем next*

};

const sendUser = (req, res, next) => {

res.send(users[req.params.id]);

};

Если пользователь найден, вызовем функцию, переданную третьим аргументом. Осталось написать обработчик запроса:

router.get('/users/:id', doesUserExist);

router.get('/users/:id', sendUser);

Тут и начинается магия экспресса. Фреймворк написан так, что если для одного запроса есть несколько обработчиков, он будет считать функцией next следующий обработчик этого же запроса.

Разберём код ещё раз. Вот функция doesUserExist:

const doesUserExist = (req, res, next) => {

if (!users[req.params.id]) {

res.send(`Такого пользователя не существует`);

return; *// если пользователя нет, мы выйдем из функции и больше ничего происходить не будет*

}

next(); *// если движок дошёл до функции next, он будет искать следующий обработчик того же запроса*

};

Допустим, нам нужен ещё один мидлвэр, чтобы проверять уровень доступа пользователя. Мы также опишем его логику и запустим колбэк next, а затем просто вставим новый обработчик запроса между уже существующими:

router.get('/users/:id', doesUserExist);

router.get('/users/:id', doesUserHasPermission); *// Просто добавили обработчик, не трогая код других мидлвэров*

router.get('/users/:id', sendUser);

Обратите внимание: колбэк мидлвэра принято всегда называть next. Старайтесь не отходить от этого правила, чтобы ваш код был понятен другим разработчикам.

**Подключаем мидлвэр**

Для использования мидлвэр мы будем пользоваться методом use, но с одним параметром — самим мидлвэром:

app.use(middleware);

*// index.js*

const express = require('express');

const routes = require('./routes.js');

const { PORT = 3000 } = process.env;

const app = express();

const logger = (req, res, next) => {

console.log('Запрос залогирован!');

next();

};

app.use(logger);

app.use('/', router);

app.listen(PORT, () => {

console.log(`App listening on port ${PORT}`);

});

**Продвинутые мидлвэры: парсер данных**

В предыдущем уроке вы написали мидлвэр для логирования данных. Но этого не придётся делать часто — для типовых задач уже есть готовые инструменты. Они позволят сэкономить время и не изобретать велосипед. О таких готовых решениях расскажем в уроке.

**Сборка пакетов: body-parser**

Как вы помните из предыдущей темы, при передаче между клиентом и сервером данные бьются на блоки, а на принимающей стороне собираются обратно. Тогда мы писали код для сборки пакетов:

const postForm = (req, res) => {

let body = ''; *// устанавливаем в переменную body пустую строку*

req.on('data', (chunk) => {

body += chunk; *// добавляем каждый приходящий пакет в body*

});

req.on('end', () => { *// когда всё пришло*

*// можем работать с объектом из колбэк метода on*

});

};

Тяжеловато. Особенно для такой базовой задачи как приём данных.

От ненужной сложности нас защищает мидлвэр body-parser. Он самостоятельно объединяет все пакеты, так что мы можем об этом не задумываться:

const postForm = (req, res) => {

const { body } = req;

*// затем мы можем просто обращаться к body как к объекту*

};

**Подключим к проекту**

Body-parser устанавливают из NPM:

npm i body-parser

Затем его нужно импортировать:

*// app.js*

const bodyParser = require('body-parser');

Все методы для работы с пакетами находятся в объекте bodyParser, остаётся только выбрать нужные: это зависит от типа данных, которые нам приходят.

app.use(bodyParser.json()); *// для собирания JSON-формата*

app.use(bodyParser.urlencoded({ extended: true })); *// для приёма веб-страниц внутри POST-запроса*

Аргументом методу bodyParser.urlencoded мы передаём объект опций. "extended: true" означает, что данные в полученном объекте body могут быть любых типов. При значении false, в свойства body могут попасть только строки и массивы. Это уже относится к более тонкой настройке, и мы не будем подробно затрагивать эту тему, но оставим ссылку на статью в дополнительных материалах.

Если вам могут прийти данные разных типов, просто задействуйте все нужные методы, — как в примере, Body-parser сам разберётся, какие данные пришли и как их собирать.

**Отдача html и статичных файлов в Express**

Мы разобрались с передачей данных в ответе на запрос. Но пользователю нужно отдавать и статичные файлы: разметку, стили, скрипты, изображения. Писать обработчики для всей этой информации неудобно и не нужно, вместо этого научимся пользоваться методом static.

По умолчанию у пользователей нет доступа к файлам на вашем сервере. Это можно изменить методом static. Он принимает адрес папки на вход и делает её доступной «снаружи»:

app.use(express.static(\_\_dirname)); *// сделали всё, что есть на сервере, доступным пользователю*

Такой код даст доступ к корню сервера, но там могут храниться данные, которые пользователям видеть не нужно, например, ключи доступа к другим ресурсам или базы данных. Файлы, которые нужно отдавать клиентам, складывают в отдельную папку — её принято называть "public".

app.use(express.static(path.join(\_\_dirname, 'public'))); *// теперь клиент имеет доступ только к публичным файлам*

Метод static задаёт определённую логику отправки файлов. Если обратиться к корню сервера, мы автоматически попадём в папку public и получим файл index.html. Получается, что не нужно добавлять в запрос имя файла, достаточно просто обратиться в корень — по адресу "/".

Публичная папка — отправная точка в проекте, которую мы уже указали с помощью express.static. Осталось добавить относительные пути ко всем файлам:

| **НЕТ** | **ДА** |
| --- | --- |
| /public/style.css | /style.css |

# Кеширование ответа сервера в express

Кеширование позволяет сохранить определённые данные на клиенте или сервере, чтобы не отвечать на один и тот же запрос несколько раз. Это ускоряет работу приложений и взаимодействие сервера с клиентом.

GET-запросы кешируются браузерами по умолчанию. POST-запросы, наоборот, по умолчанию не кешируются, а PUT, PATCH и DELETE не кешируются вообще.

За логику кеширования отвечают специальные заголовки GET и POST запросов, которые указываются на сервере. Об этих заголовках и расскажем.

### Кеширующие заголовки

**Expires**

Данные могут устаревать. Заголовок Expires позволяет настроить, как долго закешированные данные будут актуальны. После того, как этот период истечёт, данные будут запрошены с сервера, а не взяты из кеша. Сам кеш при этом обновится:

Expires: Fri, 20 May 2016 19:20:49 IST

**Cache-Control**

В этом заголовке указывают одну из директив для настройки обращения с закешированными данными. Вот наиболее используемые:

// управление кешированием

Cache-Control: only-if-cached // если ответ в кеше, взять его оттуда, не отправляя запрос

Cache-Control: no-cache // кеш сразу становится просроченным, а затем данные ответа в кеше проверяются

