ИТМО

**И.Б. Государев**

# ОСНОВЫ РАЗРАБОТКИ ВЕБ-ПРИЛОЖЕНИЙ НА ПЛАТФОРМАХ NODE.JS И DENO

**Санкт-Петербург**
**2023**

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

**И.Б. Государев**

# ОСНОВЫ РАЗРАБОТКИ ВЕБ-ПРИЛОЖЕНИЙ НА ПЛАТФОРМАХ NODE.JS И DENO

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

РЕКОМЕНДОВАНО К ИСПОЛЬЗОВАНИЮ В УНИВЕРСИТЕТЕ ИТМО
по направлению подготовки 09.04.04 Программная инженерия
в качестве Учебно-методическое пособие для реализации основных
профессиональных образовательных программ высшего образования
магистратуры

**Санкт-Петербург**
**2023**

Государев И.Б., Основы разработки веб-приложений на платформах Node.js и Deno– СПб: Университет ИТМО, 2023. – 177 с.

Рецензент(ы):
Жуков Николай Николаевич, кандидат физико-математических наук, доцент (квалификационная категория "ординарный доцент") факультета программной инженерии и компьютерной техники, Университета ИТМО.

Данное учебно-методическое пособие развивает идеи первой книги автора «Введение в веб-разработку на языке JavaScript» и онлайн-курса «Серверные веб-технологии и системы управления контентом» на платформе openedu.ru. В пособии рассматриваются фундаментальные основы и прикладные аспекты использования языка JavaScript (и TypeScript) для разработки веб-ресурсов. Рассматриваются основные аспекты разработки бэкэнда: шаблонизация, маршрутизация, скаффолдинг, отладка, развёртывание, контейнеризация. Обучающимся предложены задания развивающего и проблемного типа, нацеленные на формирование профессиональных компетенций в области веб-разработки. Все примеры и задания доступны в интерактивной части пособия (сайт kodaktor.ru), которая является зарегистрированным в Роспатенте средством электронного обучения и содержит не только образцы кода, но и скринкасты по ряду рассматриваемых в текстовой части вопросов. Структура пособия позволяет организовать изучение материала в заочной и дистанционной форме.

**Университет ИТМО** – ведущий вуз России в области информационных и фотонных технологий, один из немногих российских вузов, получивших в 2009 году статус национального исследовательского университета. С 2013 года Университет ИТМО – участник программы повышения конкурентоспособности российских университетов среди ведущих мировых научно-образовательных центров, известной как проект «5 в 100». Цель Университета ИТМО – становление исследовательского университета мирового уровня, предпринимательского по типу, ориентированного на интернационализацию всех направлений деятельности.

© Университет ИТМО, 2023
© Государев И.Б., 2023

Содержание

**Глава 1. Рекомендации по изучению материалов пособия и основные термины ..... 5**
Методические рекомендации преподавателям и студентам ..... 5
Подготовка среды для работы с JavaScript ..... 6
Понятия фронтэнда и бэкэнда ..... 13
Протоколы, используемые в вебе при создании веб-приложений ..... 21

**Глава 2. Использование Node.js для отправки сетевых запросов и их обслуживания ..... 30**
Метод net.connect ..... 30
Метод net.Server ..... 32

**Глава 3. Fetch API ..... 36**
Основа: AJAX ..... 37
О потоках ..... 39
Отправка запросов ..... 43

**Глава 4. Модель приложения на Node.js ..... 45**
Отладка ..... 54
Простейшая маршрутизация ..... 63
Блокирующий код ..... 64
Выдача статики ..... 68
Приём запроса с телом сообщения ..... 71
Модуль connect ..... 77
Деплой приложения на Next.js/vercel ..... 77
Использование ngrok ..... 83

**Глава 5. Библиотека Express ..... 86**
Установка Express ..... 87
Понятие middleware. Виды middleware и их стек ..... 93
Маршрутизация. Создание express.Router и дерева маршрутов ..... 99
Шаблонизация ..... 107
Приём запроса с телом ..... 112
Запуск приложения в формате сервиса ..... 116
Использование ядер и потоков ..... 117

**Глава 6. Нано и микросервисы, REST API ..... 123**
Создание декларативного сервиса на основе OpenAPI ..... 124
Критерии RESTful ..... 126
Операции с ресурсами на основе HTTP-глаголов ..... 128

Реализация Express-приложения в формате микросервиса ..... 129
Скаффолдинг ..... 139
Создание микросервиса на основе проекта Hydra ..... 140

**Глава 7. HTTPS и HTTP/2 ..... 145**
Локальный сертификат ..... 145
Модуль node:https ..... 148
Контейнер companion ..... 148
Модуль node:http2 ..... 151

**Глава 8. Веб-сокеты ..... 153**
Клиентская реализация WebSocket ..... 153
socket.io ..... 158

**Глава 9. Использование Deno и Bun ..... 160**
Deno ..... 160
Bun ..... 170

**Заключение ..... 171**
Вопросы для самопроверки ..... 172
Примеры заданий автоматизированной проверки ..... 174

**Литература ..... 176**

## Глава 1. Рекомендации по изучению материалов пособия и основные термины

### Методические рекомендации преподавателям и студентам

Это пособие было подготовлено по следам ранее созданного онлайн-курса «Серверные веб-технологии и системы управления контентом». За период времени, прошедший после запуска курса, ситуация в экосистеме Node.js претерпела изменения, и автор решил постараться отразить некоторые из них в текстовой форме. Курс находится на площадке «Открытое образование» по адресу https://openedu.ru/course/ITMOUniversity/NODEJS и содержит в основном видеоуроки в формате скринкастов. Курс и данное пособие образуют единый комплекс, в силу логики выстраивания материала ориентированный главным образом на последовательную самостоятельную работу студентов, и от преподавателя не требуется специальных мер по организации учебной деятельности, нужно лишь обеспечить сопровождение решения упражнений и выполнения контрольных мероприятий.

Пререквизитами являются знание основ JavaScript и сетевых технологий, общий уровень компетенций в области Linux. В книге Д. Флэнагана [1] дано наиболее полное описание JavaScript в формате подробного справочника, а предыдущей работой автора данного пособия является [2] с попыткой сфокусировать внимание на интересных аспектах JavaScript с точки зрения исследования языка (в [2] обсуждается асинхронность JavaScript и промисы).

Первые пять глав пособия соответствуют первым пяти неделям курса. Рекомендуется сначала параллельно изучать материалы первых трёх глав/недель. Затем целесообразно посмотреть материалы 4-й недели и прочитать 4-ю главу, далее посмотреть материалы 5-й недели и прочитать 5-ю главу. Далее работа с курсом и пособием может осуществляться параллельно.

В пособии представлены 20 упражнений для самостоятельной работы. Вероятно, оптимально будет организовать встречи для кросс-проверки результатов (peer-to-peer), на которых обучающиеся могут обменяться своими достижениями. Их также можно рассматривать как материалы лабораторных работ в случае очной организации обучения.

Автор предлагает считать задания с автоматизированной проверкой (примеры приведены перед списком литературы) основой для организации аттестации. Курс будет считаться пройденным при условии успешного выполнения этих заданий и итогового теста (примеры вопросов которого также приведены в заключительной части пособия).

Также преподавателям и студентам рекомендуется в начале изучения материалов совместно составить список тем докладов по темам, расширяющим и дополняющим курс (например: «Синхронизация состояния веб-приложения на клиенте и сервере» или «Серверный рендеринг в современных фреймворках») и делиться друг с другом индивидуальным опытом освоения таких технологий и инструментов.

Первые шаги связаны с терминологией и настройкой среды для работы.

## Подготовка среды для работы с JavaScript

По состоянию дел на период написания этого пособия язык JavaScript, как и в течение многих предыдущих лет, является основным языком разработки клиентских веб-приложений. С другой стороны, есть основания ожидать роста популярности WebAssembly и расширения соответствующей экосистемы, которая позволит использовать в рамках браузеров принципиально другие языки, такие как C. А с третьей стороны в рамках Web3 приложения могут выполняться в Ethereum-блокчейне и быть написанными на языке типа Solidity. Но пока мы исходим из того, что при наличии желания понимать и контролировать практически весь процесс разработки веб-приложения с его клиентской и серверной частью JavaScript является естественным выбором. Разработчик может писать код на TypeScript, Elm, ReasonML и т.д., но результатом так или иначе будут JavaScript-сценарии. Поэтому довольно логично, что JavaScript становится универсальным языком решения практически всех задач веб-разработки, включая автоматизацию и разработку бэкэнда (или, во всяком случае, слоя бэкэнда-для-фронтэнда).

