**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.

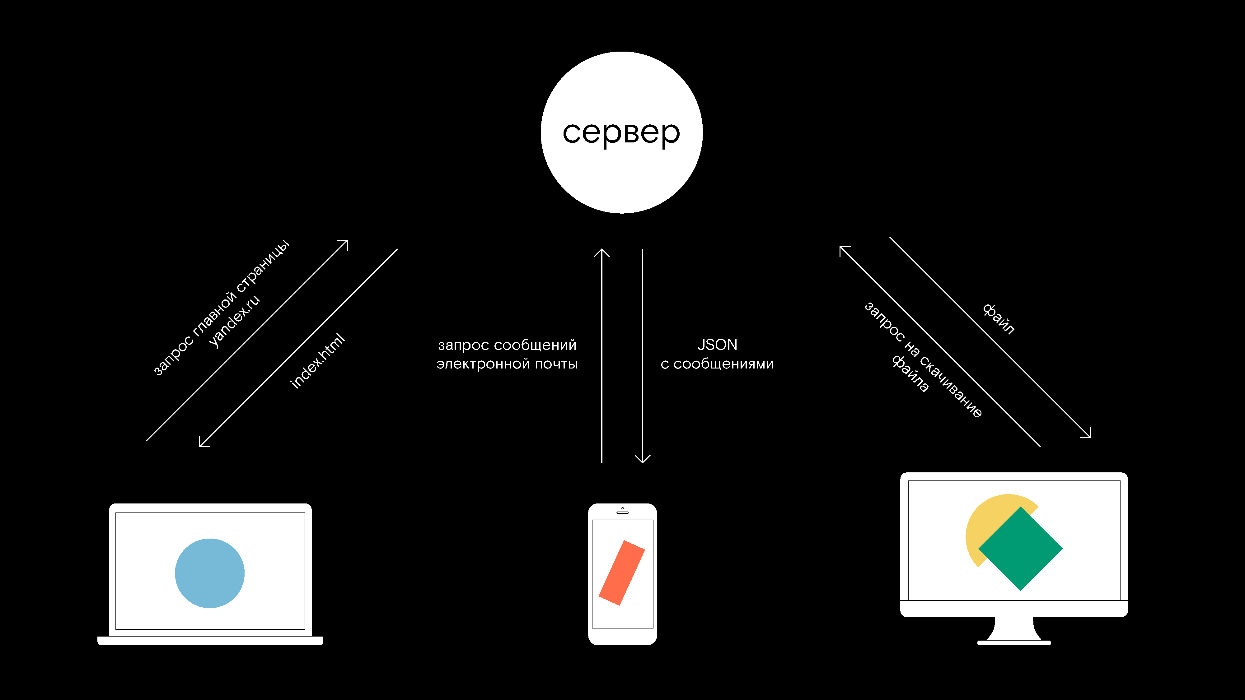
**Какие навыки получите:**

* 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:

Скопировать кодJSX

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.

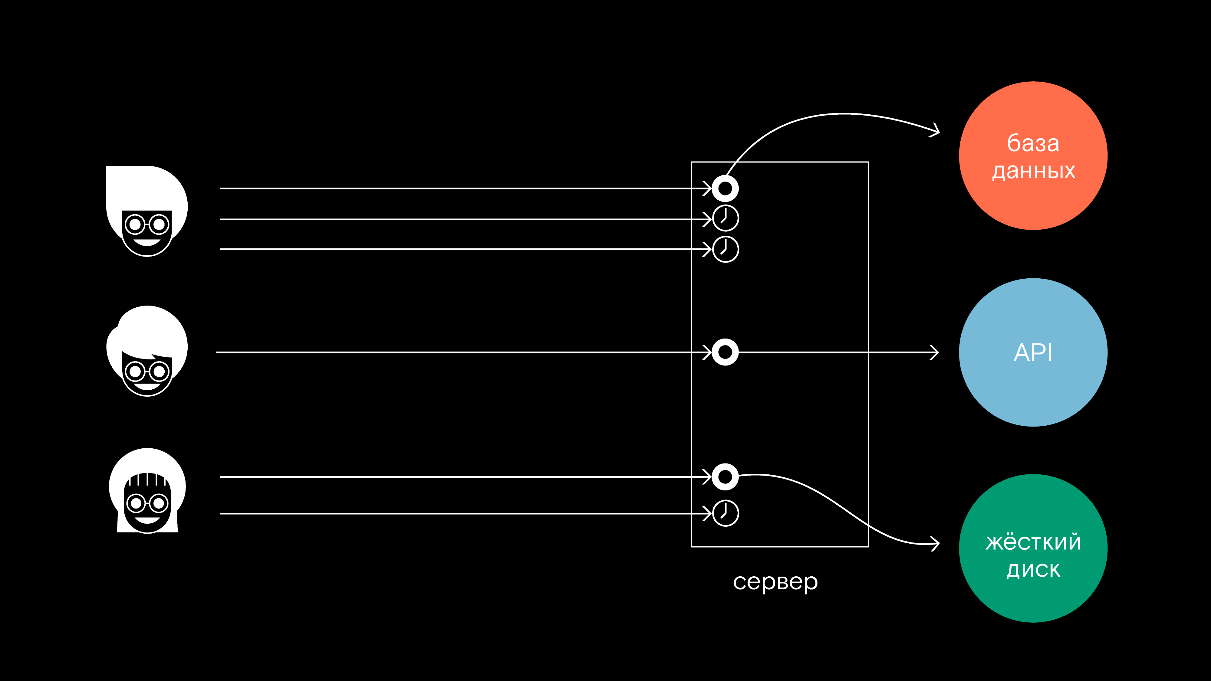
Приятный бонус: изучать новый язык не придётся, всё будет написано на уже знакомом JavaScript.