// управление временем жизни

Cache-Control: max-age=30000 // максимальное время в секундах, в течение которого ресурс считается актуальным

~~~~// управление обновлением кеша

Cache-Control: must-revalidate // если кеш просроченый, будет отправлен запрос на проверку данных в кеше

// другое

Cache-Control: no-store // кеш отключён

Данные от клиента к серверу идут не напрямую, а через другие машины — прокси-серверы. На этих промежуточных машинах данные тоже могут быть закешированы. Этим также можно управлять, установив заголовок Cache-Control:

Cache-Control: private // данные могут быть закешированы только на клиенте

Cache-Control: public // данные могут быть закешированы на промежуточных серверах

Cache-Control: proxy-revalidate // перед использованием данных из кеша прокси-сервера, их актуальность должна быть проверена

Cache-Control: no-transform // прокси не могут применять к ресурсу никакие преобразования. Другие настройки — private, public и proxy-revalidate — таких ограничений не накладывают

В Cache-Control можно указать сразу несколько директив через запятую:

Cache-Control: max-age=30000, must-revalidate

Кроме Expires и Cache-Control, есть два заголовка, которые отвечают за валидацию кеша — ETag и Last-Modified.

**ETag**

Чтобы при каждой валидации не отправлять кеш серверу целиком, был придуман механизм хеширования. Он заключается в следующем: определённый алгоритм анализирует кеш и на основе этого создаёт строку — [хеш](https://praktikum.yandex.ru/learn/web/courses/dbf98e55-0f76-444b-850c-4538708ad571/sprints/1425/topics/b4072eed-2089-45c0-9382-98ea71202341/lessons/72688d32-ec42-41de-a818-a219c59d88b2/) (о нём говорили в теме о технологии Git). Такая строка уникальна для любого набора данных. Можно хранить в кеше только эту строку и передавать её для валидации — это куда эффективнее.

В заголовок Etag можно записать хеш. Если тело ответа меняется, изменится и хеш:

ETag: "abcd1234567n34jv"

**Last-Modified**

Заголовок Last-Modified хранит дату последнего изменения данных на сервере, что позволяет валидировать данные в кеше:

Last-Modified: Fri, 10 May 2016 09:17:49 IST

### Кеширующие заголовки в express

Express по умолчанию работает так:

* Создаёт заголовок ответа Etag, который посылает в каждом ответе сервера. Таким образом кешируется каждый ответ.
* Значение заголовка запоминает браузер и при следующем GET-запросе отправляет Etag внутри другого заголовка — If-None-Match.
* Если значение заголовка If-None-Match совпадает с каким-то кешем на сервере, ответ придёт в статусе 304 Not Modified. И браузер возьмёт значение ответа из своего кеша.

POST-запросы по умолчанию не кешируются. Да и на практике к кешированию POST-запросов прибегают редко. Но оно возможно: достаточно вручную выставить кеширующие заголовки.

# CORS. Обработка ошибок

Сейчас наше приложение отлично работает на локальном сервере и по умолчанию оно готово принимать запросы с любых страниц.

В учёбе это не принципиально, но в реальных проектах — небезопасно. Чтобы это исправить, нужно задать заголовки для поддерживаемых сайтов с безопасными запросами. Подробнее мы рассматривали [CORS в предыдущем спринте](https://praktikum.yandex.ru/learn/web/courses/4da123a7-d4aa-4bc2-b299-b48f921da09c/sprints/2798/topics/5dc7307b-8486-40a5-b418-0888ac73337e/lessons/8f98665e-1efd-4a6c-baed-10f7ec7891b7/).

Чтобы сделать приложение более безопасным, в express ему добавляют мидлвэр:

app.use(function(req, res, next) {

res.header('Access-Control-Allow-Origin', 'https://praktikum.tk');

next();

});

В этом же мидлвэре настраивают разрешённые заголовки и методы:

app.use(function(req, res, next) {

res.header('Access-Control-Allow-Origin', 'https://praktikum.tk');

res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type, Accept');

res.header('Access-Control-Allow-Methods', 'GET,HEAD,PUT,PATCH,POST,DELETE');

next();

});

### Как разрешить запросы с нескольких ресурсов

В заголовок Access-Control-Allow-Origin можно записать либо один URL, либо сразу все. Иногда нужно разрешить кросс-доменные запросы с нескольких ресурсов, но не всех. Например, с локального сервера и продакшн-сайта.

У любого запроса есть заголовок Origin. Он содержит адрес, с которого идёт этот запрос. Мы можем создать массив разрешённых доменов, а затем проверять, есть ли среди них тот, что указан в заголовке Origin.

Если домен найден среди разрешённых, просто переписываем значение Origin в заголовок ответа Access-Control-Allow-Origin:

*// Массив разешённых доменов*

const allowedCors = [

'https://praktikum.tk',

'http://praktikum.tk',

'localhost:3000'

];

app.use(function(req, res, next) {

const { origin } = req.headers; *// Записываем в переменную origin соответствующий заголовок*

if (allowedCors.includes(origin)) { *// Проверяем, что значение origin есть среди разрешённых доменов*

res.header('Access-Control-Allow-Origin', origin);

}

next();

});

Защита от кросс-доменных запросов — это хорошо. Но ещё лучше уметь этой защитой управлять.

**Заключение**

Первое знакомство с Express позади. Работать с ним заметно проще, чем писать код на ноде без сторонних библиотек и фреймворков.

Вспомним, что вы изучили в этой теме.

**Роутеры**

Роутеры помогают удобно организовать код. Мы можем разбить их на отдельные файлы и передавать как колбэки методам обработки запросов. В экспрессе есть встроенный роутер:

const router = require('express').Router();

router.get('/users/:id', doesUserExist);

Если все роуты содержатся во входном файле, роутер можно не создавать, а передавать методу use самого приложения:

app.use('/users/:id', doesUserExist);

Роутер удобно использовать, чтобы объединять логику работы с одним эндпоинтом. Например, для работы с пользователями мы можем сделать и передать в приложение такой роутер:

const router = require('express').Router();

router.get('/', getAllUsers);

router.post('/', createUser);

router.get('/:id', doesUserExist);

router.patch('/:id', updateUser);

app.use('/users', router);

Так мы можем описать логику работы с пользователем, например, в отдельном файле, а затем передать её в приложение. В таком случае у нас будут реализованы методы GET/POST для эндпоинта /users и GET/PATCH для эндпоинта /users/:id. Удобно!

**Мидлвэры**

Основной шаблон экспресса — middleware. Это функции промежуточной обработки, которые выполняются в порядке их добавления. Мидлвэры можно добавлять всему приложению или отдельному роуту:

*// мидлвэра всего приложения*

app.use(bodyParser.urlencoded({ extended: true }));