Язык JavaScript развивается достаточно динамично, в конвергенции с другими языками, вбирая в себя некоторые их характеристики. Ежегодно нововведения фиксируются в очередной версии стандарта (например, ECMAScript-2023). Часто они заранее «обкатываются» на свежих версиях платформ, поэтому важно следить за этими обновлениями.

Целью этого пособия не является доказательство или обоснование преимуществ использования JavaScript для разработки на внебраузерной платформе. Но некоторые очевидные вещи всё же отметим. Если много лет назад допустимо было создавать программы на JavaScript прямо в браузере, напрямую записывая инструкции в элемент script, то современная разработка для браузера ведётся в специальном окружении, которое включает не только редактор кода или IDE, но и линтеры, средства для работы с каскадными стилями, а также библиотеку типа React с сопутствующими инструментами. Все эти составляющие JavaScript-экосистемы подключаются к проекту в виде прямых или девелоперских зависимостей через менеджер зависимостей, в качестве которого в большинстве случаев используется npm, являющийся частью экосистемы Node.js.

Когда мы говорим о разработке на JavaScript не для браузера, то скорее всего имеем в виду разработку на платформе Node.js, которая нацелена:

- на создание сценариев, работающих не в браузере или не только в браузере (например, для вычислений или для автоматизации работы в окружении);
- на создание серверных веб-приложений.

Сначала кратко обсудим первый аспект.

Node.js и Deno – это программное обеспечение, относящееся к классу сред исполнения (кратко – runtime). Runtime не просто присутствует в памяти во время выполнения программы, а предоставляет виртуальную машину, которая и осуществляет выполнение. В основе Node.js и Deno находится движок – виртуальная машина V8, разрабатываемая в Google. На том же движке работает и браузер Chrome. При этом Deno содержит ещё и транспилятор TypeScript. В 2022 г. на профильных ресурсах обсуждали также новый инструмент под названием bun.sh – к этому мы вернёмся в Главе 9. Сложно сказать, что будет с этим инструментом дальше – но он интересен как феномен развития, поэтому и фиксируется в этом пособии.

Node используется в двух основных «режимах»:

- выполнение в стиле интерфейса командной строки;
- выполнение «долгоиграющих» сценариев, таких как веб-сервера.

Первый вариант обычен, например, при создании React-приложения по шаблону. Современная клиентская веб-разработка, как правило, подразумевает наличие многих файлов с исходным кодом (таких как файлы компонентов). Чтобы подготовить production-версию, эти файлы нужно собрать в единый бандл, а перед этим прогнать через тесты. Эти задачи можно было бы решать разными способами, но промышленным стандартом стало использование npm-пакетов на платформе Node, которые работают практически совершенно одинаково на разных ОС. Например, для тестирования может быть использован инструмент Jest, для сборки (бандлинга) – Webpack. Кроме того, вызов соответствующих команд автоматизируется с помощью инструментов типа create-react-app. Возможно и код редактировать инструментом, который устанавливается как node-зависимость, но это не очень удобно, хотя популярные редакторы кода Atom и Visual Studio Code всё равно написаны на языке JavaScript/TypeScript с использованием фреймворка Electron.

Рассмотрим пример работы в рамках первого варианта. Допустим, мы хотим получить инструмент, который извлекает с сайта информацию о времени по Гринвичу. На начальном уровне сложности мы можем сделать следующее:

- установить платформу Node.js;
- убедиться, что также установлен менеджер npm (и если не установлен, то установить; подойдёт и альтернатива в виде yarn либо pnpm);
- создать сценарий на JavaScript.

Чтобы работать с Node, кроме установки вы можете использовать реплит (repl.it), node.new, виртуальный сервер, виртуальную машину, Docker.

Инструкции по установке Node.js приведены по адресу https://kodaktor.ru/g/node

**Примечание**
Платформа Node.js, как и вообще весь веб, изначально «настроены» на работу под управлением Linux/UNIX, поэтому для работы удобнее всего использовать либо сразу эти ОС (а также, разумеется, MacOS), либо виртуальные машины или контейнеры, в том числе подсистему WSL. В тексте пособия приводятся варианты команд, которые работают в перечисленных типах операционных систем.

Для MacOS с официального сайта скачивается pkg-файл. Для Ubuntu 20.04 и 22.04 работают следующие команды:

```
apt install nodejs npm -y
npm install -g n
n latest
```

Для управления версиями используется пакет **n**, и команда n latest как раз позволяет перейти к свежайшей версии, также можно указать точный номер:

Рисунок 1. Результат установки Node.js на MacOS и переход на другую версию

Может оказаться необходимым вначале выполнить `apt update` для обновления списков пакетов. После установки той или иной версии всегда можно использовать для выполнения кода именно её, примерно так:

```
/usr/local/n/versions/node/10.0.0/bin/node index.js
```

В целом универсальным и безопасным вариантом является использование Docker-контейнера (на этом этапе можно выполнить инструкцию https://kodaktor.ru/docker_intro_2022.pdf для тренировки работы с Docker). Если имеется файл с JavaScript и Docker, то всегда можно использовать контейнер с Node в качестве «открывашки» примерно следующим образом (с точностью до номера версии):

```
docker run  --rm  -v "$PWD":/usr/src/app -w /usr/src/app node:19 node index.js
```

(Чтобы зайти в REPL в контейнере достаточно выполнить
`docker run -it  --rm  node:19` **`node`**
– здесь t означает использование терминала для ввода, а i для вывода)

Пусть Node.js установлен, и у нас есть файл index.js со следующим содержимым:

```
fetch('https://www.worldtimeserver.com/current_time_in_GB.aspx')
.then(x => x.text())
.then(x => console.log(x.match(/\d\d:\d\d:\d\d ../)[0]));
```

Тогда для запуска сценария нужно выполнить команду:
`node .`
Точка означает файл по умолчанию (index.js).
Если всё в порядке и сервер времени доступен, то мы увидим результат в формате

```
12:34:56 AM
```

Мы можем переформулировать код с обращением к Интернет-ресурсу с помощью **await**

```
(async() => {
    const result = await (await fetch('https://www.worldtimeserver.com/current_time_in_GB.aspx')).text();
    console.log(result.match(/\d\d:\d\d:\d\d ../)[0]);
})();
```

Интерфейс Fetch API стал доступен в Node относительно недавно, и на этапе написания пособия всё ещё являлся экспериментальным. Более подробно о нём мы будем говорить в Главе 3.

Если мы работаем под Linux или MacOS и умеем выяснять куда после установки попал исполнимый файл node (which node), то можно воспользоваться синтаксисом shebang (hashbang) и добавить указание в файл, после чего сделать его исполнимым.

> **Примечание.**
> Летом 2022 г. включение синтаксиса hashbang в грамматику языка стало законченным предложением (finished proposal, 4-я стадия) и таким образом кандидатом на часть стандарта ES2023 (к лету 2023 г.). См. https://github.com/tc39/proposal-hashbang

```
#!/usr/bin/env node

fetch('https://www.worldtimeserver.com/current_time_in_GB.aspx')
.then(x => x.text())
.then(x => console.log(x.match(/\d\d:\d\d:\d\d ../)[0]));
```

После назначения права исполнения (chmod +x index.js) его можно запускать как команду, например ./index.js (а если добавить путь к нему в PATH то и просто по имени).

В случае использования await оборачивание кода в async-функцию…

```
(async () => {
   …
})();
```

…необходимо в файлах .js вне проекта поскольку на текущем этапе await можно использовать вне функции только в модулях. Чтобы сделать файл модулем, достаточно изменить его расширение на .mjs

Тогда сценарий сведётся к двум строкам:

```
const result = await (await
fetch('https://www.worldtimeserver.com/current_time_in_GB.aspx')).text();
console.log(result.match(/\d\d:\d\d:\d\d ../)[0]);
```

Второй способ задать режим модуля – использовать манифест проекта package.json и добавить в описание type="module"

В этом сценарии используется самый примитивный способ извлечь информацию – регулярный паттерн. Нет гарантии, что разработчики сайта оставят строку времени такой же. Чуть лучше будет извлекать данные из тега, который имеет вид

```
<span id="theTime" class="fontTS">12:34:56</span>
```

Но для этого уже нужно будет использовать парсер HTML, а для этого, в свою очередь, нужно создать проект и загрузить зависимость.