# 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.

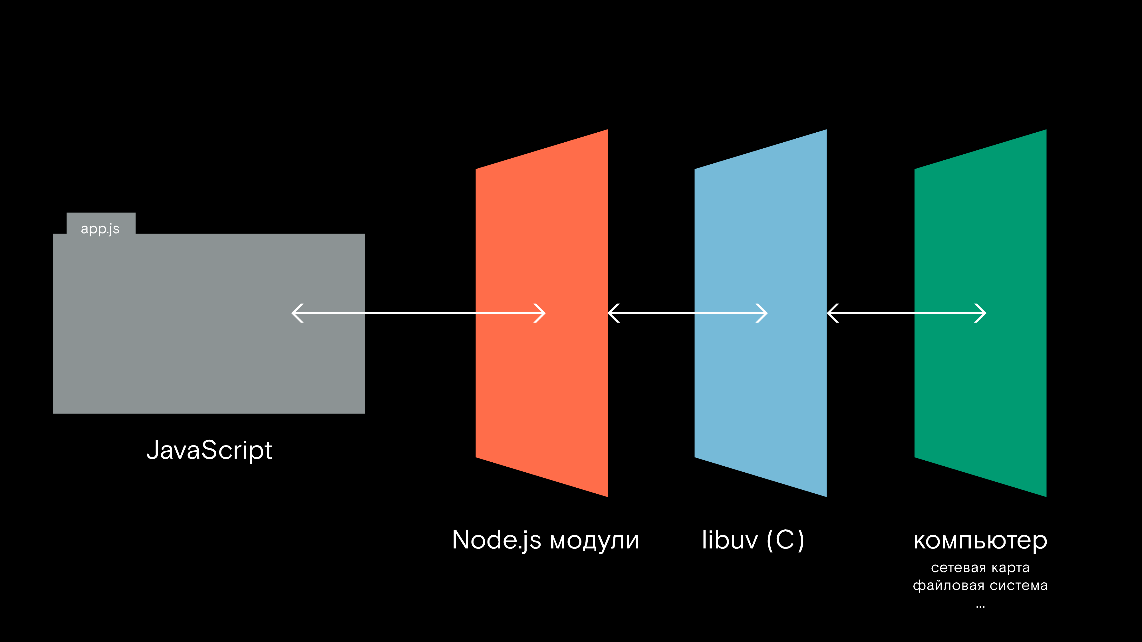
# Как работает Node.js?

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:

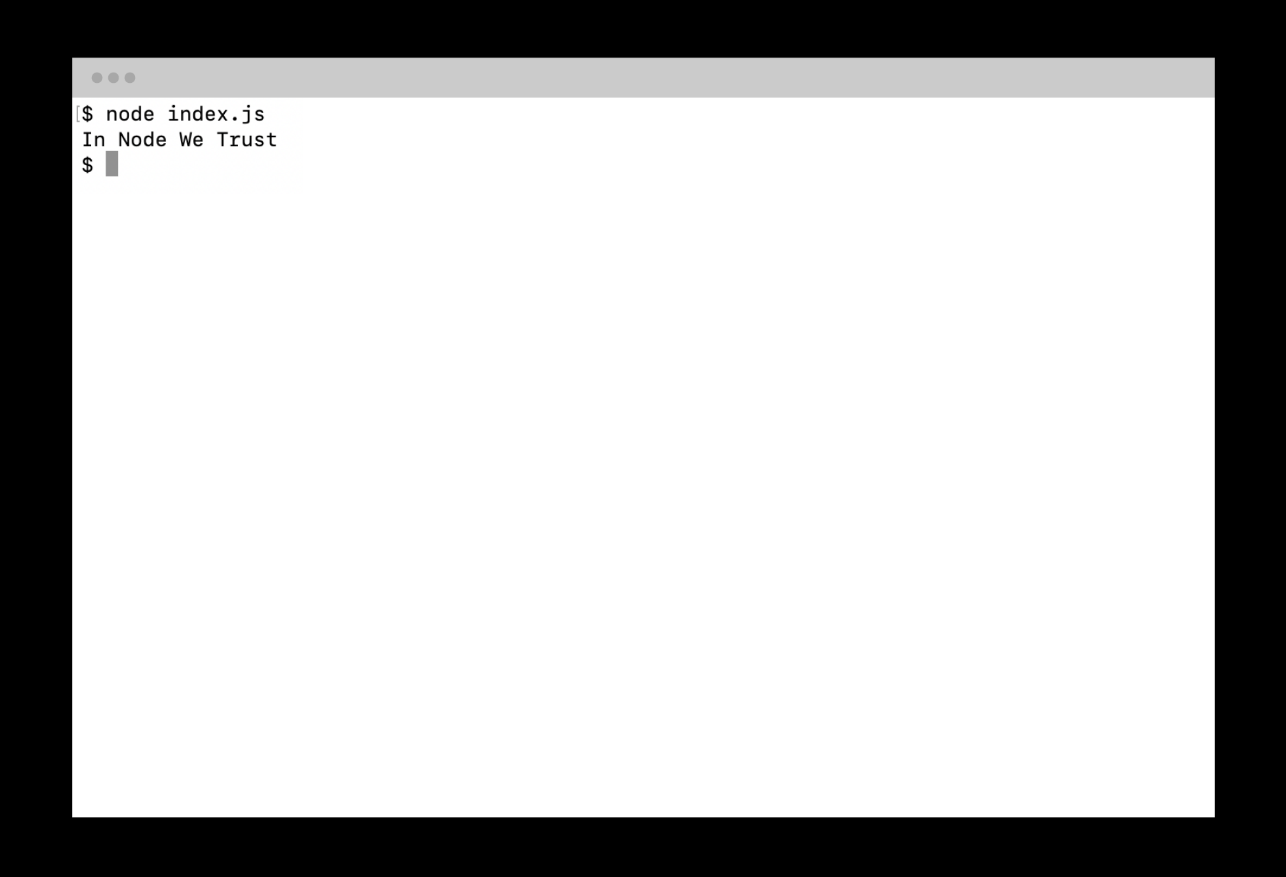
[Библиотека пакетов NPM](https://praktikum.yandex.ru/learn/web/courses/370a2c73-45bf-439f-a747-ef4e3c0db48f/sprints/1703/topics/af1c30a0-05e5-46b5-8570-6742270bc6d8/lessons/548a7bd4-8c6e-4846-8bdb-7ffe699ab718/)