*// мидлвэры роута*

app.use('/users/:id', checkRequest);

router.get('/users/:id', doesUserExist);

Мидлвэры роутов срабатывают только для запросов, удовлетворяющих шаблону роута, который передаётся первым аргументом, например: /users/:id.

Эти функции позволяют организовать код, чтобы его было проще поддерживать. Например, такой код проверяет, есть ли пользователь, и возвращает его, если он есть:

router.get('/users/:id', doesUserExist);

router.get('/users/:id', sendUser);

Впоследствии, если вы решите добавить функциональность, то уже не будете трогать существующий код, а просто добавите ещё один мидлвэр:

router.get('/users/:id', doesUserExist);

*// добавили промежуточную проверку*

router.get('/users/:id', doesUserHasPermission);

router.get('/users/:id', sendUser);

**Готовые пакеты**

Если вам в голову приходит какая-то функциональность, скорее всего она уже есть в готовом пакете для экспресса в npm. Такие пакеты подключают к приложению как мидлвэры. Так мы подключали body-parser:

*// подключаем body-parser как мидлвэру всего приложения*

app.use(bodyParser.urlencoded({ extended: true }));

Надеемся, вам понравилось работать с экспрессом: с ним код писать проще и быстрее. Мы будем и дальше работать с фреймворком. Эта тема — только начало.

# Дополнительные материалы

Советуем пройти этот урок уже после того, как сделаете проектное задание.

### Как решить проблемы совместимости JavaScript

В Node.js есть та же проблема совместимости, что и в браузерах. Новые возможности JavaScript появляются в ноде не сразу. Эту проблему решает удобная [таблица](https://kangax.github.io/compat-table/es6/), по которой можно проверить поддерживается функциональность или ещё нет.

### Особенности асинхронности в Node.js

Асинхронная модель ноды очень похожа на браузерную. Однако и тут есть свои нюансы. Почитать о них можно [в этой статье](https://medium.com/webbdev/js-db3d35ffed7e).

### Как передавать огромные файлы

В предыдущей теме вы познакомились с потоками. Когда разворачиваете сервер на Node.js, умение работать с потоками не нужно — в express за нас это делает парсер. Но если приходится работать с файлами весом от десятка мегабайт, потоки незаменимы. Иначе вам не построить стабильное и быстрое приложение. Разобраться с потоками поможет [этот материал](https://jscomplete.com/learn/node-beyond-basics/node-streams) (на английском языке).

### Как выстроить сложную логику роутинга

Обычно чем больше проект, тем больше в нём роутеров. Мы рассказали об азах организации роутинга по модулям. Если хотите разобраться, как разбить на модули сложную логику роутинга, советуем прочитать [этот туториал](https://scotch.io/tutorials/keeping-api-routing-clean-using-express-routers) (на английском языке).

**Введение. Зачем нужны базы данных?**

Всё это время мы программировали сервер: создавали приложение, писали роутеры, мидлвэры, разбирались с обработкой запросов. Посмотрим, зачем это было нужно в целом. Всё сводилось к одному: клиент присылает запрос и получает ответ. Ответ — что-то ценное для пользователя. И это представляет ценность, потому что содержит в себе информацию. Иными словами мы описывали алгоритмы для работы с данными.

Чтобы сервер мог отдавать клиенту данные, их нужно где-то хранить. Пока мы пользовались одним из двух вариантов:

1. Записывали данные в переменную. Пока сервер работает, всё хорошо, но стоит перезагрузить его, данные окажутся стёрты. Этот вариант не годится.
2. Хранили данные в файле. Это тоже так себе способ. Создал разработчик файл users.json и записал в него данные всех пользователей:

[

{

"name": "Ada Lovelace",

"about": "Mathematician, writer",

"\_id": "dbfe53c3c4d568240378b0c6"

},

{

"name": "Tim Berners-Lee",

"about": "Inventor, scientist",

"\_id": "d285e3dceed844f902650f40"

},

{

"name": "Alan Kay",

"about": "Computer scientist",

"\_id": "7d8c010a1c97ca2654997a95"

},

{

"name": "Alan Turing",

"about": "Mathematician, cryptanalyst",

"\_id": "f20c9c560aa652a72cba323f"

},

{

"name": "Bret Victor",

"about": "Designer, engineer",

"\_id": "8340d0ec33270a25f2413b69"

}

]

Клиент хочет получить доступ к своей учётной записи. Для этого нужно:

* 1. Прочитать файл.
  2. Пройтись по массиву объектов.
  3. При каждой итерации сравнивать свойство \_id с запрошенным.

Чем больше пользователей, тем больше нагрузка на серверный процессор, поэтому всё будет работать медленно. А для более сложных запросов описывать логику обращения к данным сложно и неудобно.

К тому же, мы хотим выбрать пользователя по идентификатору. Для этого нужно прочитать файл, пройтись по массиву объектов и, сравнивая \_id с искомым, найти пользователя.

Базы данных решают все эти проблемы. Это удобные структуры организации информации, которые позволяют быстро и безопасно работать с данными.

**Какие бывают базы. SQL и NoSQL**

При работе с данными важны скорость и надёжность. Этого не достичь без баз данных. Но прежде, чем изучать прикладные инструменты, нужно понять общую логику организации данных. Существуют два подхода: реляционные и нереляционные базы данных. В этом уроке расскажем, что представляет собой каждый подход и какой лучше выбрать.

**Реляционные БД**

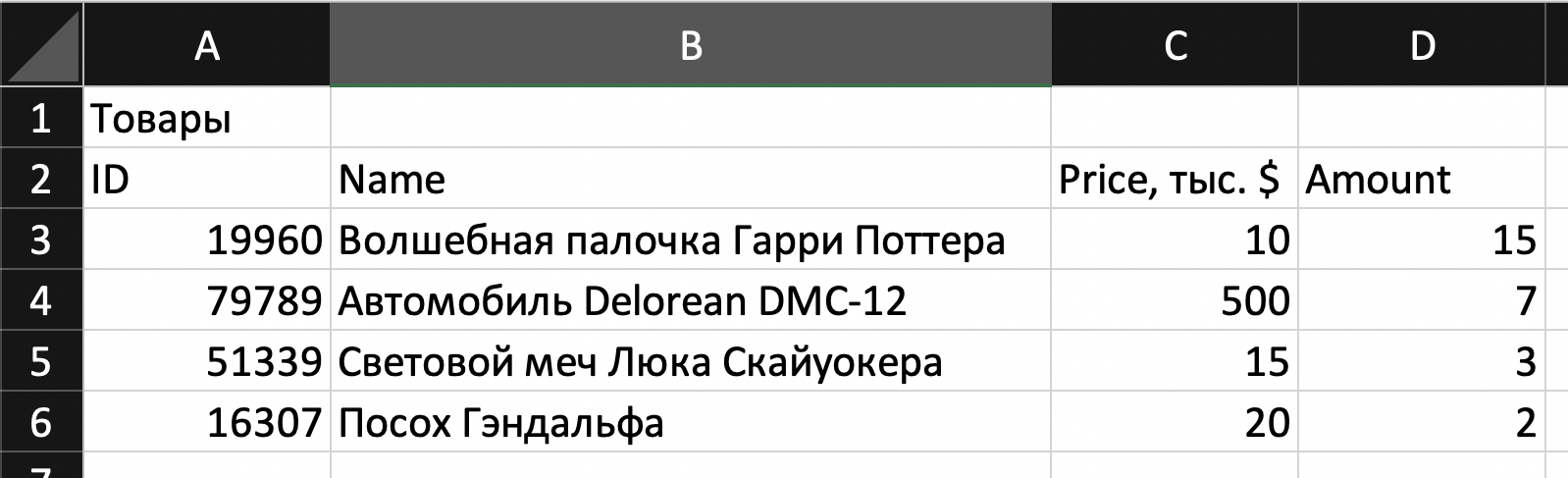
Такие базы данных состоят из таблиц и связей между ними.

К Стасу Басову приходит бизнесмен Виталий Сергеевич и просит сделать интернет-магазин. Фронтенд Стас напишет быстро, но нужно разобраться с организацией данных. Виталий Сергеевич просит: «Нужно, чтобы пользователи могли регистрироваться, а потом смотреть историю заказов».

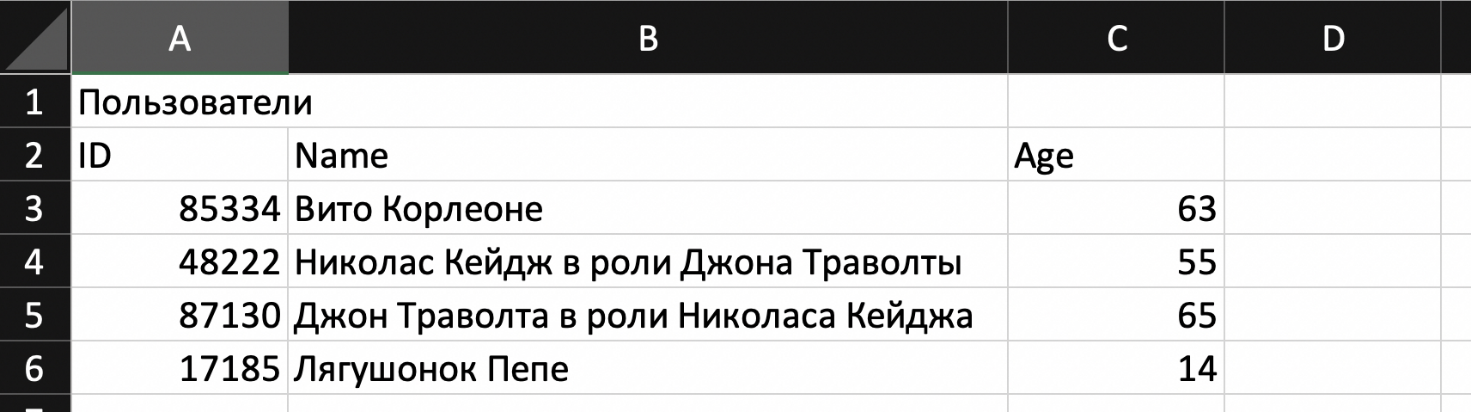
Уже в этом требовании есть 3 сущности: товары, пользователи и заказы. Поможем Стасу организовать их в реляционную БД.

При реляционном подходе данные собирают в таблице. Каждая сущность — строка в таблице. Нужно заранее определиться со структурой данных, ведь столбцы для каждой сущности будут общими.

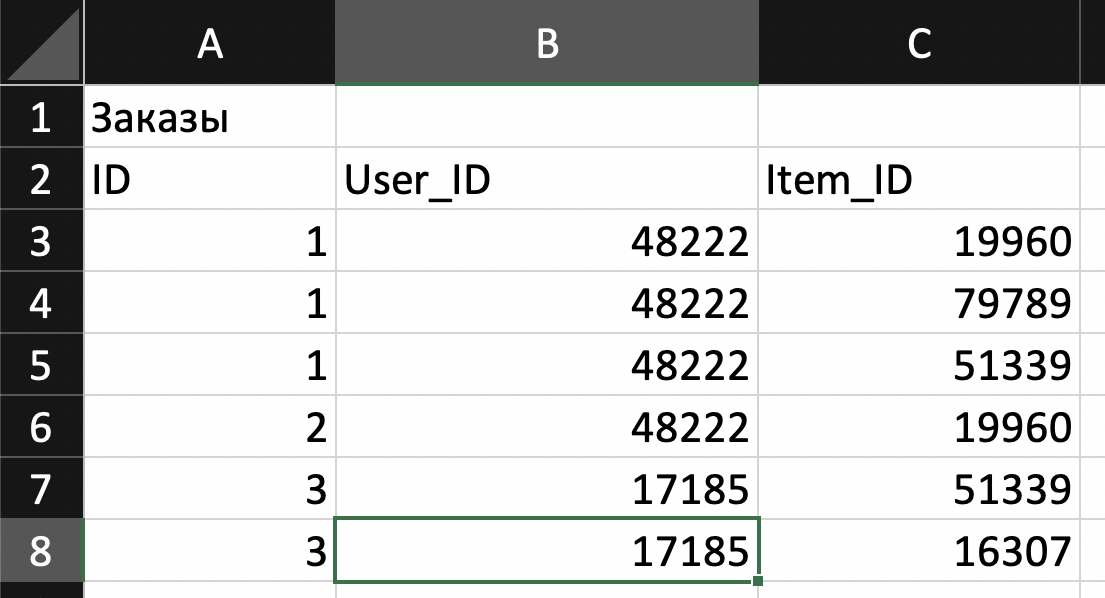
Сложим в таблицу товары:



И пользователей:



Отлично. Теперь Стасу нужна ещё одна таблица — таблица заказов. Каждый заказ относится к какому-то пользователю и содержит один или несколько товаров. Нам не нужно хранить здесь информацию о каждом товаре и пользователе. Достаточно указать идентификатор пользователя и идентификатор товара:



Чтобы составить список товаров и отрисовать его на стороне клиента, нужно сделать запрос к базе данных, который возьмёт имя каждого товара из таблицы «Товары» и сгруппирует данные по таблице «Заказы», взяв оттуда только товары нужного пользователя.