Выполним в папке команду `npm init -y` или `yarn init -y`

В результате мы получим базовый манифест проекта package.json – главным образом он нужен для хранения информации о зависимостях, но также и для определения характеристик проекта и хранения сценариев автоматизации. Вероятно, лучше приучить себя при работе со сценарием создавать отдельную папку и в ней инициализировать проект.

Далее следует выполнить команду установки зависимости:
`npm i jsdom`

> **Примечание**
> В тексте пособия в основном используются команды npm, причём по возможности однобуквенные. Команда `npm i` является синонимом `npm install`. Если вы используете yarn, то синонимом будет `yarn add`

Файл package.json приобретёт примерно следующий вид:

```
{
  "name": "demo",
  "version": "1.0.0",
  "type": "module",
  "license": "MIT",
  "dependencies": {
    "jsdom": "^20.0.0"
  }
}
```

Файл со сценарием принимает следующий вид: (https://kodaktor.ru/jsdom.mjs):

```
#!/usr/local/bin/node
```

```
import { JSDOM } from 'jsdom';

const myURL = 'https://www.worldtimeserver.com/current_time_in_GB.aspx';
const result = await (await fetch(myURL)).text();
const { document } = (new JSDOM(result)).window;
console.log(document.querySelector('#theTime').textContent.trim());
```

Пакет jsdom относится к числу тех, которые пытаются сделать сценарии изоморфными, т.е. работающими максимально эквивалентно в браузере и вне его. В данном случае он работает как своего рода полифилл (подстановка, воспроизводящая (пока) отсутствующую функциональность) во внебраузерном сценарии, позволяя использовать интерфейс document, которого нет в Node.js

☞ Мы рассмотрим ещё два примера решения задачи извлечения данных из веб-страницы в Главе 9

Отметим, что мощь Node.js возрастает по мере увеличения функциональности API. Мы употребляем здесь термин API в разных контекстах:

- Web API – веб-платформа, преимущественно браузерные технологии
- Node API – Node-платформа, в основном внебраузерные технологии
- RESTful API – функции, реализуемые веб-серверным приложением через обращение к URL в некоторой форме согласно схеме.

По адресу https://nodejs.org/dist/latest-v19.x/docs/api/fs.html вы можете ознакомиться с текущей на период подготовки данного пособия документацией Node API (на примере API взаимодействия с файловой системой). Обращения к файлам и папкам играют огромную роль в работе внебраузерных приложений. В качестве примера рассмотрите сценарий, выводящий в консоль собственный код («квайн»), обратив внимание на ссылку import.meta (возможность, включённая в стандарт ECMA2020):
https://kodaktor.ru/quine.mjs

При работе в браузере для быстрого экспериментирования используется консоль, а в Node.js – REPL, как и на многих других платформах. Выполнение команды node без указания имени файла приводит к переходу в режим REPL, т.е. диалоговый режим. С другой стороны, есть тенденция создавать собственные наборы команд на основе командной оболочки операционной системы, это называется CLI. У фронтэндовых инструментов и фреймворков имеются свои CLI, например webpack-cli.

Используя Node.js и конкретно пакет **commander** можно достаточно легко и быстро создать собственный CLI, но рассмотрение этого вопроса выходит за рамки данного пособия. Перейдём к рассмотрению важной и уже частично использованной терминологии.

### Понятия фронтэнда и бэкэнда

Традиционно принято считать, что веб – клиент-серверная технология, как и интернет в целом. Интернет по сути понятие скорее техническое; протоколы, которые определили интернет как сеть сетей, – это правила, описывающие бесшовную прозрачную доставку данных между компьютерами, принадлежащими к разным сетям. Веб же, в свою очередь, описывает правила высокоуровневого доступа к ресурсам, размещённым в интернете. Некоторое время назад были распространены почтовые клиенты типа Outlook Express или TheBat. Сегодня огромное количество пользователей работает с почтой через веб-интерфейс. Из одной из многих служб или сервисов, то есть вариантов использования интернета, таких как Gopher, веб превратился в главный гипермедийный интерфейс к ресурсам интернета.

При этом историю веба разделяют на периоды. **Первый** этап – «наивный» веб, в котором отправка запроса приводила к полной перезагрузке окна браузера и который имел минимальные средства автоматизации. **Второй** этап связывается с развитием автоматизации, появлением «веб-платформы» – API для программного обращения, в том числе AJAX (сценарии страницы обращаются к серверам асинхронно, в фоновом режиме, при этом части страницы обновляются независимо друг от друга), развитием соцсетей, авторского контента и сервисов типа онлайн-редакторов мультимедийного контента, а также с усилением позиций мегакорпораций, и, прежде всего Google. **Третий** этап связывается с блокчейном, токенами, Dark/Deep Web, анонимностью, децентрализацией (концепция Web3), а также семантичностью и онтологичностью (концепция Web 3.0). Под веб-платформой часто понимаются API, которые реализует браузер (https://developer.mozilla.org/en-US/docs/Web/API). Развитие платформы WebAssembly тоже скорее относится к «третьему» вебу. Разумеется, программная часть крайне важна, но не следует забывать и о вебе-для-пользователя.

Фронтэнд – это средство веба, отвечающее за взаимодействие с пользователем через гипермедийный интерфейс. Бэкэнд же отвечает за размещение и динамическую генерацию ресурсов, а также регулирование доступа к ним. Фронтэнд-технологии работают на стороне клиента, то есть в основном браузера. Основной размах фронтэнд приобрёл благодаря развитию вспомогательных инструментов, библиотек и фреймворков, которые размножаются с невероятной скоростью. Примером типичного фронтэнд-решения, которое, во-первых, обрело собственную экосистему с инструментами, библиотеками и рынком труда, а во-вторых, частично перешло в область бэкэнда, является компонентно-ориентированная библиотека React, которую также часто называют фреймворком. Начавшись с

решения проблемы повторного использования кода разметки со встроенным JavaScript и стилями, React занял нишу на другой стороне благодаря системе роутинга и технологии серверного рендеринга. Ещё одним связующим фронтэнд и бэкэнд понятием является состояние. Современные веб-приложения, основанные на потоке данных, оперативно сихнронизируют состояние между клиентом и сервером, используя для этого такие библиотеки, как Logux. Бэкэнд-технологии работают на стороне сервера, но с точки зрения веба главное, что они работают не в браузере. На бэкэнде гораздо большее разнообразие языков; в каждом браузере работает своя копия сценария, а на сервере может работать одно приложение, обслуживающее миллионы клиентов с их браузерами. Мощности браузера ограничены, в браузере пока нет реалистичного доступа к файловой системе. Серверным сценариям доступны огромные объёмы дискового пространства и вычислительной мощности, в том числе благодаря облакам и подпискам. На бэкэнде тоже успело сформироваться множество библиотек и фреймворков. Фулстэк-разработка охватывает уровни фронтэнда и бэкэнда, причём они пересекаются и переходят друг в друга. Фронтэнд-разработчик использует компоненты экосистемы Node.js, такие как npm-пакеты или сборщики бандлов, которые написаны на небраузерном JavaScript. По сути, для создания клиентского сценария используются части кода, не способные работать в браузере. На JavaScript написаны инструменты для тестирования. Таким образом Node.js охватывает также и область DevOps. Мы получаем моноязыковой фуллстэк – полный набор технологий, находящихся в экосистеме JavaScript и обслуживающих потребности как фронтэндеров, так и бэкэндеров. Основу веба составляет протокол HTTP, а также протоколы, которые взаимодействуют с ним, например WebSocket (веб-сокеты).

Разницу между веб-страницами, веб-приложениями и веб-сервисами мы рассмотрим далее в Главе 4.

Рассмотрим в качестве примера, как можно исполнять сценарий, написанный на TypeScript для работы с клиентской веб-страницей, с помощью инструмента, который создает временный девелоперский веб-сервер.