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

Скопировать кодJAVASCRIPT

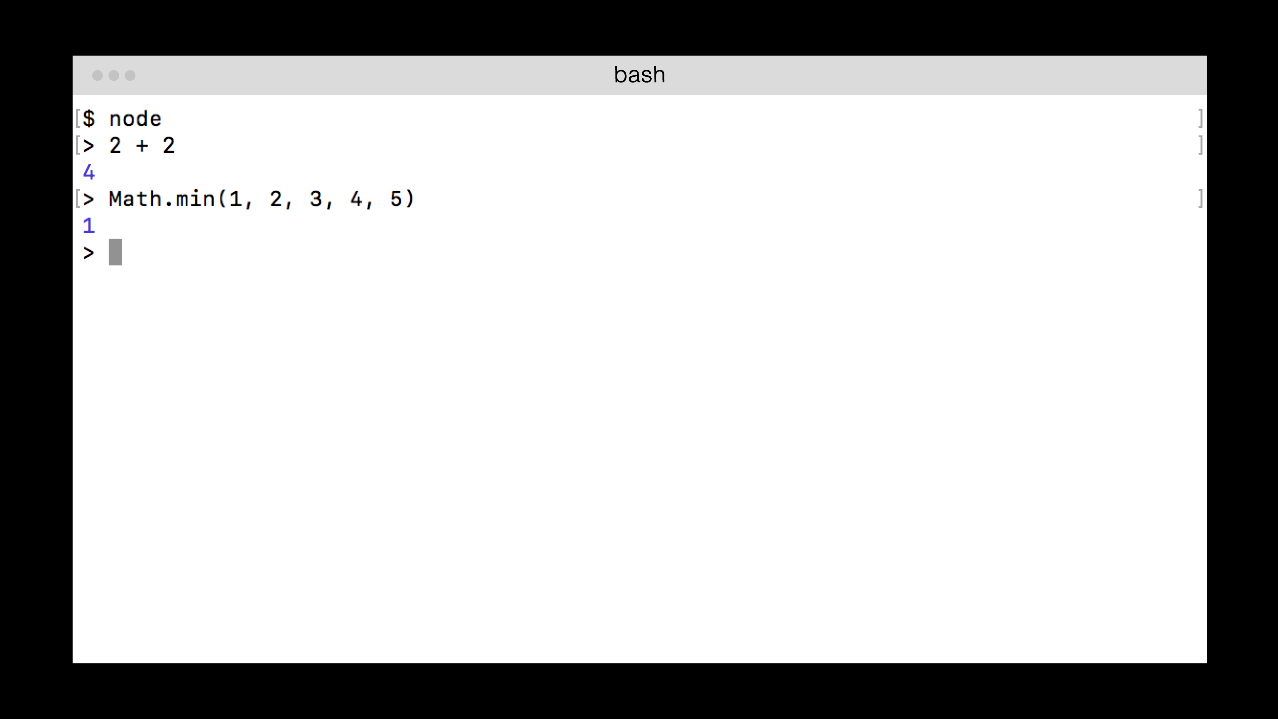
console.log('In Node We Trust');

Сохраните файл и запустите в терминале командой node index.js (для начала с помощью команды cd имя\_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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

const http = require('http');

# Создайте сервер

Для этого достаточно строчки кода:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

const http = require('http');