В этом и состоит суть реляционных баз данных. Информация организована в таблицы с предопределённой структурой. Мы можем добавить столбец, но тогда каждый элемент таблицы получит ещё один параметр. Мы можем получить информацию, которая нигде не содержится — список товаров с названиями. Для этого используют связи между таблицами.

Стас сдал проект, и Виталий Сергеевич доволен. Когда пользователь заходит на страницу «История заказов», он делает запрос к базе. Запрос на ходу собирает данные из разных таблиц и предоставляет их пользователю.

У такого подхода есть плюсы и минусы. Данные разделены по разным таблицам. Каждая таблица содержит информацию только о тех сущностях, за которые отвечает. В таблице «Товары» есть всё о товарах и никакой информации о заказах или пользователях. Аналогично с таблицей «Пользователи».

Тем не менее таблиц может быть десятки и сотни. Чем больше таблиц нужно обойти запросу, тем больше производительности нужно серверу. Мы можем поставить более мощный процессор и добавить оперативной памяти, но продолжать это до бесконечности не выйдет — в какой-то момент мы упрёмся в потолок производительности. Эту проблему решают нереляционные базы данных.

**Нереляционные БД**

Вместо таблиц в этих БД используют коллекции — нежёсткие структуры данных, которые состоят из документов. Объясним на примере.

К Стасу снова приходит Виталий Сергеевич и просит добавить функциональность: если пользователь совершил покупку, ему дают скидку на следующий товар.

При реляционном подходе Стас написал бы код так, что при оформлении заказа к базе отправлялся бы запрос: «Проверь, есть ли в таблице „Заказы“ вот такой идентификатор». Исходя из этого высчитывалась бы скидка.

В нереляционной БД другой подход. При заказе достаточно давать пользователю параметр, который бы говорил, что заказ уже сделан. Тогда не нужно проводить дополнительный поиск по базе заказов:

{

"id": "1234252",

"name": "Внук Виталия Сергеевича",

"madeOrder": true

},

{

"id": "1208772",

"name": "Стас Басов",

}

Это и есть документ, который хранит пользователей. Мы не привязаны к жёсткой структуре: у одного пользователя есть свойство "madeOrder", а другого — нет.

Тут Стас подумал и решил: «А переделаю всё в нереляционную БД!» Сначала он переписал таблицу с товарами в документ:

*// Документ с товарами*

{

"id": "19960"

"name": "Волшебная палочка Гарри Поттера",

"price": 10,

"amount": 15

},

{

"id": "79789",

"name": "Автомобиль Delorean DMC-12",

"price": 500,

"amount": 7

},

{

"id": "51339",

"name": "Световой меч Люка Скайуокера",

"price": 15,

"amount": 3

},

{

"id": "16307",

"name": "Посох Гэндальфа",

"price": 20,

"amount": 2

}

Затем то же сделал с таблицей пользователей:

*// Документ с пользователями*

{

"id": "85334",

"name": "Вито Корлеоне",

"age": 63

},

{

"id": "48222",

"name": "Николас Кейдж в роли Джона Траволты",

"age": 55

},

{

"id": "87130",

"name": "Джон Траволта в роли Николаса Кейджа",

"age": 65

},

{

"id": "17185",

"name": "Лягушонок Пепе",

"age": 14

}

Стасу осталось оформить таблицу с заказами. Тут и выявляется основное отличие реляционных баз данных от нереляционных. В последних мы не будем делать запрос, который пойдёт в связанные таблицы, а просто соберём все нужные данные в одном месте:

*// Документ с заказами*

{

"id": "39439",

"user": {

"id": "48222",

"name": "Николас Кейдж в роли Джона Траволты",

"age": 55

},

"items": [

{

"id": "19960",

"name": "Волшебная палочка Гарри Поттера",

"price": 10,

"amountOrdered": 2

},

{

"id": "79789",

"name": "Автомобиль Delorean DMC-12",

"price": 500,

"amountOrdered": 1

},

{

"id": "51339",

"name": "Световой меч Люка Скайуокера",

"price": 15,

"amountOrdered": 1

}

]

},

{

"id": "48241",

"user": {

"id": "17185",

"name": "Лягушонок Пепе",

"age": 14

},

"items": [

{

"id": "16307",

"name": "Посох Гэндальфа",

"price": 20,

"amountOrdered": 1

},

{

"id": "51339",

"name": "Световой меч Люка Скайуокера",

"price": 15,

"amountOrdered": 1

}

]

},

Теперь не нужно собирать данные о заказах, можно просто взять их из документа с товарами. Плюс такого метода в скорости и простоте обработки запроса. Минус — в необходимости дублировать одни и те же данные. Если пользователь изменит имя, придётся менять его и в документе с заказами.

**Какую базу выбрать?**

Общего ответа на этот вопрос нет. Всё зависит от задачи, которую вы решаете.

Реляционные БД сильно нагружают процессор и оперативную память сервера. Такие базы масштабируются вертикально, то есть заменой сервера на более мощный.

Нереляционные БД можно масштабировать горизонтально, то есть добавлять новые серверы. На одном сервере разместить пользователей, на другом — заказы, на третьем — товары. Это делает базу данных практически неограниченно масштабируемой.

Мы выбрали нереляционный подход. Почему — расскажем в следующем уроке.

# MongoDB

В прошлом уроке мы говорили, что выбор подхода к организации данных зависит от задачи. Мы хорошенько подумали и выбрали NoSQL.

### Почему NoSQL?

Раньше аббревиатура NoSQL означала Not SQL (англ. «не SQL»). Позже подход вбирал некоторые свойства реляционного подхода, и теперь NoSQL расшифровывается как Not Only SQL (англ. «не только SQL»). Современные NoSQL базы данных комбинируют все плюсы нереляционных и некоторые сильные стороны реляционных баз данных.

Для проекта Mesto важны особенности нереляционного подхода. Если Mesto станет конкурировать по популярности с Instagram, нам будет проще масшабировать свой продукт и выдерживать конкуренцию.

Кроме того, NoSQL базы хранят данные в структурах, близких JSON. Их просто обрабатывать из JavaScript, а значит, нужно меньше заморачиваться с преобразованием данных.

### Почему MongoDB?

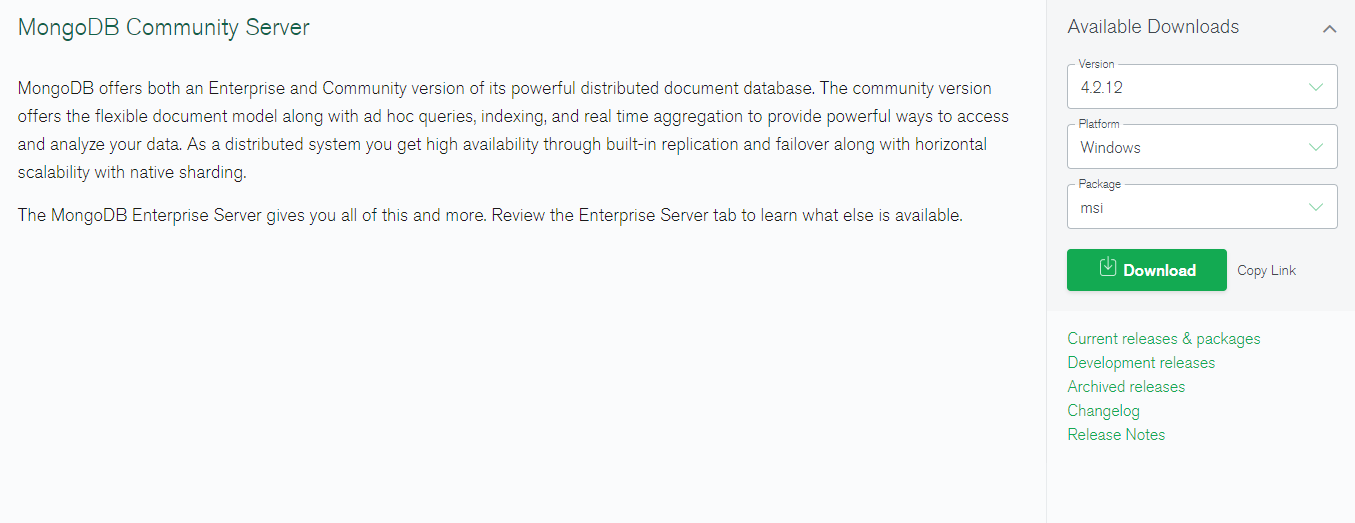
Когда подход выбран, нужно определиться с базой данных. Вариантов много: RavenDB Cassandra, MongoDB, Redis, BigTable и другие. Мы остановили свой выбор на MongoDB. Это самая распространённая NoSQL база данных среди прочих, да ещё и самая популярная среди БД, используемых с Node.js. Если в вакансии нужна Node.js, скорее всего нужна и MongoDB.

## Установка Mongo

Установите Mongo по инструкции для вашей ОС:

### Windows

**Скачайте установщик**



1. Перейдите по [ссылке](https://www.mongodb.com/download-center/community?jmp=docs)
2. в выпадающем списке Version выберите версию с текстом 4.4.5;
3. в выпадающем списке OS выберите Windows x64 X64;
4. в выпадающем списке Package выберите MSI;