В пустой папке ней мы инициализируем новый проект. После появления файла манифеста package.json установим в этот проект в раздел девелоперских зависимостей пакет parcel.

```
npm i -D parcel
```

Это сборщик проектов, функциональность которого аналогична функциональности сборщика Webpack, с рядом интересных особенностей, одна из них заключается как раз в поддержке TypeScript.

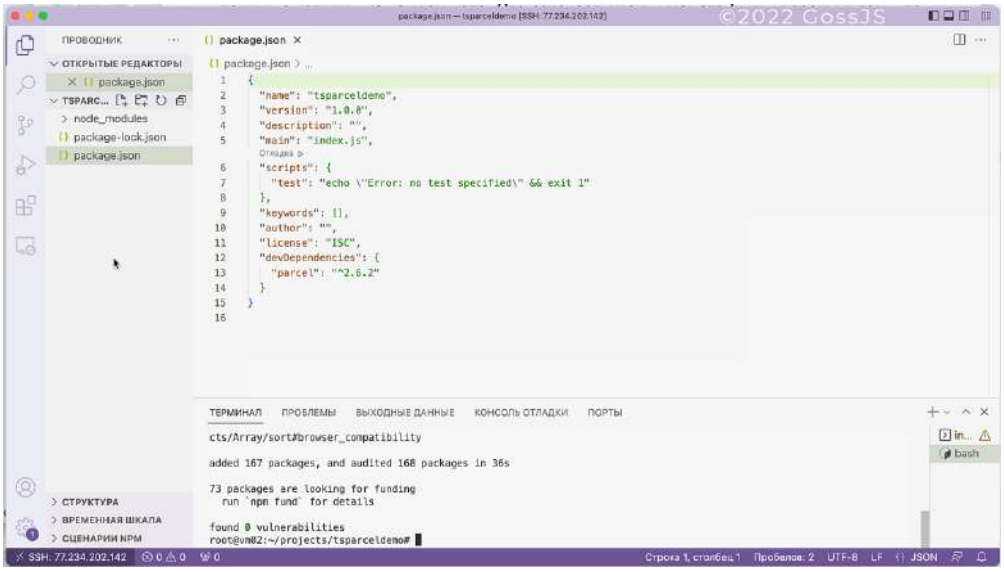

Рисунок 2. Parcel как зависимость в проекте Node.js

Нам нужно будет создать 2 файла. Первый файл — это веб-страница, второй – сценарий на TypeScript. Страница будет являться точкой входа (entry point) для сборки. Мы должны будем указывать её при вызове команды parcel в качестве entry point.

Это элементарная страница, в которой самый важный элемент — script, в котором атрибут src в открывающем теге связывает эту точку входа с файлом, содержащим сценарий.

```
<!DOCTYPE html>
<html>
 <head>
  <title>TypeScript</title><meta charset="utf-8">
 </head>
 <body>
   <h2>TypeScript</h2>
   <p><a href="http://www.w3c.org">W3C</a></p>
   <script src="index.ts"></script>
 </body>
</html>
```

Как можно видеть из примера,  у нас на веб-странице есть заголовок h2. Мы напишем сценарий, который добавляет еще заголовок h1. В файле index.ts в переменную со строковым типом мы запишем разметку. Далее мы создадим ссылку с типом HTMLElement на тело страницы document.body и, наконец, воспользуемся методом insertAdjacentHTML для того, чтобы вставить нашу разметку после открывающего тега <body>.

```
const s: string = '<h1>Привет, пользователь!</h1>';
const el: HTMLElement = document.body;
el.insertAdjacentHTML('afterbegin', s);
```

Мы можем запустить девелоперский сервер, выполнив команду npx parcel index.html

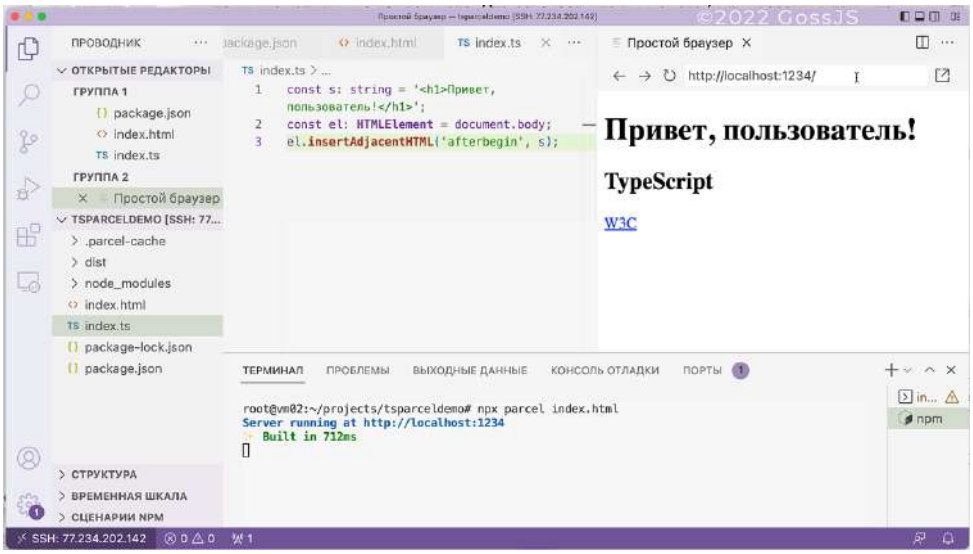

Рисунок 3. Запуск девелоперского сервера

После успешного выполнения команды parcel сообщает, что сервис запущен. Он будет доступен по адресу localhost:1234 (если этот порт окажется занят, то будет выбран свободный порт).

В папке node_modules/@parcel/reporter-dev-server можно найти код используемого в parcel девелоперского сервера.

Таким образом среда Node.js, исполняя команды npm (или любого другого менеджера пакетов), обеспечивает решение задач разработки для фронтэнда. Аналогично, Node.js управляет работой линтеров кода, препроцессоров CSS и т.д.

Рассмотрим некоторые моменты обращения с зависимостями чуть подробнее.

Если мы возьмём пустой проект и выполним команду npm i express, то
А) в файле package.json появится раздел dependencies

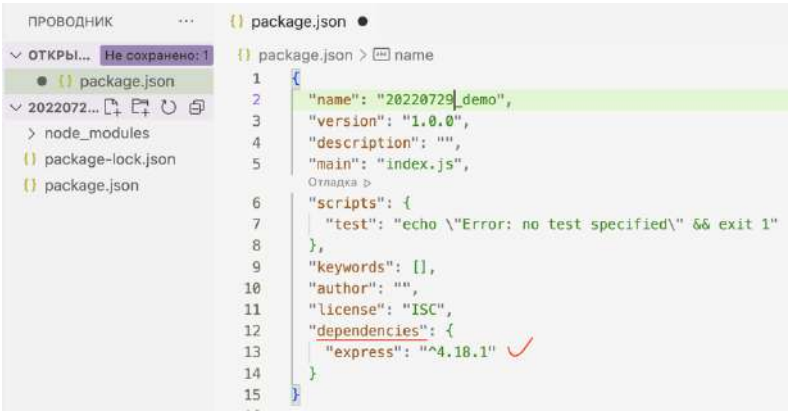


Рисунок 4. Express в списке зависимостей проекта

Б) содержимое будет скачано из интернета, мы получим папку node_modules, в которой будут находиться папка проекта express и множество папок его зависимостей. Содержимое папки express при этом не будет включать собственную папку node_modules.

Если мы хотим иметь express у себя на локальном компьютере, то можем поместить содержимое папки express отдельно и выполнить в ней npm i, тогда зависимости библиотеки express окажутся внутри в папке node_modules.

Допустим, у нас после этого есть папка /Users/ilyagosudarev/Яндекс.Диск.2021/express

Рисунок 5. Express и папка node_modules

Теперь в локальных проектах мы можем использовать этот путь чтобы забирать express из локальной папки, а не из интернета

npm i /Users/ilyagosudarev/Яндекс.Диск.2021/express

Кроме того, мы можем загрузить пакет из произвольного источника, указывающего на файл .tar.gz с архивом проекта или из репозитория. (Этот файл распаковывается в папку package, в которой размещаются файлы проекта.)