*// передадим обработчик*

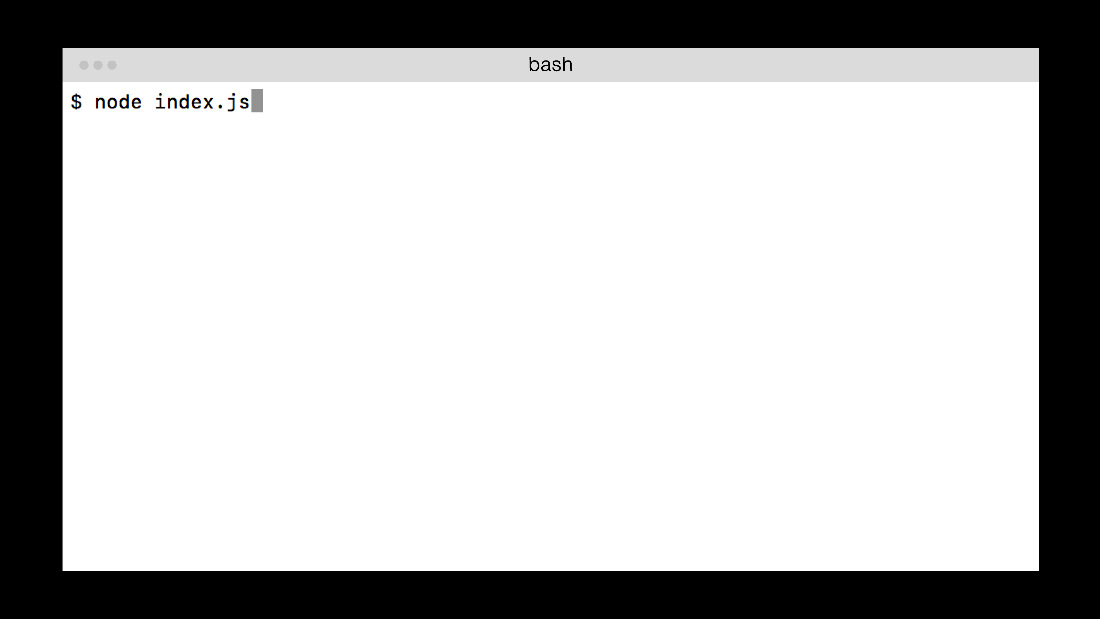
const server = http.createServer(() => {

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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

*// 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:

Скопировать кодJAVASCRIPT

*// 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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодBASH

NODE\_ENV=production node index.js

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

Внутри скриптов переменные окружения хранятся в объекте process.env:

Скопировать кодJAVASCRIPT

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:

Скопировать код

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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

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.

Скопировать кодJAVASCRIPT

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.

Пример

const http = require('http');

const { PORT = 3000 } = process.env;

const markup = `

<!DOCTYPE html>

<html>

<head>

<title>Я просто код</title>

<meta charset="UTF-8">

<style>

.container {

max-width: 500px;

display: flex;

flex-direction: column;

align-items: center;

margin: 0 auto;

padding-top: 100px;

}

p {

font-family: sans-serif;

font-size: 90px;

margin: 0;

}

img {

width: 512px;

margin-top: 50px;

}

</style>

</head>

<body>

<div class="container">

<p>Я бы обнял тебя — но&nbsp;— я просто код</p>

<img src="https://pictures.s3.yandex.net/code.gif" alt="мой код">

</div>

</body>

</html>

`;

// напишите код здесь

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

res.writeHead(200, {

'Content-Type': 'text/html'

});

// в методе end тоже можно передать данные

res.end(markup);

});

server.listen(PORT);

Без деструктиризации эту задачу можно решить условной конструкцией или оператором ИЛИ ||.

# 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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

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.

Скопировать кодJAVASCRIPT

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.

Пример

const http = require('http');

const { PORT = 3000 } = process.env;

const { BASE\_PATH } = process.env;

const mainPageMarkup = `

<!DOCTYPE html>

<html>

…………

</html>

`;

const submitSuccessMarkup = `

<!DOCTYPE html>

<html>

………………..

</html>

`;

const todos = [];

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

if (req.url === '/submit' && req.method === 'POST') {

let body = '';

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

body += chunk;

});

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

console.log(body); // например, здесь

// выделим всё, что идёт после знака равно, и добавим это в массив todos

todos.push(body.split('=')[1]);

console.log(todos);

res.writeHead(200, 'OK', {

'Content-Type': 'text/html'

});

res.end(submitSuccessMarkup);

});

}

if (req.url === '/' && req.method === 'GET') {

res.writeHead(200, {

'Content-Type': 'text/html'

});

res.end(mainPageMarkup);

}

});

server.listen(3000);

# Система модулей Node.js

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

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

*// 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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

*// 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:

Скопировать кодJAVASCRIPT

*// 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:

Скопировать кодJAVASCRIPT

*// index.js*

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

const someFunction = utils.someFunction;

const someValue = utils.someValue;

Проще сделать это деструктуризацией:

Скопировать кодJAVASCRIPT

*// 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.

Скопировать кодBASH

*# вот так ES6 модули работают*

node -experimental-modules index.mjs

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

Скопировать кодJAVASCRIPT

*// 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.

Скопировать кодJAVASCRIPT

*// 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.