5. нажмите Download.

**Запустите установщик**

|  |  |
| --- | --- |
| Запустите установщик | image |
| Нажмите Next | image |
| Поставьте галочку, что вы согласны с лицензионным соглашением и нажмите Next | image |
| Выберите Complete | image |
| Нажмите Next | image |
| Оставьте галочку около Install MongoDB Compass и нажмите Next | image |
| Нажмите Install | image |
| Дождитесь окончания установки | image |
| И нажмите Finish | image |

**Создайте директорию для базы данных**

Откройте Git Bash и перейдите в корень диска C:\

cd /C

Создайте директорию для базы данных

mkdir -p data/db

На этом установка завершена.

**Добавьте путь к папке bin в переменную окружения**

1. **Найдите путь к папке bin — в ней хранятся исполняемые файлы MongoDB.** Обычно он выглядит как-то так:

C:\Program Files\MongoDB\Server\4.2\bin

1. **Откройте настройки переменных окружения.** Нажмите клавишу win и начните вводить «перемен» ("env" для интерфейса на английском). В результатах поиска выберите «Изменение переменных среды текущего пользователя» или «Edit the System Environment Variables».
2. В верхнем окне выберите переменную Path и нажмите «Изменить...».
3. В открывшемся окне нажмите «Создать», скопируйте путь к папке bin, сохраните результат и нажмите «Ок» во всех открытых окнах. Переменная окружения добавлена.

Или воспользуйтесь [официальной инструкцией](https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/#install-mdb-edition)

### macOS

Mongo на macOS устанавливается из Homebrew — популярного пакетного менеджера macOS. Его нужно установить перед установкой Mongo. Для этого запустите терминал и запускайте там все команды из инструкции.

Если вы уже устанавливали Homebrew, сразу переходите к пункту 4.

#### **Пошаговая инструкция**

1. Установите утилиты разработчика от Apple. Для этого в терминале запустите:

xcode-select --install

1. Чтобы установить утилиты разработчика, терминал скачает большой файл. Поэтому установка займёт какое-то время — можете налить себе чаю. Хорошая новость в том, что это придётся сделать один раз.

Во время установки у вас попросят подтвердить лицензию на установку. После того как команда выполнится, возможно, понадобится ещё раз подтвердить ещё одну лицензию на использование компонентов среды разработки XCode. Для этого запустите команду:

sudo xcodebuild -license

1. Теперь ваш компьютер готов к установке менеджера пакетов brew. Скопируйте эту команду в терминал и запустите:

/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

1. Пришло время установить Mongo. По очереди запустите эти команды:

brew tap mongodb/brew

brew install mongodb-community@4.4

Или воспользуйтесь официальной инструкцией: <https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/>

### Linux

Процесс установки описан в [официальной инструкции](https://docs.mongodb.com/manual/administration/install-on-linux/).

# Взаимодействие с Mongo

Прежде чем приступить к уроку, запустите сервер MongoDB: откройте терминал (macOS и Linux) или Git Bash (Windows) и введите команду:

mongod

Если вы устанавливали Mongo на macOS через brew и команда mongod не запустила сервер, воспользуйтесь альтернативой. Но обратите внимание на номер версии mongodb — она должна совпадать с установленной:

brew services start mongodb-community@4.2

Сервер запущен. Можно приступать к чтению.

## Как взаимодействовать с MongoDB

Есть несколько способов. Мы расскажем о двух: графическом интерфейсе и через сервер на ноде.

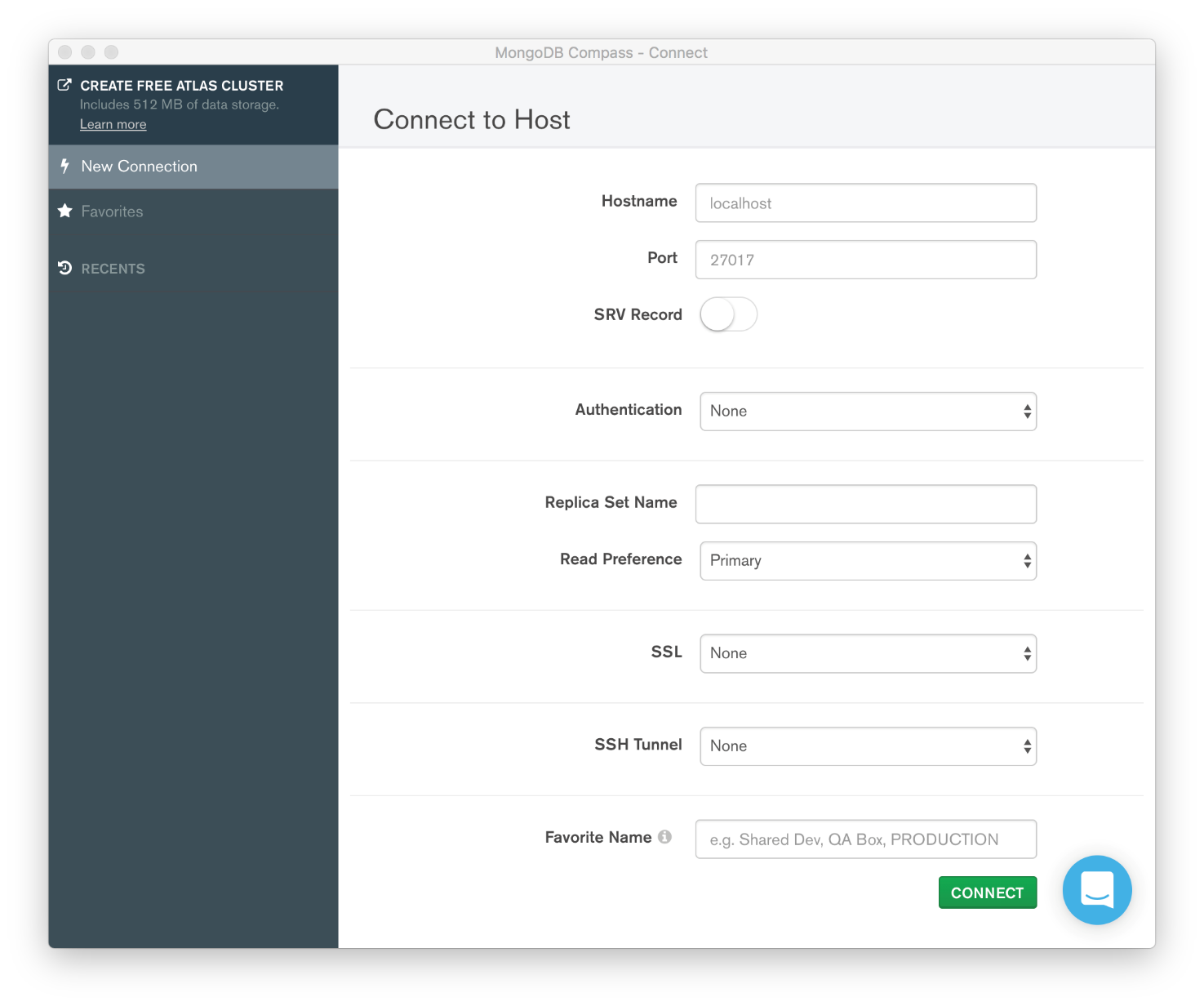
### Графический интерфейс

Это очень наглядный способ. Через графический интерфейс можно просматривать и редактировать базы данных и записи в них. От слов — к делу.

Если работаете на macOS или Linux, скачайте и установите Compass — графический интерфейс для MongoDB. На Windows он уже установлен вместе с самой Mongo.

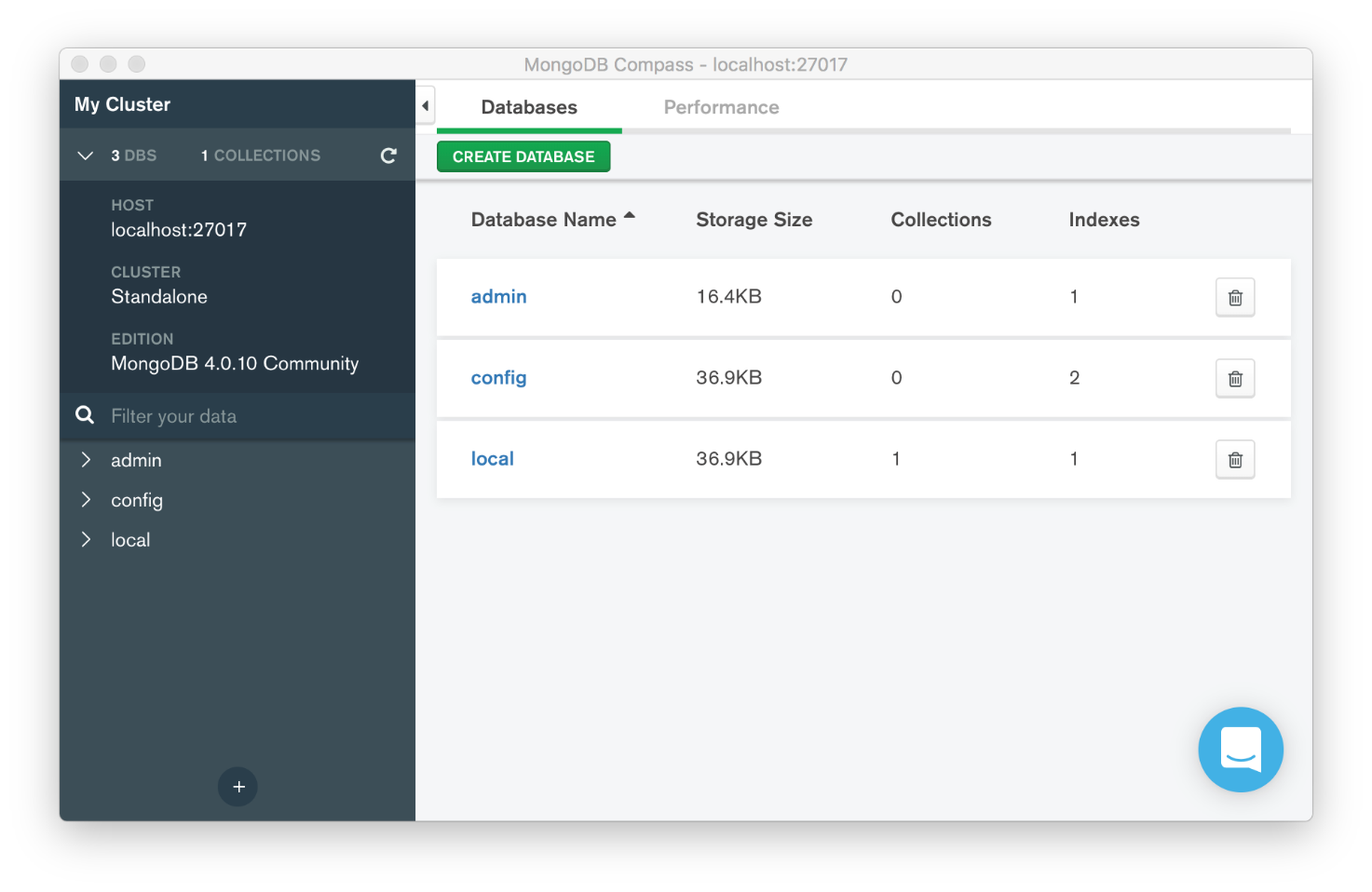
Ссылка на установщик Compass: <https://www.mongodb.com/download-center/compass>.

Откройте Compass. Перед вами окажется такой экран:

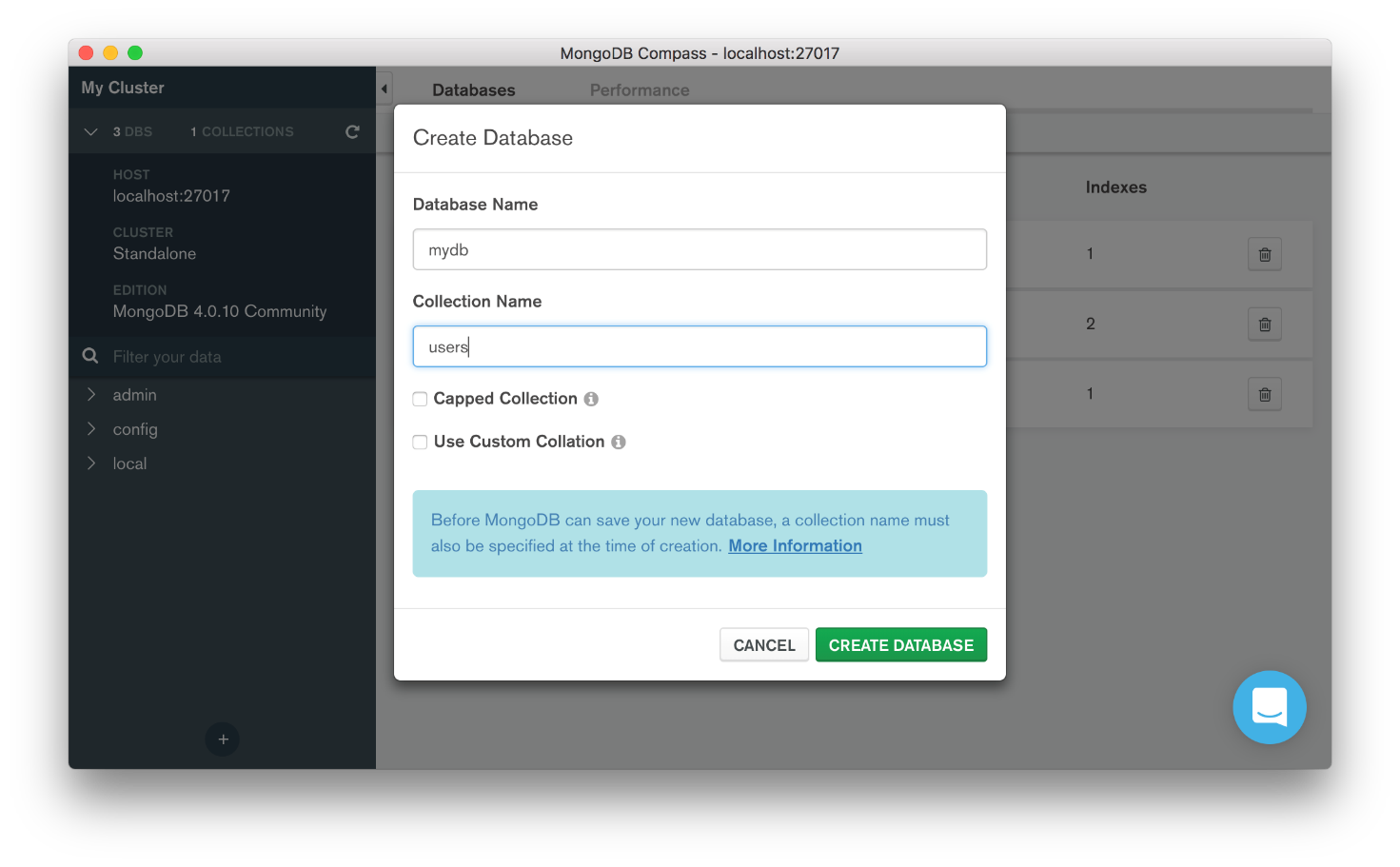


Оставьте всё как есть и нажмите Connect: Compass подключится к MongoDB серверу. Если этого не произошло и появилась красное сообщение об ошибке, проверьте, запущен ли MongoDB сервер.

Compass покажет уже имеющиеся базы данных. Создадим ещё одну. Нажмите Create Database:

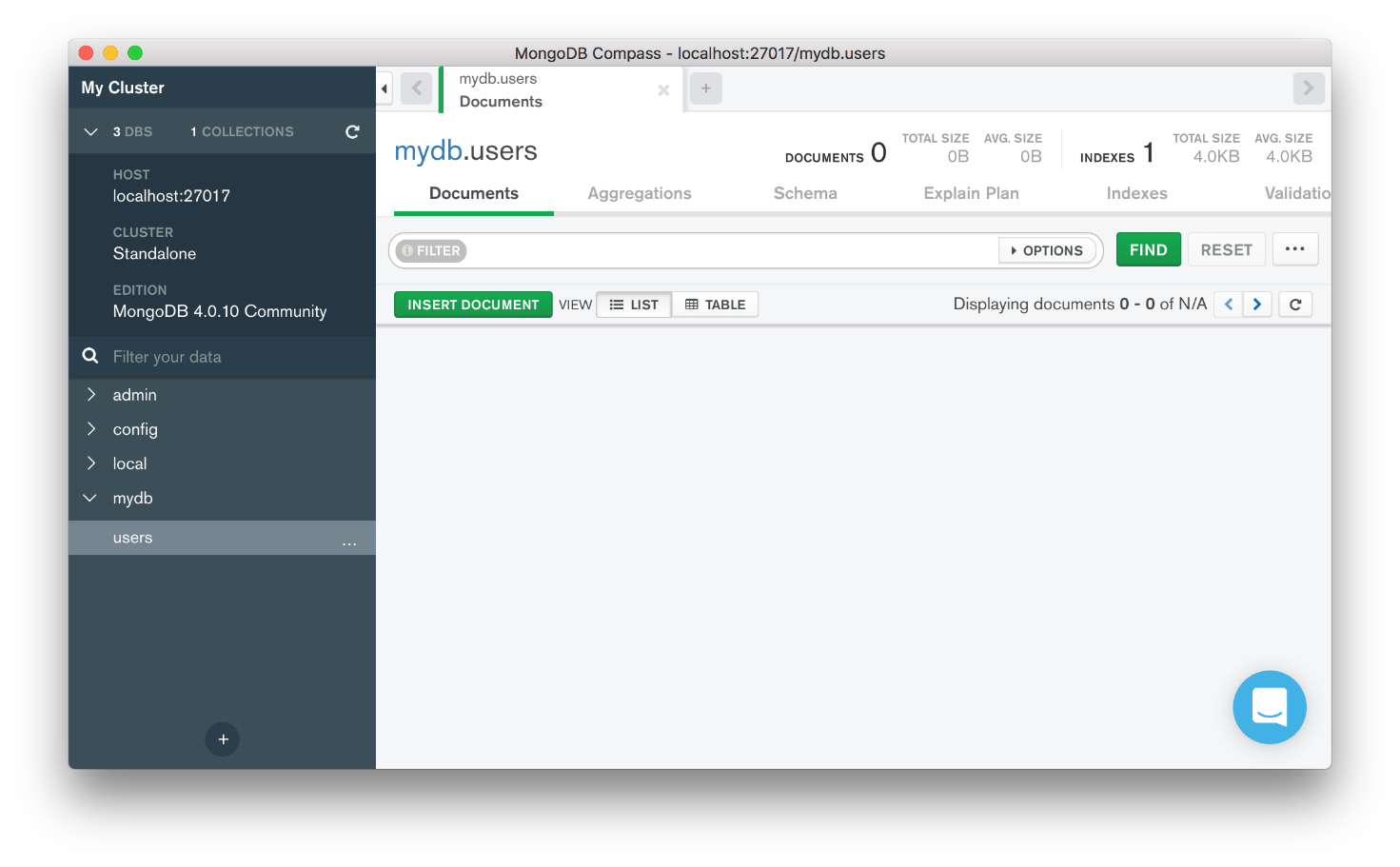


Введите имя новой БД (mydb) и первой коллекции в ней (users). Затем жмите Create Database:

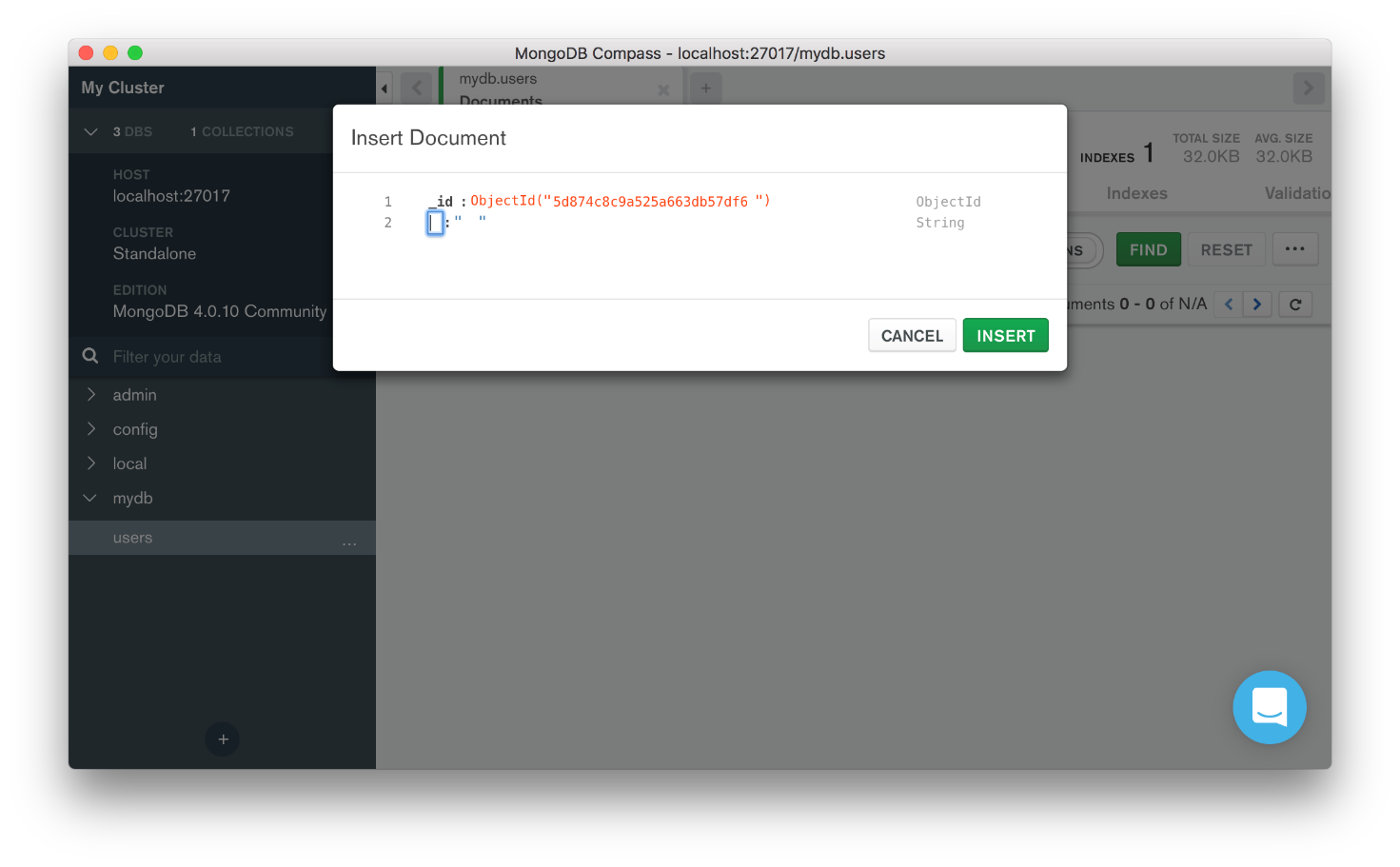


Имя базы нечувствительно к регистру: userdb и UserDB — одна и та же база. Чтобы не создавать конфликтующих имён, пользуйтесь только строчными буквами.

База данных состоит из коллекций, например, users, posts, comments. Ограничимся пока что одной — users. Нажмите Insert Document и создайте в ней документ:



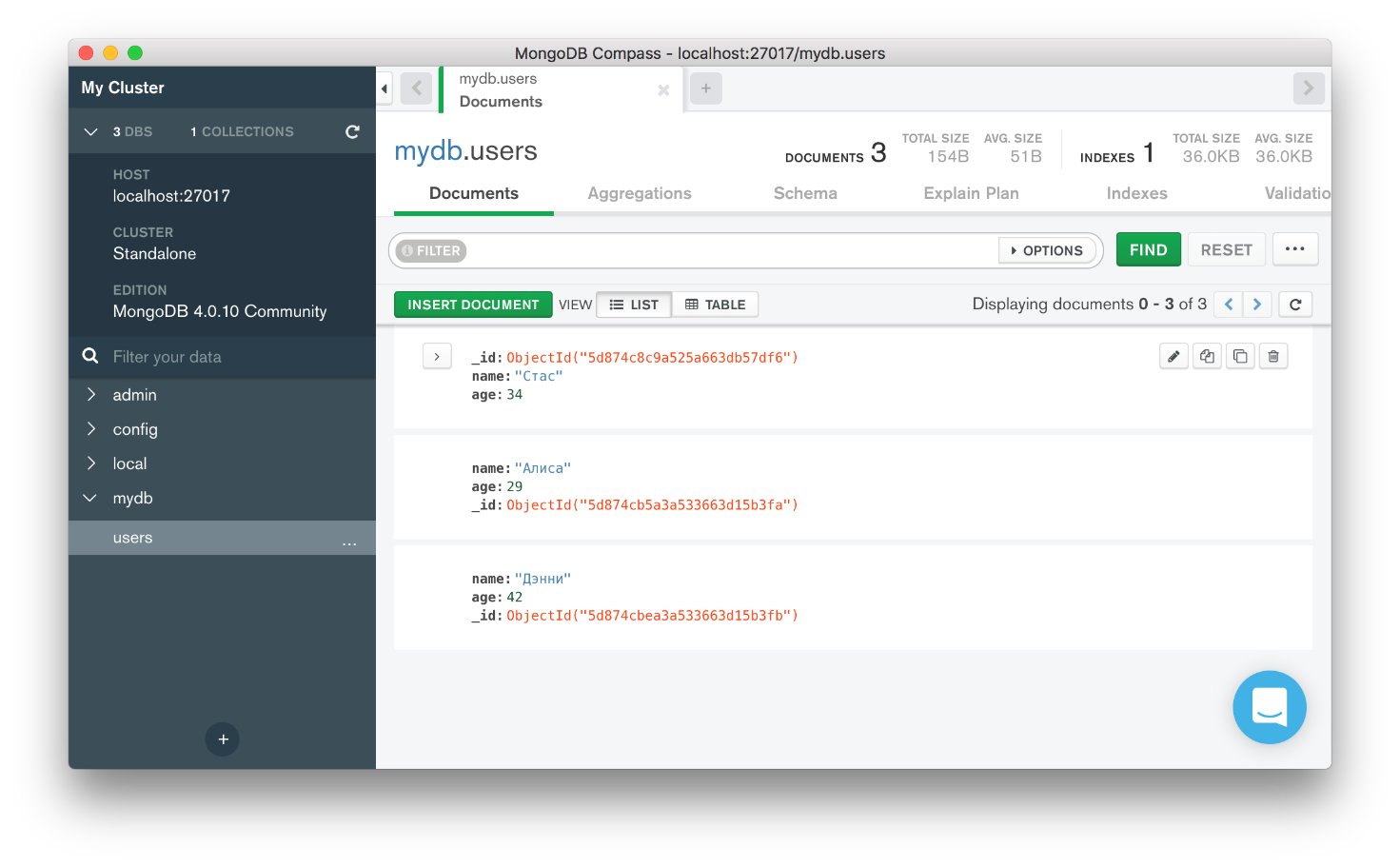
Откроется окно создания нового документа. У него уже есть одно поле — \_id:



Это уникальный идентификатор документа, MongoDB создаёт его сам. Добавьте ещё два поля: name и age. В правом столбце можно выбрать тип данных, записанных в поле. Для age выберите числовой тип — Int32:



Затем нажмите Insert — и документ будет создан. Создайте ещё несколько:



Документы в коллекции можно фильтровать. Это помогает искать их в базе. Просто напишите запрос в текстовом поле Filter. Запрос должен представлять собой объект:



Вот ещё несколько примеров запросов:

{} *// найдёт все документы коллекции*

{ name: 'Стас' } *// найдёт документы, где поле name равно "Стас"*

{ name: 'Стас', age: 34 } *// найдёт документы, где поле name равно "Стас" и поле age равно 34*

Искать документы через графический интерфейс очень удобно. Но добавлять их таким способом приходится нечасто — обычно этим занимается сервер. Для этого базу данных нужно связать с приложением на JavaScript. Этим мы и займёмся в следующем уроке.

## Дополнительные Ссылки

**Установщик Compass**

<https://www.mongodb.com/download-center/compass>

**Compass. Как писать запросы**

<https://docs.mongodb.com/manual/tutorial/query-documents/>

**Mongo Shell. Документация**

<https://docs.mongodb.com/manual/mongo/#working-with-the-mongo-shell>

**Подключение к mongo из JavaScript. Mongoose**

В предыдущем уроке мы рассказывали, как взаимодействовать с базой данных из графического интерфейса. Но документы в БД создаёт сервер, а не человек.