Чтобы создать на основе своего кода пакет, который затем можно инсталлировать по URL, нужно указать в package.json имя и версию, а затем выполнить npm pack

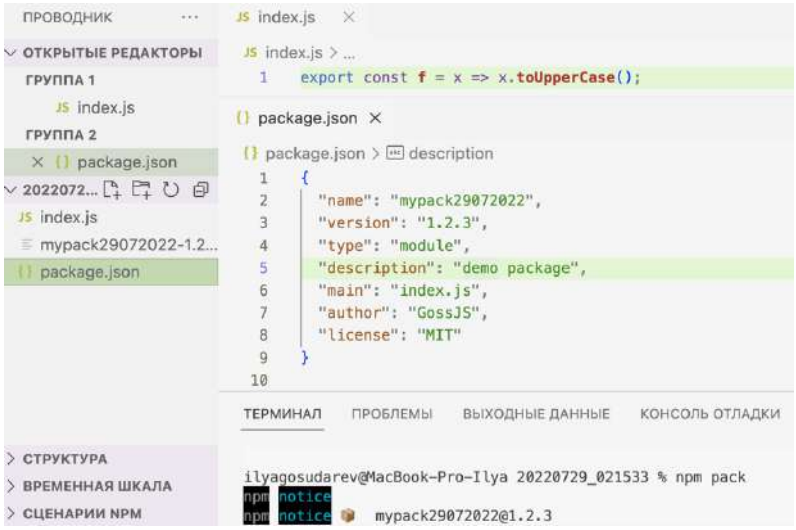


Рисунок 6. Выполнение команды npm pack

**Упражнение 1**

Выполните в новой папке следующие действия:

1. npm init -y
2. npm i https://kodaktor.ru/mypack29072022-1.2.3.tgz

Рисунок 7. Изменившийся раздел зависимостей

3. Создайте файл index.mjs
   Напечатайте в нём

```
import { f } from 'mypack29072022';
console.log(f('hello world'));
```

4. Выполните сценарий node index.mjs

Вы должны будете увидеть в консоли вывод заглавными буквами HELLO WORLD

Напомним, что традиционным способом включения сценарием в веб-страницы был и остаётся элемент script. Этот вариант изначально не подразумевал никакой модульности и, соответственно, изоляции фрагментов кода друг от друга. Чтобы предотвратить порчу объявленных на глобальном уровне имён, авторы кода и, особенно, библиотек, программисты привыкли оборачивать код в IIFE типа !function(){…}(). На платформе Node.js была реализована собственная модульность (CJS) на основе функции require, но она не прижилась на браузерной стороне. На современном этапе более или менее достигнута изоморфность в этом аспекте на основе нативной ES-модульности (import, export). В браузере мы используем script с атрибутом type="module" и достигаем практически эквивалентной функциональности. Например, вот тот же сценарий, экспортирующий функцию, в виде веб-страницы: https://kodaktor.ru/import_esm

Многие современные библиотеки предоставляют разные варианты своих дистрибутивов, так чтобы их можно было подключить просто как script

https://kodaktor.ru/_react2021/counter

или в виде ESM

https://kodaktor.ru/effector_react

В качестве популярного примера подключения широко используемой библиотеки нельзя не упомянуть lodash. Если мы работаем на внебраузерной платформе, то будем подключать библиотеку как npm-пакет (часто тоже есть версии специально для модулей ES)

```
npm i lodash-es
```

Но если при этом мы просто проводим эксперименты на Node.js или создаём отдельный сценарий (без папки проекта и манифеста), то можем воспользоваться традиционной формой сценария, такой как https://kodaktor.ru/lodash.js

(Его можно подключить и к веб-странице <script src="https://kodaktor.ru/lodash.js"></script>, после чего просто использовать глобальное имя _ и так же работает ESM-импорт https://kodaktor.ru/import_esm_lodash ).

Мы можем загрузить такой файл на свой локальный компьютер (допустим, в папку /Users/) и далее использовать разными способами:

В произвольном файле сценария напишем:

```
require('/Users/lodash.js');
```

в REPL:

```
require('/Users/lodash.js');
```

или

```
.load /Users/lodash.js
```

или

```
eval((await import('/Users/lodash.js')).default);
```

В связи с этим возникает вопрос: можно ли работать с кодом по URL напрямую?

Да, но это противоречит соображениям безопасности (впрочем, см. Главу 9). Тем не менее в порядке эксперимента можно позволить себе попробовать:

```
eval(await (await fetch(‘https://kodaktor.ru/lodash.js ‘)).text());
```

В период написания этого пособия можно было услышать разные мнения экспертов по поводу перехода от require к ESM. Мы будем стараться использовать ESM всюду где это достаточно рационально. Но, например, в экспериментах с использованием Node REPL это (пока что) просто невозможно, поэтому списывать require со счетов мы в любом случае не будем.

Экосистема npm является живой, развивающейся, по ней на момент написания этого пособия не было представлено исчерпывающей литературы на русском языке, но, во-первых, базовые сведения изложены в такой книге как [3], а во-вторых д-р Аксель Раушмайер публикует в блоге https://2ality.com/ обновляющиеся отчёты об эволюции JavaScript и TypeScript.

В этих источниках мы можем найти интересные сведения о npm-пакетах, которые не всегда легко отыскать в официальной документации.

В качестве примера рассмотрим, в чём разница между разделами dependencies и devDependencies в файле package.json. Она в первую очередь смысловая: зависимости из раздела devDependencies нужны только в процессе разработки проекта, но не в runtime (не в процессе его работы, запуска, использования). Библиотека модульного тестирования — типичная зависимость разработчика. Как следствие, такие зависимости, например, не должны включаться в бандл. Но система npm не ограничивается смысловым различием: npm будет устанавливать девелоперские зависимости, только если запустить npm install *внутри каталога этого конкретного пакета*. То есть, когда какой-то другой разработчик или пользователь будет устанавливать наш пакет как свою зависимость, этот раздел будет проигнорирован. Тем не менее нужно отслеживать ситуацию: системы развёртывания типа Heroku или Netlify могут по-своему трактовать это различие.

Протокол передачи данных – это стандарт, описывающий правила взаимодействия функциональных блоков при передаче данных, в случае веб-технологий основным протоколом будем считать HTTP (https://www.rfc-editor.org/rfc/rfc7230).

Это протокол прикладного уровня передачи данных, изначально в виде гипертекстовых документов в формате HTML, а в настоящий момент используется для передачи фактически произвольных данных. Основой HTTP является подход «клиент-сервер», то есть предполагается существование потребителей, клиентов которые инициируют соединение и посылают запрос, и поставщиков – серверов, которые ожидают соединения для получения запроса, производят необходимые действия и возвращают обратно сообщения с результатом.

HTTP в настоящее время повсеместно используется во Всемирной паутине для получения информации с веб-сайтов. Если информация, которая передаётся с помощью протокола HTTP шифруется с использованием протокола TLS, работающего на том же уровне, то мы имеем дело с безопасной версией HTTP – HTTPS.

HTTP – это протокол, в котором при каждом соединении клиент и сервер заново «узнают» друг друга, предыдущее состояние не сохраняется.

После передачи необходимой порции информации, соединение разрывается, после чего клиент должен инициировать его заново, и это очень мало подходит для реализации таких приложений как чат. В чате, когда один участник взаимодействия посылает реплику, сервер должен оповестить об этом других участников, то есть играть активную роль. Канал связи используется в оба конца: от клиента серверу и от сервера к клиенту. Для решения такой задачи чаще всего используется протокол, расширяющий возможности HTTP, протокол WebSocket.

Для исследования посылки запросов и получения ответов, проще всего использовать инструменты браузера и, конкретно вкладку, "Network", где мы можем видеть подробную информацию о деятельности браузера и сервера.

Отправим запрос на сервер kodaktor.ru, работающий на платформе Node.js и посмотрим заголовки.