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

*// app.js*

const fs = require('fs');

const readFile = () => {

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

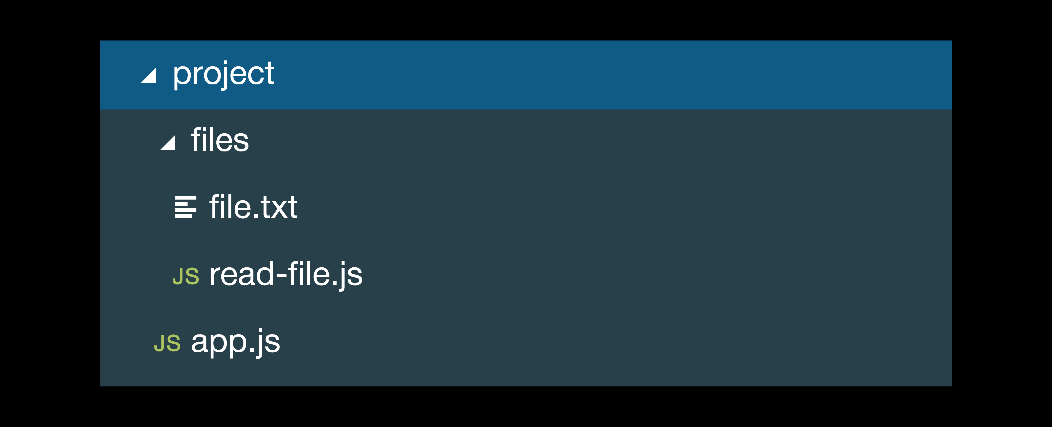
console.log(data); *// Выводим его содержимое в консоль*

}); *// 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:

Скопировать кодJAVASCRIPT

*// app.js*

const fs = require('fs');

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

readFile();

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

Скопировать кодJAVASCRIPT

*// 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.

Скопировать кодJAVASCRIPT

*// 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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

*// 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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

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.

Скопировать кодJAVASCRIPT

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.

Скопировать кодJAVASCRIPT

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.

Скопировать кодJAVASCRIPT

const fs = require('fs');

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

if (err) {

console.log(err);

return;

}

console.log('Файл удалён!');

});

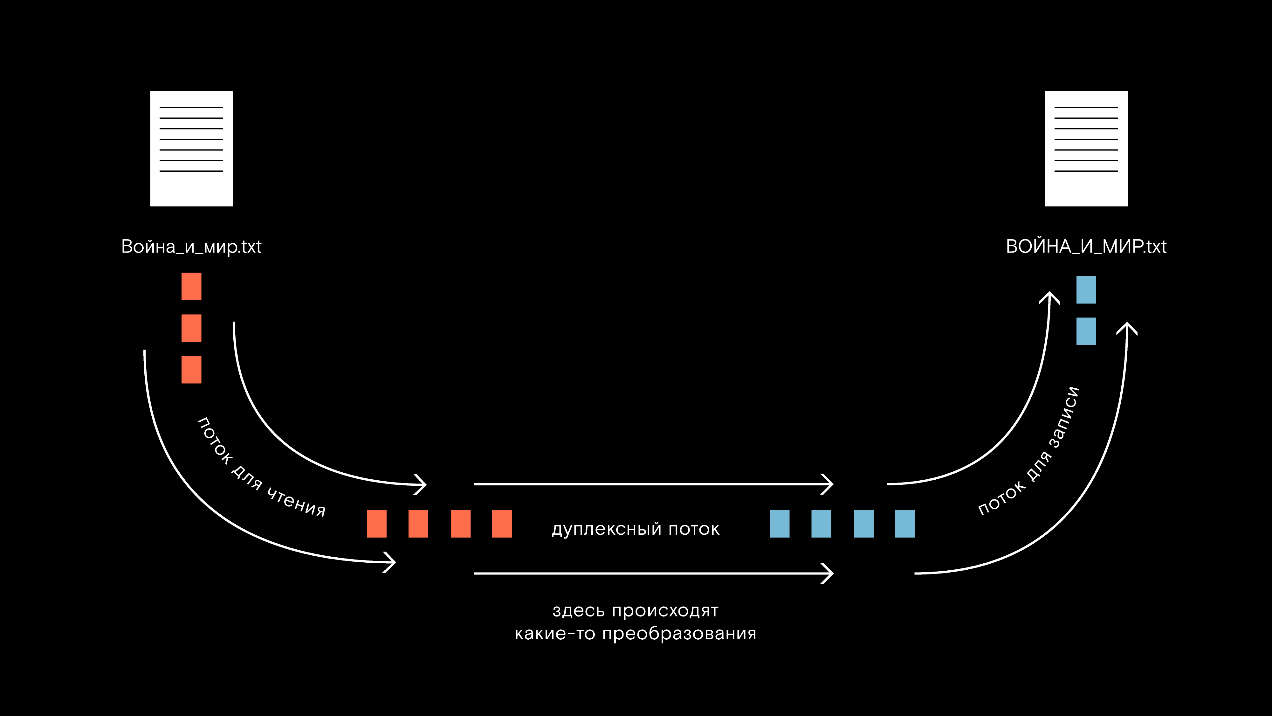
# Много чего ещё

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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

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:

Скопировать кодJAVASCRIPT

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.