Нужно как-то «подружить» сервер и базу данных. Наш сервер запрограммирован на JavaScript, который не умеет работать с документами. Зато в JS есть другие структуры данных — объекты. Наша задача: научить JavaScript работать с документами как с объектами.

Чтобы подружить JS с документами, существуют специальные инструменты — ODM, или Object Document Mapper (англ. «сопоставитель объектов и документов»). У каждой БД есть свой сопоставитель. У MongoDB он называется Mongoose и представляет собой мост между двумя мирами: миром документов в базе данных и миром объектов JavaScript.

**Устанавливаем Mongoose**

Это делается через npm. Откройте в терминале папку с проектом и запустите команду:

npm i mongoose

Теперь его можно импортировать как модуль и пользоваться.

**Подключаемся к серверу mongo**

Эту задачу выполняет метод mongoose.connect:

*// app.js — входной файл*

const express = require('express');

const mongoose = require('mongoose');

const app = express();

*// подключаемся к серверу mongo*

mongoose.connect('mongodb://localhost:27017/mydb', {

useNewUrlParser: true,

useCreateIndex: true,

useFindAndModify: false

});

*// подключаем мидлвары, роуты и всё остальное...*

app.listen(3000);

Этот метод принимает на вход 2 параметра: адрес сервера базы данных и объект опций.

1. Адрес состоит из двух частей. Первая — mongodb://localhost:27017 — адрес сервера mongo по умолчанию. Он запускается на localhost на 27017 порту. Вторая часть — mydb — имя базы данных.
2. У объекта опций может быть много свойств. Нам важны три:
   * useNewUrlParser: true,
   * useCreateIndex: true,
   * useFindAndModify: false.

Мы не будем вдаваться в долгие объяснения, зачем нужны эти опции. Но если их не указать, у Mongoose в будущем возникнут проблемы совместимости с MongoDB. Дело в том, что логику взаимодействия MongoDB с Mongoose сейчас переписывают, и мы указываем эти опции, чтобы приложение вскоре не сломалось. Поэтому пропишите опции как у нас.

**Запускаем сервер MongoDB**

Сервер mongo запускают командой mongod:

mongod

Это нужно делать до запуска Node.js приложения. Иначе оно не сможет подключиться к базе данных и взаимодействовать с ней.

# Схемы и модели

Одна из особенностей нереляционных баз данных — отсутсвие схемы. Схема — набор требований к данным: сколько полей у записи, какой длины может быть значение каждого поля, какие символы в нём допустимы.

Схема накладывает ограничения на данные, которые записаны в базу. Иногда это удобно, иногда нет. Современные NoSQL базы позволяют использовать схемы выборочно.

В MongoDB нет поддержки схем по умолчанию, но мы можем добавить их через Mongoose. Это нужно, чтобы проверять, соответствует ли документ схеме, прежде чем записывать его в БД. Фактически, мы сможем валидировать данные перед записью.

### Создадим схему

В REST API есть ресурсы — в проекте Mesto это пользователи и карточки. Каждый из ресурсов должен соответствовать задуманной структуре: например, у пользователя должно быть имя и информация о себе. Зададим схему для пользователя через Mongoose:

*// models/user.js*

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

name: { *// у пользователя есть имя — опишем требования к имени в схеме:*

type: String, *// имя — это строка*

required: true, *// оно должно быть у каждого пользователя, так что имя — обязательное поле*

minlength: 2, *// минимальная длина имени — 2 символа*

maxlength: 30, *// а максимальная — 30 символов*

},

gender: {

type: String, *// гендер — это строка*

enum: ['м', 'ж', 'другой'] *// gender может принимать одно из трёх значений*

},

about: String, *// тип — String*

});

Такая схема говорит, что каждый пользователь в системе должен:

* обладать именем длиной от 2 до 30 символов;
* иметь гендер, который может принимать одно из трёх значений;
* иметь свойство about, причём строковое.

Иногда стандартных свойств схемы недостаточно. В Mongoose вы можете писать и более тонкие способы проверки данных. Для этого существует свойство validate. Это объект, свойствами которого являются:

* validator — функция валидации. Она должна возвращать булевое значение.
* message — сообщение об ошибке. Срабатывает в том случае, если функция валидации возвращает false.

Посмотрим на примере. В схему пользователя добавим свойство age, значение которого не должно быть меньше 18:

*// models/user.js*

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

name: { *// у пользователя есть имя — опишем требования к имени в схеме:*

type: String, *// имя — это строка*

required: true, *// оно должно быть у каждого пользователя, так что имя — обязательное поле*

minlength: 2, *// минимальная длина имени — 2 символа*

maxlength: 30, *// а максимальная — 30 символов*

},

gender: {

type: String, *// гендер — это строка*

enum: ['м', 'ж', 'другой'] *// gender может принимать одно из трёх значений*

},

age: { *// у пользователя есть возраст*

type: Number, *// возраст - число*

validate: { *// опишем свойство validate*

validator(v) { *// validator - функция проверки данных. v - значение свойства age*

return v >= 18; *// если возраст меньше 18, вернётся false*

},

message: 'Вам должно быть больше 18 лет!', *// когда validator вернёт false, будет использовано это сообщение*

}

},

about: String, *// тип — String*

});

Это простая проверка данных. Более сложную можно сделать с помощью регулярных выражений. Об этом расскажем позже.

Вернёмся к типам данных. При создании схем мы будем применять 5 основных типов данных:

String *// строка*

Number *// число*

Date *// дата*

Boolean *// логическое: true или false*

Array *// массив*

### Свойства-объекты

Если свойство документа должно быть объектом, придётся воспользоваться методом Schema дважды:

* первый вызов нужен для создания схемы объекта: в нём мы описываем, какую структуру должно иметь свойство;
* второй — для передачи описанной схемы в свойство, которое должно иметь эту структуру:

*// models/user.js*

const mongoose = require('mongoose');

*// создадим схему документа «Домашнее животное»*

const petSchema = new mongoose.Schema({

name: {

type: String,

required: true,

minlength: 2,

maxlength: 30,

},

age: Number

});

*// Когда схема готова, передадим её в свойство, которое должно соответствовать описанному шаблону:*

const userSchema = new mongoose.Schema({

...

pet: petSchema *// опишем свойство pet этой схемой*

});

### Свойства-массивы

Массивы нужны для хранения однотипных данных, поэтому описание схемы массива сводится к описанию шаблона элемента:

*// models/user.js*

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

...