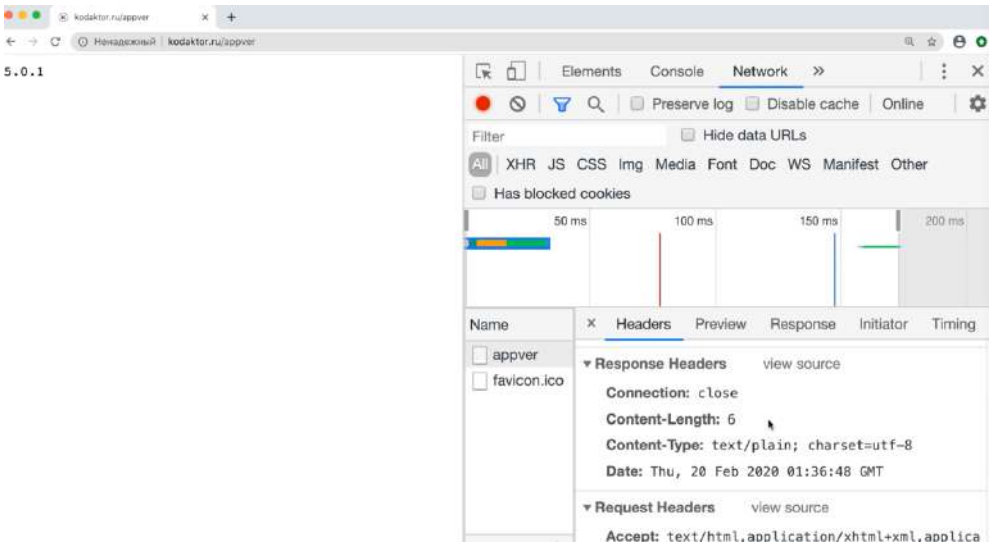


Рисунок 8. Ответ от сервера во вкладке Сеть инструментов веб-разработчика

Сначала показывается код ответа 200 – запрос успешно обработан, далее – заголовки, отправленные браузером, то есть клиентом, а затем заголовки, которые отправил сервер. Заголовок Connection: close говорит о том, что соединение завершено, Content-Length показывает количество переданных байт, Content-Type – тип содержимого или MIME-type, в данном случае это plain text (плоский, простой текст) в кодировке utf-8 (слово plain означает, что текстовые символы отображаются как есть, без интерпретации некоторых из них в качестве тегов или команд). Также присутствует заголовок с текущей датой.

То же самое будет происходить, если мы перейдём к безопасной версии протокола, потому что в данном случае сервер одинаково работает и по HTTP, и по HTTPS. Заголовки ответа те же самые.

Запрос веб-серверу строится в базовой своей форме как фраза в стиле команды:

Глагол /путь HTTP/1.1

Например,
GET / HTTP/1.1

Или соответственно:
POST /g/ko_resp HTTP/1.1

Путь отсчитывается от доменного имени сервера. Типичным http-клиентом является браузер. Когда мы вводим адрес в адресную строку или щёлкаем по гиперссылке, браузер автоматически отправляет запрос методом get.

Метод – это слово, которое здесь обозначает то же самое, что глагол.

Чтобы однозначно определить местонахождение информации в вебе, как правило, указывается полный адрес, также называемый Uniform Resource Identifier (URI).

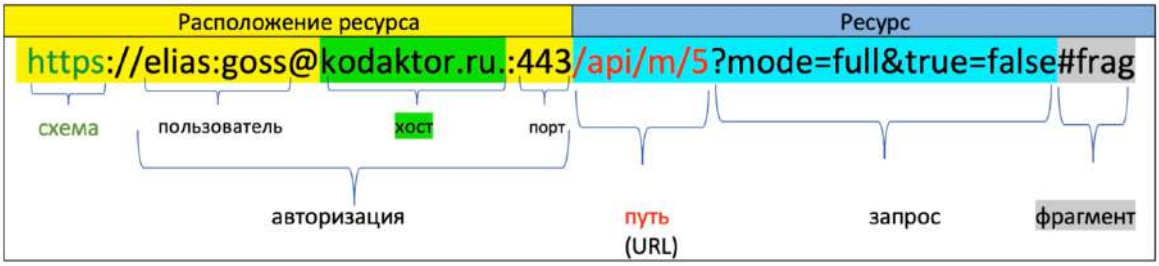


Рисунок 9. Структура URL

Иногда часть полного адреса, которая обозначена здесь как путь, называется URL, то есть в неё не включается доменное имя.

В полном адресе имеется схема протокола, возможность указать пользователя и пароля для доступа к ресурсу, доменное имя, хост, в данном случае kodaktor.ru, с точкой в конце, обозначающей корневой домен. Порт, в данном случае 443, соответствует безопасному протоколу HTTPS, и далее идёт указание или локация ресурса, относительно его главного расположения, то есть сервера с таким-то доменным именем.

В этом ресурсе мы можем выделить путь (path) или в узком смысле URL, который выглядит очень похоже на указание папок в файловой системе. В начале существования веба так и было: задача сервера состояла в том чтобы выдать клиенту файл, находящийся в данной папке и таким образом имело место прямое соответствие между путём к файлу в файловой системе сервера и URL, по которому обращается клиент. Это давно уже не обязано быть так. Файла вообще может не быть, и сервер может генерировать порции информации на лету, в ответ на конкретный запрос, интерпретируя части URL как команды и их параметры. В этом случае клиент использует сервер как удалённый исполнитель команд или процедур, а переход по URL становится в общем случае гораздо более существенным действием, чем просто получение информации.

**Примечание**

Когда URL завершается чем-то, что выглядит как файл с расширением, например https://kodaktor.ru/api2/time.html, это может говорить как о том, что с сервера загружается статика – файл time.html, так и о том, что расширение .html является опциональным и поддерживается генератором данных чтобы визуально подчеркнуть что будут выданы данные типа text/html. В случае с вышеуказанным маршрутом сервер выдаёт динамически контент с текущим временем независимо от того, каким будет расширение – пустым, либо .html либо .php (), но, чтобы показать отличие, при обращении с расширением .html текст выдаётся красным цветом. С точки зрения сервера и слэши и точка в URL-строке – просто

символы, которые сервер волен интерпретировать по своему усмотрению.

Далее у нас имеется так называемый **QueryString** или **search**, иначе говоря запрос, который состоит из пар вида ключ=значение, и если таких пар несколько, то они друг от друга отделяются знаком амперсанда. Часть после знака вопроса выделяется из URL только для удобства клиента в том смысле что сервер не обязан дифференцировать части URL до и после вопроса. И в конце может находиться **фрагмент**, обозначаемый хэшем (решёткой, знаком pound). Фрагмент или хэш значим только для браузера.

Для того чтобы получить какую-то информацию от веб-сервера, важнее всего знать доменное имя, или в крайнем случае IP-адрес.

По умолчанию, порт протокола HTTP – 80, то есть если запрос отправляется на порт 80, это скорее всего означает, что на сервере работает соответствующее программное обеспечение, откликающееся на этот порт, то есть именно ему, встретив такой номер в запросе, операционная система передаёт данные для обработки.

Большая часть компонентов URI является таким образом необязательной и может вообще не встречаться.

Информация в терминах HTTP передаётся четырьмя путями от клиента серверу:

- как заголовок;
- как глагол;
- как часть URL (информацию несёт сам путь, строка запроса – и для клиента фрагмент;
- как тело сообщения, что актуально для методов отличных от GET.

(Мы рассмотрим эти четыре пути в Главе 4 на материале модуля node:http)

Обычно считается, что при использовании глагола или метода GET используется только три первых пути. На самом деле это условие соблюдается в веб-клиентах и веб-серверах, которые достаточно строго реализуют HTTP, но при программной отправке и обработке запросов ничто не мешает заявить метод GET и при этом передать тело сообщения. Это же касается части строки после знака # (браузер не передаёт «фрагмент», но произвольный веб-клиент может и передать).

Большая часть важной информации видна в адресной строке браузера. Это нужно для самого элементарного действия – загрузки веб-ресурса для просмотра. Также с помощью GET посылаются команды открытых API, которые предоставляют веб-приложения типа социальных сетей или Google Documents.

Глаголы POST, PUT, DELETE и другие применяются для управления веб-приложением, а конкретно, например, для добавления, обновления или удаления данных.

Самый часто используемый глагол POST можно увидеть в HTML-форме.

Отметим, что такое разделение вносится именно протоколом HTTP, который находится на верхнем, прикладном уровне стека ISO/OSI. Для протокола более низкого уровня – например, TCP – нет разницы между get или post, а кроме того, нет разницы между запросом и ответом, информация просто передаётся как порции байтов в одном либо в другом направлении. И упомянутые выше веб-сокеты являются как бы вкраплением более низкого уровня в работу приложения высокого уровня.