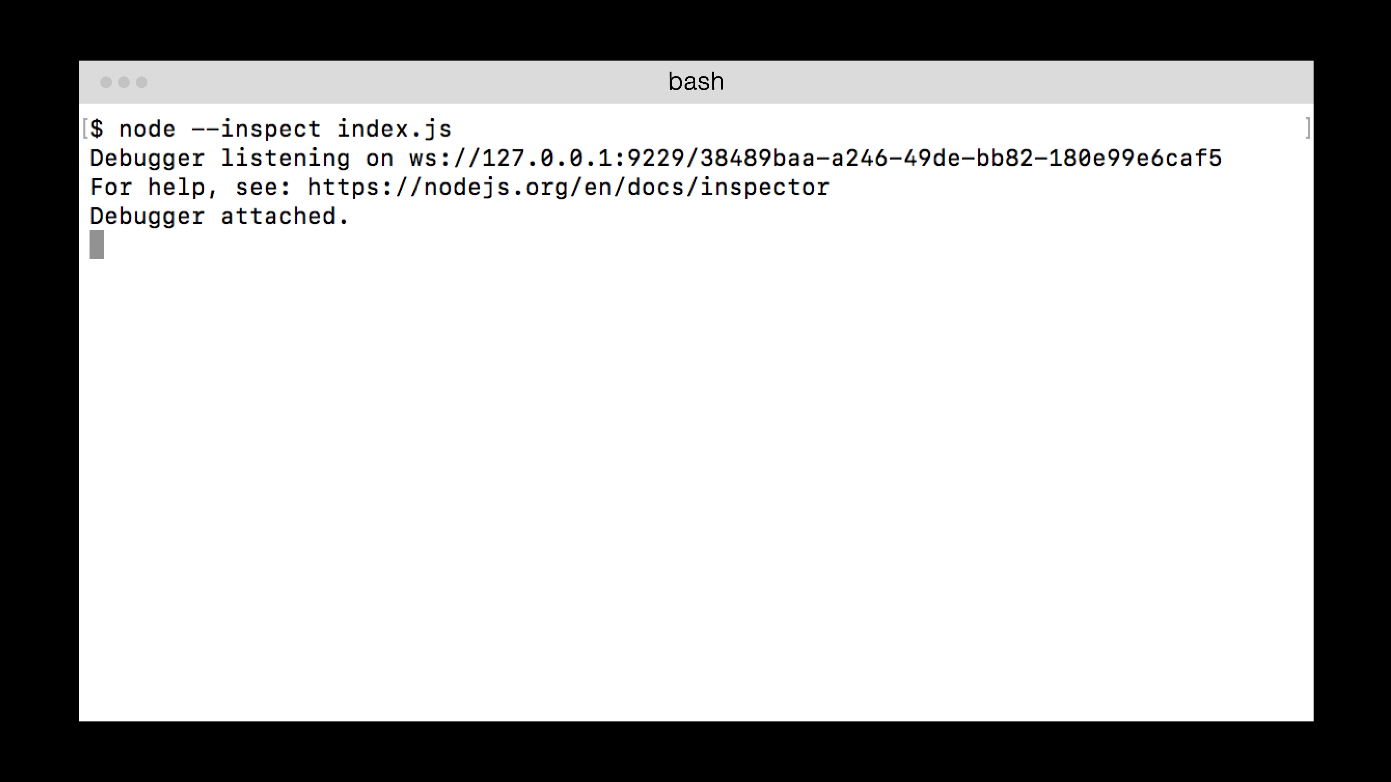
# Отладка в браузере

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

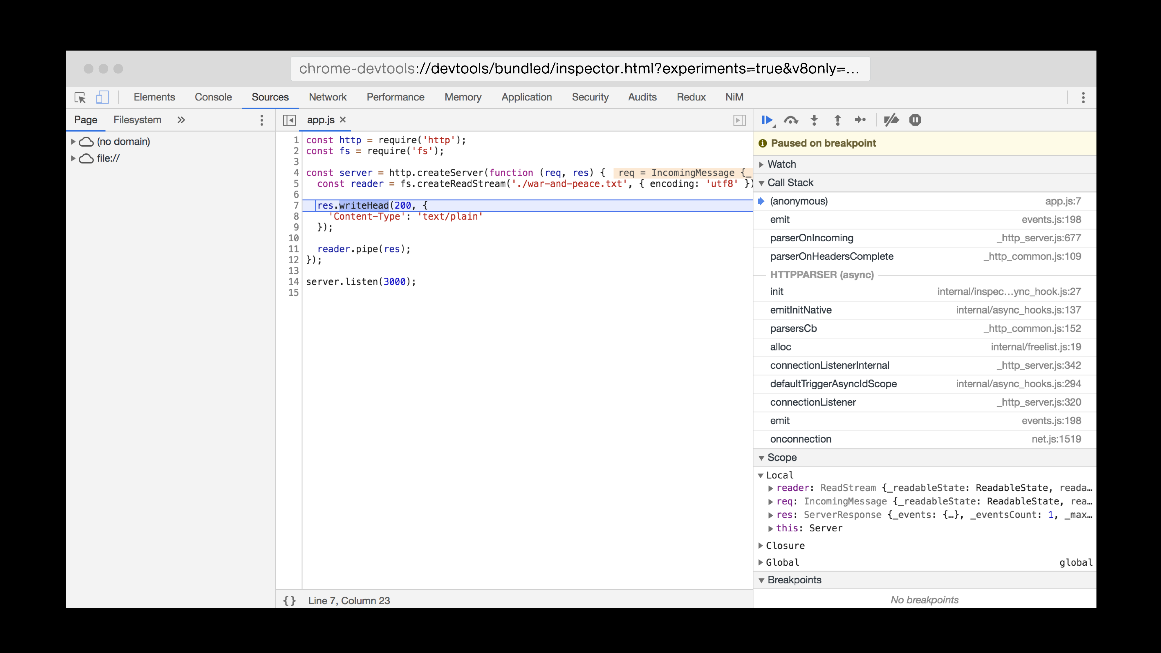
1. установите в Яндекс.Браузер или в Google Chrome расширение [Node.js Inspector Manager](https://chrome.google.com/webstore/detail/nodejs-v8-inspector-manag/gnhhdgbaldcilmgcpfddgdbkhjohddkj);
2. запустите приложение с флагом --inspect:

Скопировать кодBASH

node --inspect index.js

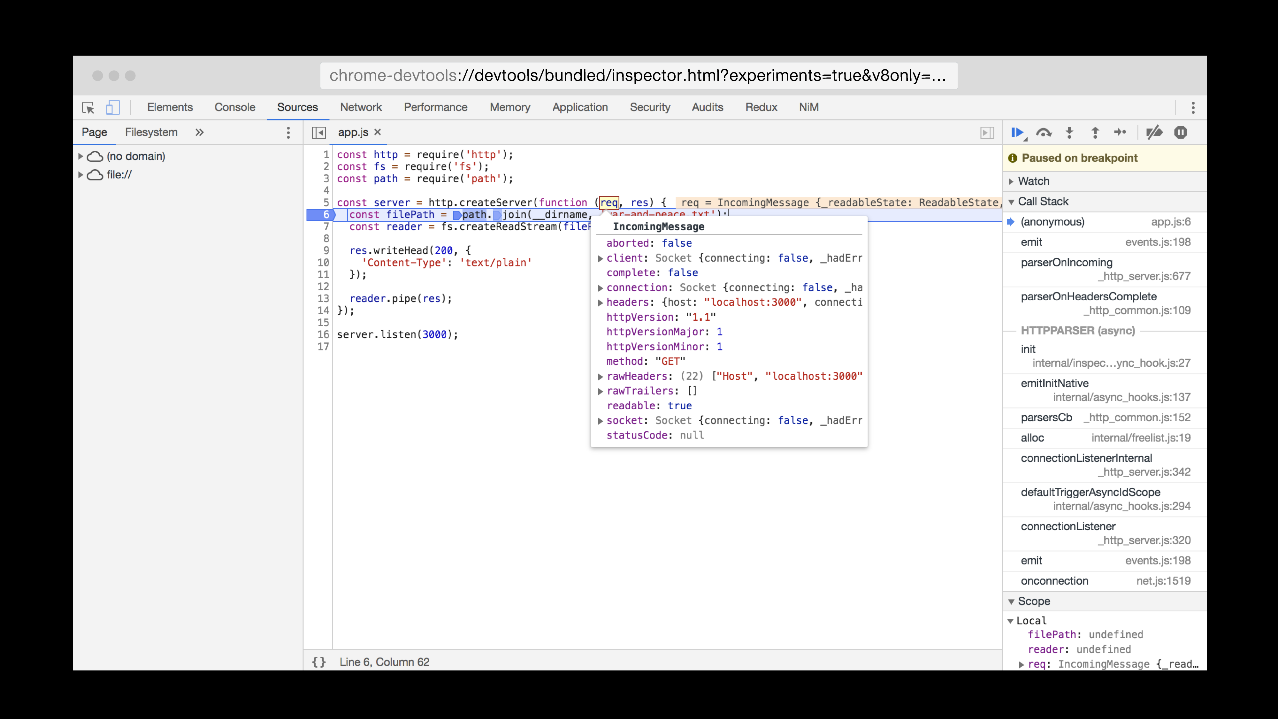


После этой команды в браузере откроется вкладка с инструментами разработчика и вашим кодом:



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/)

# Компьютерные сети

Бэкенд-разработчику важно понимать, как устроены компьютерные сети.