hobbies: [{ *// описываем схему для одного элемента и заключаем её в квадратные скобки*

type: String,

minlength: 2,

maxlength: 30,

}]

});

Эта схема определяет свойство hobbies: оно должно содержать массив строк. Каждая — длиной от 2 до 30 символов.

### Создание модели на основе схемы

Схемой мы определили, каким должен быть документ в базе. Перейдём к следующему этапу — созданию самих документов. Для этого на основе схемы строится модель.

Модель — это «обёртка» из методов вокруг схемы. Благодаря ей мы можем читать, добавлять, удалять и обновлять документы. В Mongoose модель создают методом mongoose.model.

*// models/user.js*

const mongoose = require('mongoose');

*// Опишем схему:*

const userSchema = new mongoose.Schema({

name: {

type: String,

required: true,

minlength: 2,

maxlength: 30,

},

about: String,

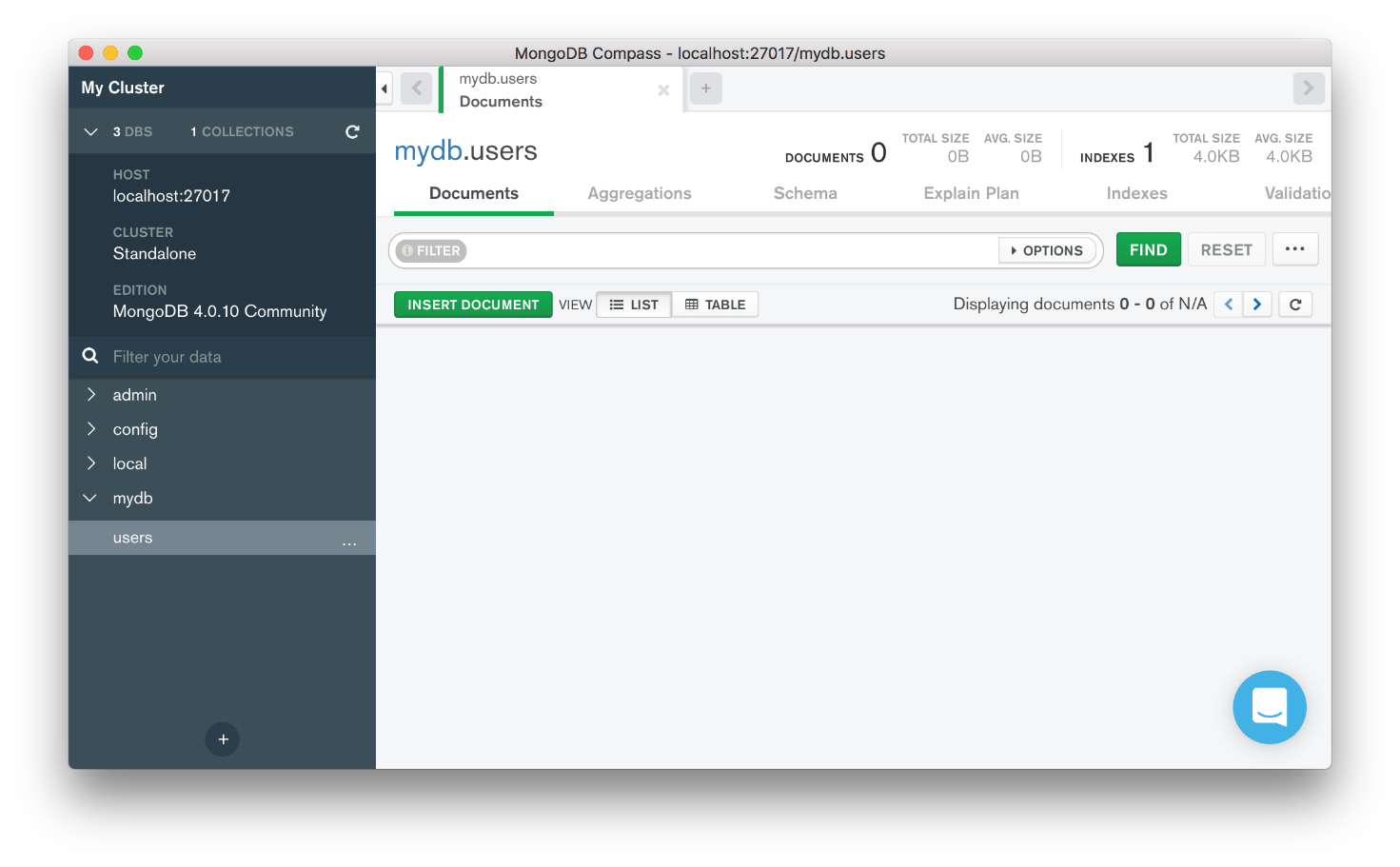
});

*// создаём модель и экспортируем её*

module.exports = mongoose.model('user', userSchema);

Мы передали методу mongoose.model два аргумента: имя модели и схему, которая описывает будущие документы.

Аккуратно — тут можно запутаться. Первый аргумент — имя модели — должно быть существительным в единственном числе. Но Compass отображает его во множественном. Дело в том, что Mongoose автоматически добавляет букву "s" в конце имени коллекции:



### Создание моделей: резюме

1. Определите ресурсы API.
2. Опишите схемы этих ресурсов.
3. Создайте модели на основе схем.

Когда модели готовы, можно использовать их для взаимодействия с базой: создавать, изменять и удалять документы в ней. Этим и займёмся в следующем уроке.

### Дополнительные ссылки

Типы данных в схемах Mongoose: <https://mongoosejs.com/docs/schematypes.html>

Больше про схемы: <https://mongoosejs.com/docs/guide.html>

# Создание, чтение, обновление и удаление документов

В предыдущем уроке мы создали модель. Она нужна, чтобы взаимодействовать с соответствующей коллекцией документов. Модель пользователя позволяет управлять коллекцией пользователей, модель карточек — коллекцией карточек.

С документом можно совершить одно из 4-х действий:

* создать (create),
* прочитать (read),
* обновить (update),
* удалить (delete).

Эти четыре действия обозначают аббревиатурой CRUD — по первым буквам английских названий. Разберём наиболее распространённые методы для каждого из этих действий.

### Создание документов — C

За это отвечает метод модели create. Он принимает на вход объект с данными, которые нужно записать в базу:

*// routes/users.js*

*/\**

*всякий код для создания роутеров и т.п.*

*\*/*

*// импортируем модель*

const User = require('../models/user');

router.post('/', (req, res) => {

const { name, about } = req.body; *// получим из объекта запроса имя и описание пользователя*

User.create({ name, about }); *// создадим документ на основе пришедших данных*

});

Метод create может быть промисом — ему можно добавить обработчики then и catch. Так обычно и делают, чтобы вернуть клиенту данные или ошибку:

*// routes/users.js*

*/\**

*всякий код для создания роутеров и т.п.*

*\*/*

const User = require('../models/user');

router.post('/', (req, res) => {

const { name, about } = req.body;

User.create({ name, about })

*// вернём записанные в базу данные*

.then(user => res.send({ data: user }))

*// данные не записались, вернём ошибку*

.catch(err => res.status(500).send({ message: 'Произошла ошибка' }));

});

### Чтение документов — R

Прочитать документ — значит найти его и получить данные. Искать можно разными способами. Три самых популярных: findById, findOne и find.

* Поиск конкретного документа. Метод findById ищет запись по идентификатору, то есть свойству \_id, которое есть у каждого документа в MongoDB. Поэтому чтобы найти конкретный документ, передайте методу findById идентификатор в строковом виде:

*// routes/users.js*

*/\**

*всякий код для создания роутеров и т.п.*

*\*/*

const User = require('../models/user');

router.get('/:id', (req, res) => {

User.findById(req.params.id)

.then(user => res.send({ data: user }))

.catch(err => res.status(500).send({ message: 'Произошла ошибка' }));

});

* Поиск одного документа по параметрам. Метод findOne возвращает первый документ, соответствующий запросу:

*// найти первое совпадение с полем name равным "Стас Басов"*

User.findOne({ name: 'Стас Басов' });

* Поиск всех документов по параметрам. Метод find работает так же, как findOne, но возвращает все документы по запросу:

*// найти всех тридцатилетних*

User.find({ age: 30 });

*// найти вообще всех*

User.find({});

### Обновление документов — U

Тут всё очень похоже на чтение. При обновлении мы находим запись и меняем её свойства.

* Обновление конкретной записи. Для этого есть метод findByIdAndUpdate, который принимает на вход идентификатор в строковом виде. Вторым параметром ему передают объект со свойствами, которые нужно обновить:

*// routes/users.js*

*// ...*

const User = require('../models/user');

router.patch('/:id', (req, res) => {

*// обновим имя найденного по \_id пользователя*

User.findByIdAndUpdate(req.params.id, { name: 'Виктор Гусев' })

.then(user => res.send({ data: user }))

.catch(err => res.status(500).send({ message: 'Произошла ошибка' }));

});

* Поиск первого совпадения с запросом и обновление его. Для этого есть метод findOneAndUpdate, который принимает на вход два объекта: первый с параметрами поиска, второй — с данными, которые нужно обновить:

*// найти первое совпадение с полем name равным "Стас Басов" и заменить имя на "Виктор Гусев"*

User.findOneAndUpdate({ name: 'Стас Басов' }, { name: 'Виктор Гусев' });

* Поиск совпадения с запросом и преобразование его. Аналогично можно превратить всех Стасов Басовых в Викторов Гусевых методом update:

*// найти все совпадения с полем name равным "Стас Басов" и заменить имя на "Виктор Гусев"*

User.update({ name: 'Стас Басов' }, { name: 'Виктор Гусев' });

Есть тонкость в работе методов обновления: по умолчанию параметр, который получает на вход обработчик then — это документ до обновления:

User.findByIdAndUpdate(req.params.id, { name: 'Виктор Гусев' })

*// user здесь — это документ до обновления*

.then(user => res.send({ data: user }));

Это можно исправить. Поэтому третьим аргументом методы обновления документов принимают объект опций. Нас интересуют три из них:

| **ОПЦИЯ** | **ЧТО ЗНАЧИТ** | **ЗНАЧЕНИЕ ПО УМОЛЧАНИЮ** |
| --- | --- | --- |
| new | передать обновлённый объект на вход обработчику then | false |
| runValidators | валидировать новые данные перед записью в базу | false |