Каким образом мы могли бы отправить запрос методом POST? Располагая только браузером, для этого проще всего создать форму. Пример находится по адресу:

https://kodaktor.ru/simpleform

Она открывается с помощью открывающего тега <form> и закрывается закрывающим таким тегом, содержит атрибут action, который указывает на сценарий, который будет отвечать на запрос, посылаемый формой.

Также есть атрибут method, и в данном случае он имеет значение POST.

Если открыть вкладку Network и посмотреть, как выглядит работа этой формы, то при нажатии на кнопку, мы увидим, что код ответа не 200, а 201.

Код состояния – часть первой строки ответа сервера при запросах по протоколу HTTP, он представляет собой целое число из трёх десятичных цифр.

Первая цифра указывает на класс состояния, за кодом ответа обычно следует отделённая пробелом поясняющая фраза на английском языке, которая разъясняет причину именно такого ответа.

Код 200 – это общий ответ о том, что запрос обработан успешно и ресурс выдан. А код 201 свидетельствует о том, что ресурс был создан.

В данном случае никакого ресурса на самом деле не создавалось, но веб-сервер, отвечающий на этот запрос, выдаёт именно этот ответ, для того чтобы показать что он распознаёт, что запрос был передан методом POST.

С цифры 4 начинаются коды, которые свидетельствуют об ошибках, связанных с неверным адресом.

Самый знаменитый код – 404 (не найдено). Это обычно результат ошибочных действий клиента, который мог, например, допустить опечатку в адресе (хотя автор сайта мог удалить страницу без какого-либо сигнализирования об этом, что в целом некорректно). С цифры 5 начинаются коды, которые свидетельствуют об ошибке на веб-сервере. Например, 500 (Internal Server Error), внутренняя ошибка сервера.

Альтернативой форме является программный вариант отправки, например, Fetch API. В свежих версиях Node.js он поддерживается так же, как в браузере, и мы рассмотрим этот вариант подробнее в Главе 3.

Прекрасными инструментами исследования процесса отправки запроса является приложения типа Postman или Insomnia.

Воспользуемся более низкоуровневыми инструментами для отправки запросов вручную, не задействуя браузер и начнём с программы telnet.

TELNET (teletype network) — сетевой протокол для реализации текстового интерфейса по сети (в современной форме — при помощи транспорта TCP), а также утилиты, реализующие клиентскую часть протокола. Современный стандарт протокола описан в RFC 854. В Windows, возможно, придётся установить telnet в разделе настройки дополнительных компонентов, а в Linux / MacOS он, скорее всего, доступен «из коробки».

Отправим серверу имя на русском языке, чтобы его ответом стало приветствие.

IP-адрес сервера kodaktor.ru – 151.248.115.32.

На этом IP-адресе находится несколько веб-серверов, для того чтобы указать, к какому именно мы хотим обратиться, необходимо послать заголовок Host.

Как корректно передать серверу через командную строку русское имя «Иван» методом GET?

Для этого нам нужно закодировать эти буквы с помощью процентного представления, которое формируется по определённому алгоритму. Русские буквы в URL оказываются парами вида %xx%yy, где x и y – шестнадцатеричные символы; например, буква Я (заглавная) выглядит как %d0%af.

Чтобы получить процентное представление, можно в консоли браузера или в REPL Node.js выполнить:

```
encodeURIComponent('Иван')
```

Результат:

'%D0%98%D0%B2%D0%B0%D0%BD'

Чтобы теперь передать это значение серверу с помощью telnet, в командной оболочке следует ввести

```
telnet 151.248.115.32 80
```

…и нажать Enter.

Соединение будет установлено, после чего остаётся ввести строку запроса с полученным выше значением

```
GET /unsafe_telnet?name=%D0%98%D0%B2%D0%B0%D0%BD HTTP/1.1
```

И завершить запрос двойным нажатием Enter (что приводит к появлению пустой строки)

```
-▶  telnet 151.248.115.32 80
Trying 151.248.115.32...
Connected to xn--l1afjd5e.xn--p1ai.
Escape character is '^]'.
GET /unsafe_telnet?name=%D0%98%D0%B2%D0%B0%D0%BD HTTP/1.1

HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 17
ETag: W/"11-pRL9A5O7GCtk+HBvE6PMBg"
Date: Sun, 31 Jul 2022 01:12:46 GMT
Connection: keep-alive

Hello, Иван!
Connection closed by foreign host.
```

Рисунок 10.  Telnet-подключение

Отправить запрос POST без формы можно только с использованием специальных инструментов, либо программно.

Заголовки – это строки, которые отделяются друг от друга переводами строки.

В мире Linux для отделения строк используется символ с кодом 10, который в JavaScript обозначается как \n

```
> '\n'.charCodeAt(0)
10
```

В мире Windows используется комбинация с символом \r

```
> '\r'.charCodeAt(0)
13
```

Когда мы набираем в обычном текстовом редакторе строку и нажимаем Enter, то в Linux добавляется символ \n, а в Windows – \r\n. Мы будем называть это в целом «переводом строки».

В GET-запросах нет тела, поэтому после заголовков он завершается, и это отмечается как два перевода строки (\n\n или \r\n\r\n).

В POST-запросах после заголовков и двух переводов строки следует тело сообщения. Клиент должен сообщить серверу длину тела в заголовке запроса Content-Length.

Например, отправим запрос на адрес https://kodaktor.ru/api/req/Илья/Госс и в качестве тела сообщения отправим name=Илья&familyname=Госс. Для этого нужно URL-кодировать русские буквы, т.е. запрос отправится по адресу

https://kodaktor.ru/api/req/%D0%98%D0%BB%D1%8C%D1%8F/%D0%93%D0%BE%D1%81%D1%81

Тело сообщения может выглядеть так:

name=%D0%98%D0%BB%D1%8C%D1%8F&familyname=%D0%93%D0%BE%D1%81%D1%81

Подсчитаем длину:

'name=%D0%98%D0%BB%D1%8C%D1%8F&familyname=%D0%93%D0%BE%D1%81%D1%81'.length

(получаем 65)

Ещё один заголовок, который важен в этом случае – это тип содержимого. Базовый тип для отправки POST-запроса – application/x-www-form-urlencoded

В командной оболочке следует ввести

```
telnet 151.248.115.32 80
```

и нажать Enter.

Соединение будет установлено (клиент скорее всего проинформирует, что Escape character is '^]' – т.е. что комбинацию клавиши CTRL и закрывающей квадратной скобки можно нажать для перехода в режим REPL программы telnet), после чего сначала мы последовательно вводим строки заголовочной части, нажимая после каждой Enter

- POST /api/req/%D0%98%D0%BB%D1%8C%D1%8F/%D0%93%D0%BE%D1%81%D1%81 HTTP/1.1
- Host:kodaktor.ru
- Content-Type:application/x-www-form-urlencoded
- Content-Length:65

Затем ещё раз Enter и строку с телом сообщения, после которой последний раз Enter

```
➜  telnet 151.248.115.32 80
Trying 151.248.115.32...
Connected to xn--l1afjd5e.xn--p1ai.
Escape character is '^]'.
POST /api/req/%D0%98%D0%BB%D1%8C%D1%8F/%D0%93%D0%BE%D1%81%D1%81 HTTP/1.1
Host:kodaktor.ru
Content-Type:application/x-www-form-urlencoded
Content-Length:65

name=%D0%98%D0%BB%D1%8C%D1%8F&familyname=%D0%93%D0%BE%D1%81%D1%81
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET,POST,DELETE,PUT,OPTIONS
Access-Control-Expose-Headers: X-Resp,Content-Type, Accept, Access-Control-Allow-Hea
Access-Control-Allow-Headers: X-Resp,Content-Type, Accept, Access-Control-Allow-Head
Content-Type: application/json; charset=utf-8
X-By: Ilya Goss (c)2017
X-Test: POST538c73d64461e13907bb95c51c38bfbc
Content-Length: 79
ETag: W/"4f-8Qv7yWopSfHQuR0rJQyoFQ"
set-cookie: connect.sid=s%3AqicwW-B_IuMT927LbT3h5F1W1zIsAIiV.Ujzz8aVbhtu0%2BrVclItCl
s=Tue, 02 Aug 2022 00:43:13 GMT; HttpOnly
Date: Mon, 01 Aug 2022 21:56:33 GMT
Connection: keep-alive

{"name":"Илья","familyname":"Госс","_METHOD":"POST","_NAME":"Илья"}Connection closed
```

Рисунок 11. Отправка POST-запроса в telnet

Telnet это очень низкоуровневый и далеко не самый удобный инструмент для экспериментов с протоколом HTTP, преимущество его в наглядности при обучении – все данные представлены «как есть». Он совсем не подходит для отправки данных по HTTPS, потому что обмен данными с участием TLS сложнее.