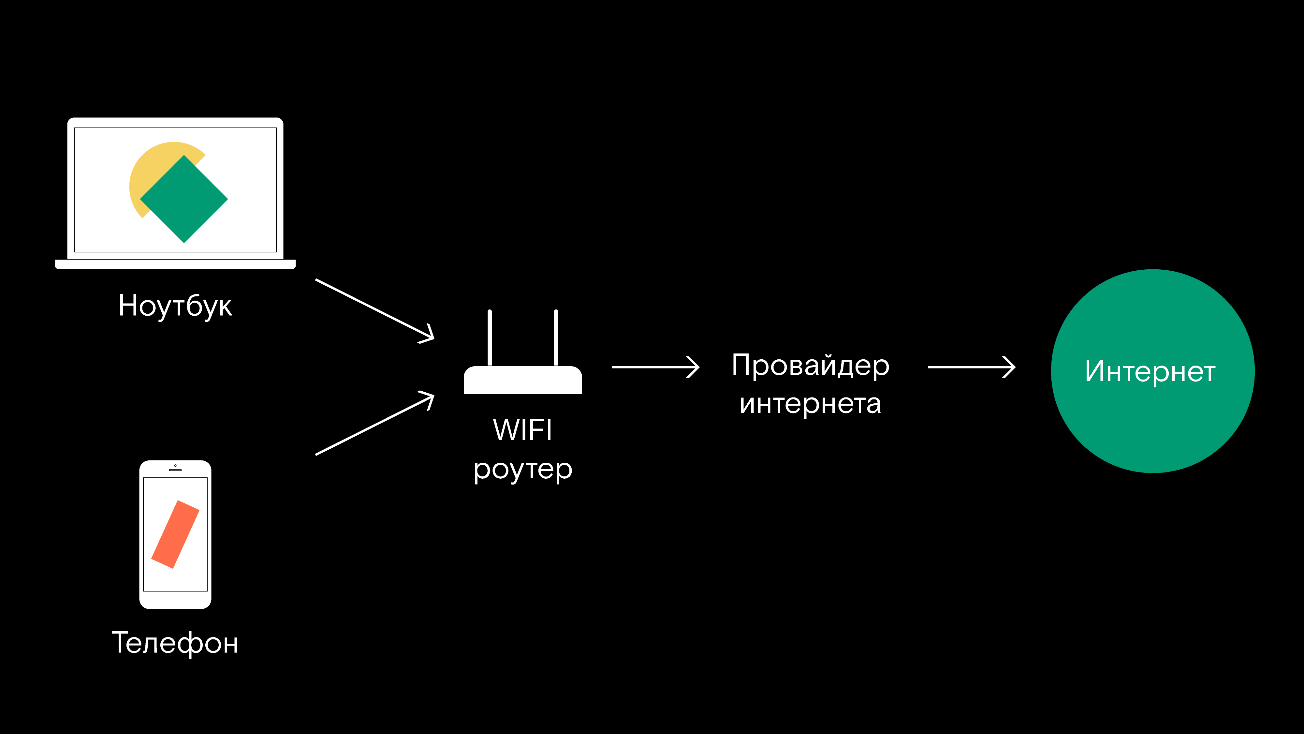
Наступит момент, когда вам потребуется защищать свой сервер от атак, настраивать шифрование и соединяться с серверами, используя различные протоколы — и всё это непременно с непроницаемым выражением лица, как у Джонни Ли Миллера в «Хакерах». Для этих задач (и чтобы добиться такой же мимики) надо понимать,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



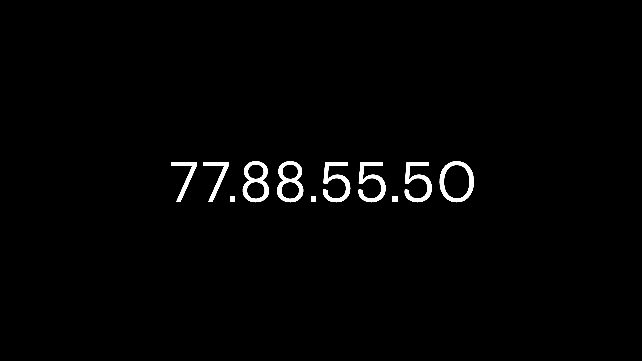
В вашей квартире наверняка есть роутер. Он обеспечивает телефону и компьютеру доступ в интернет, передавая им данные по кабелю или радиоволнам (если это Wi-Fi роутер). Роутер, в свою очередь, подключен к интернет-провайдеру, с помощью которого можно войти в глобальную сеть интернет.

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.

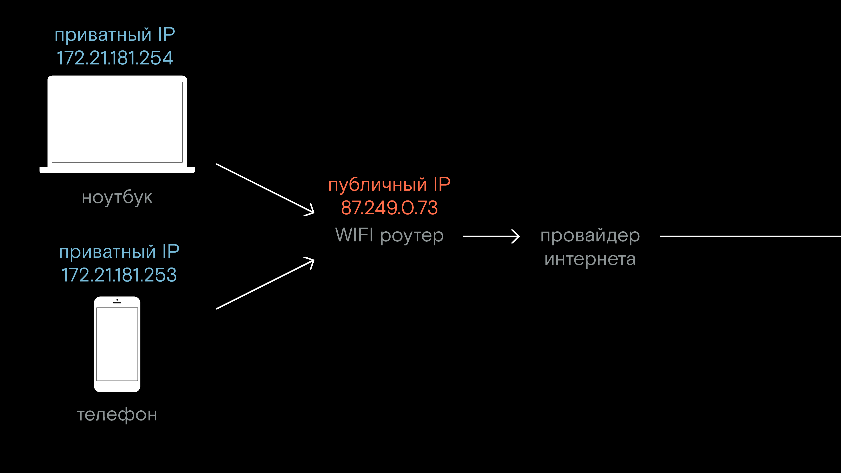


## Как найти друг друга в сети? IP-адрес

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, которые 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:

Скопировать кодBASH

*# 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.