| upsert | если документ не найден, создать его | false |

Благодаря объекту опций мы можем передать в then уже обновлённую запись. А также настроить валидацию и создать документ, если он не был найден:

User.findByIdAndUpdate(

req.params.id,

{ name: 'Виктор Гусев' },

*// Передадим объект опций:*

{

new: true, *// обработчик then получит на вход обновлённую запись*

runValidators: true, *// данные будут валидированы перед изменением*

upsert: true *// если пользователь не найден, он будет создан*

}

)

.then(user => res.send({ data: user }))

.catch(user => res.send({ "Данные не прошли валидацию. Либо произошло что-то совсем немыслимое" }));

### Удаление документов — D

* **Удаление конкретной записи.** Для этого есть метод findByIdAndRemove:

*// routes/users.js*

*// ...*

const User = require('../models/user');

router.delete('/:id', (req, res) => {

User.findByIdAndRemove(req.params.id)

.then(user => res.send({ data: user }))

.catch(err => res.status(500).send({ message: 'Произошла ошибка' }));

});

* **Удаление первого совпадения.** Этой цели служит findOneAndRemove:

*// удалим пользователя на основе имени*

User.findOneAndRemove({ name: 'Стас Басов' });

* **Удаление всех совпадений.** Для этого вызывайте deleteMany:

*// удалим всех тридцатилетних*

User.deleteMany({ age: 30 });

## Ссылки

В этом уроке мы перечислили лишь основные методы, но у mongoose моделей методов очень много. Все их можно найти в официальной документации:

<https://mongoosejs.com/docs/api/model.html>

**Как структурировать код. Контроллеры**

В предыдущем уроке мы связали роуты с базой данных. Когда на роут приходит запрос, мы обращаемся к базе и возвращаем клиенту ответ:

*// routes/users.js*

const User = require('../models/user');

router.post('/', (req, res) => {

const { name, about } = req.body;

*// записываем данные в базу*

User.create({ name, about })

*// возвращаем записанные в базу данные пользователю*

.then(user => res.send({ data: user }))

*// если данные не записались, вернём ошибку*

.catch(err => res.status(500).send({ message: 'Произошла ошибка' }));

});

Если роутов много, они теряются в коде. Для удобства работы разделим код на два файла: файл контроллеров и файл маршрутов. Вынесем обработчики в отдельную папку controllers*.*

*// controllers/users.js*

*// это файл контроллеров*

const User = require('../models/user');

module.exports.createUser = (req, res) => {

const { name, about } = req.body;

User.create({ name, about })

.then(user => res.send({ data: user }))

.catch(err => res.status(500).send({ message: 'Произошла ошибка' }));

};

Скопировать кодJAVASCRIPT

*// routes/users.js*

*// это файл маршрутов*

const { createUser } = require('../controllers/users');

router.post('/', createUser);

**Контроллер** — функция, ответственная за взаимодействие с моделью. То есть это функция, которая выполняет создание, чтение, обновление или удаление документа.

Контроллер в express также называют «последней мидлвэрой». Потому что внутри неё мы не вызываем next, а возвращаем ответ пользователю.

Файл контроллеров описывает логику обработки запросов. А файл маршрутов определяет, при каком запросе эту логику применять.

# Настраиваем связи

Раньше NoSQL означала полностью нереляционный подход к созданию баз данных. Со временем нереляционные базы данных стали вбирать в себя некоторые сильные стороны реляционных. И одна из таких сильных сторон — связи. В этом уроке разберёмся, как настроить связи между моделями в MongoDB.

## 1. Настраиваем связь двух схем

Построение связи начинается со схемы. Представим приложение, в котором есть две сущности: пользователи и объявления. Вот схема пользователя:

const userSchema = new mongoose.Schema({

name: { *// у пользователя есть только имя*

type: String,

minlength: 2,

maxlength: 20,

required: true,

},

});

module.exports = mongoose.model('user', userSchema);

Пользователи могут создавать объявления, которые описываются двумя полями: title и text — название объявления и его текст:

const adSchema = new mongoose.Schema({

title: {

type: String,

minlength: 2,

maxlength: 20,

required: true,

},

text: {

type: String,

minlength: 2,

required: true,

},

});

module.exports = mongoose.model('ad', adSchema);

Но нужно как-то определять, кто автор каждого объявления. Поэтому объявлению необходимо ещё одно поле — создатель, назовём его creator. В этом поле будем хранить ссылку на автора объявления.

Лучшая ссылка из одного документа на другой — идентификатор. Mongo автоматически создаёт поле \_id — уникальный идентификатор для каждого документа. Этот идентификатор позволяет связать один документ с другим.

Чтобы сделать это на уровне схемы, полю следует установить специальный тип — mongoose.Schema.Types.ObjectId и свойство ref. В это свойство записывают имя модели, на которую мы ссылаемся:

const adSchema = new mongoose.Schema({

title: {

type: String,

minlength: 2,

maxlength: 20,

required: true,

},

text: {

type: String,

minlength: 2,

required: true,

},

*// создаём поле creator*

creator: {

type: mongoose.Schema.Types.ObjectId,

ref: 'user',

required: true

},

});

## 2. При создании документа записываем \_id

В первом пункте мы создали модель, указав, что в поле creator должен быть идентификатор документа пользователя. Теперь этот идентификатор нужно записывать в поле creator при создании нового объявления:

*// controllers/ads.js*

const Ad = require('../models/ad');

module.exports.createAd = (req, res) => {

const { title, text, creatorId } = req.body;

Ad.create({ title, text, creator: creatorId })

.then(ad => res.send({ data: ad }));

};

## 3. Получаем всю информацию — метод populate

Мы связали две модели. Тем не менее схема объявления хранит только идентификатор пользователя. Чтобы получить всю информацию об авторе объявления, нужно вызвать метод populate, передав ему имя поля:

*// controllers/ads.js*

const Ad = require('../models/ad');

module.exports.getAds = (req, res) => {

Ad.find({})

.populate('creator')

.then(ad => res.send({ data: ad }));

};

Чтобы отправить в ответе несколько полей, полученных из связей, методу populate следует передавать массив:

*// controllers/ads.js*

const Ad = require('../models/ad');

module.exports.getAds = (req, res) => {

Ad.find({})

.populate(['creator', 'followers'])

.then(ad => res.send({ data: ad }));

};

## Ссылки

Более подробно про связи можно узнать в документации mongoose

<https://mongoosejs.com/docs/populate.html>