Гораздо более удобным является CURL или cURL. Он берёт на себя отправку таких заголовков как Content-Length и делает совершенно прозрачным обмен данными по HTTPS в большинстве случаев.

Все предыдущие шаги с telnet заменяются одной строкой с curl:

```
curl https://kodaktor.ru/api/req/%D0%98%D0%BB%D1%8C%D1%8F/%D0%93%D0%BE%D1%81%D1%81 -d 'name=%D0%98%D0%BB%D1%8C%D1%8F&familyname=%D0%93%D0%BE%D1%81%D1%81' -i
```

Чтобы сделать запрос с помощью JavaScript, будь то в REPL или в консоли браузера, мы можем использовать Fetch API, с которого начали Главу 1 и который рассмотрим в Главе 3:

```
await ((await fetch('https://kodaktor.ru/api/req/%D0%98%D0%BB%D1%8C%D1%8F/%D0%93%D0%BE%D1%81%D1%81', {method: 'POST', headers: {'Content-Length':'65','Content-Type':'application/x-www-form-urlencoded'}, body: 'name=%D0%98%D0%BB%D1%8C%D1%8F&familyname=%D0%93%D0%BE%D1%81%D1%81'})).text());
```

**Примечание**

В браузере Chrome чтобы выполнить fetch, скорее всего, нужно, чтобы во вкладке был открыт какой-нибудь сайт (т.е. чтобы это не была пустая вкладка).

**Примечание**

Мы рассмотрели основные характеристики базового протокола HTTP версии 1.1. Он несёт в себе черты диалогового режима, режима REPL – цикл «запрос-ответ». По мере развития в HTTP появляются функции, частично пересекающиеся с веб-сокетами (бинарный режим, возможность действовать активно, т.е. отправлять данные без явного отдельного запроса). По разным оценкам, летом 2022 г. примерно половина всех веб-сайтов использовала HTTP/2. Обе эти версии находятся на следующем после TCP уровне в ISO/OSI-стеке. В то же время в июне 2022 г. организация IETF стандартизировала HTTP/3 в документе RFC 9114, и эта версия базируется уже не на TCP, а на QUIC (мультиплексированном транспортном протоколе на основе UDP). По данным https://w3techs.com/technologies/details/ce-http3, около четверти из 10 миллионов веб-сайтов мирового веба в конце 2022 г. уже использовали HTTP/3.

Ранее мы использовали код, который одинаково работает в браузере или в REPL платформы Node.js. Теперь мы обратимся к коду, который работает только в Node.js.

Модуль **node:net** предоставляет асинхронный сетевой API для создания серверов на основе протокола TCP и отправки запросов. Все эти действия совершаются через TCP-сокет.

### Метод net.connect

Рассмотрим сначала отправку запроса:

```
void net.connect(80, '151.248.115.32').on('data', x =>
console.log(String(x))).write('POST
/api/req/%D0%98%D0%BB%D1%8C%D1%8F/%D0%93%D0%BE%D1%81%D
1%81 HTTP/1.1\nHost:kodaktor.ru\nContent-Type:application/x-www-form-
urlencoded\nContent-
Length:65\n\nname=%D0%98%D0%BB%D1%8C%D1%8F&familyname=%D0
%93%D0%BE%D1%81%D1%81');
```

Это обращение из REPL имеет тот же эффект, что рассмотренные в первой главе обращения с помощью telnet, Fetch API и curl. Методы on и write объединены в цепочку. Такое возможно благодаря тому, что результат соединения – сокет – реализован как **поток**, в свою очередь реализующий интерфейс EventEmitter. Событие data возникает в нём в нашем случае после того, как в него записали данные. Эти данные являются сырым содержанием HTTP-запроса. Собственно говоря, именно их мы ранее отправляли с помощью telnet. Но здесь они представлены как JavaScript-строка. В JavaScript есть синтаксис шаблонных строк с кавычками в виде обратного косого апострофа, и это позволяет вместо впечатывания символов \n просто нажимать Enter:

```
import { connect } from 'node:net';
connect(80, '151.248.115.32')
.on('data', x => console.log(String(x)))
.write(`POST
/api/req/%D0%98%D0%BB%D1%8C%D1%8F/%D0%93%D0%BE%D1%81%D
1%81 HTTP/1.1
Host:kodaktor.ru
Content-Type:application/x-www-form-urlencoded
Content-Length:65

name=%D0%98%D0%BB%D1%8C%D1%8F&familyname=%D0%93%D0
%BE%D1%81%D1%81`);
```

Листинг 1. Отправка запроса клиентским методом модуля node:net

Получается, что у нас асимметричное соединение – на «той» стороне работает HTTP-сервер, а «здесь» мы используем TCP-клиент. Но этот пример хорошо показывает, что запрос на «языке» HTTP это просто строка, и сервер делает веб-сервером именно способность разобрать её, интерпретировать и выдать ответ в подобном же формате. Поэтому следующим шагом мы создадим простейший веб-сервер с помощью модуля node:net.

**Примечание**
Обратите внимание, что современный стиль работы с модулями-пакетами предусматривает использование префикса node: перед именами встроенных модулей, к числу которых относится net. Далее в тексте исключения (ссылки без префиксов) сделаны для случаев, когда в той или иной среде префиксы (на этапе работы над пособием) не поддерживались.

**Примечание**
Интерфейс EventEmitter является важнейшим набором характеристик которые разделяют самые разные объекты и структуры платформы – это прежде всего способность регистрировать сигналы или события, связывая с ними функциональный код (коллбэк), и испускать эти сигналы, тем самым инициируя синхронное исполнение связанного кода.

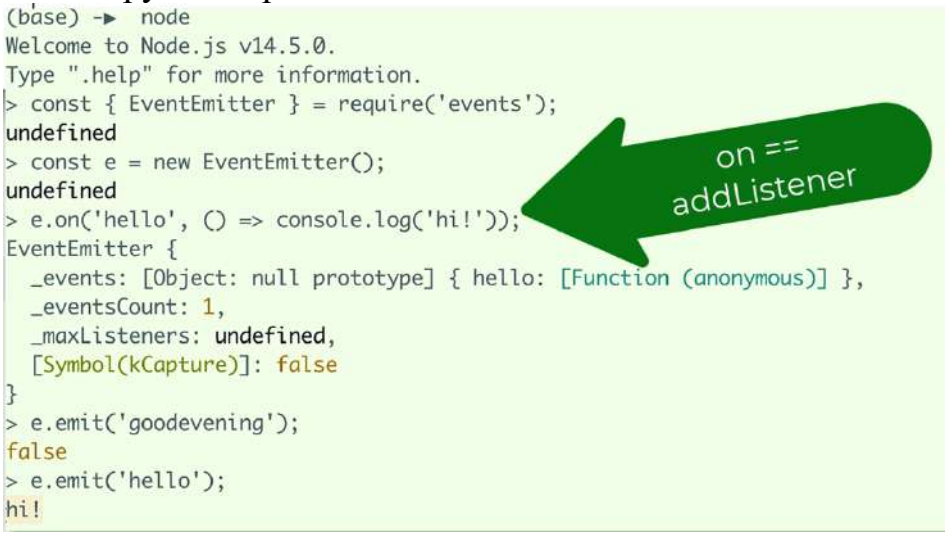

Рисунок 12. EventEmitter

### Метод net.Server

Мы можем запустить простейший TCP-сервер одной строкой прямо в REPL:

```
net.Server(sock => sock.end('hello')).listen(1234);
```

Этот сервер в ответ на все запросы выдаёт лишь строку hello. Браузер Chrome не прочитывает такой ответ (во всяком случае на порту, ином чем 80):

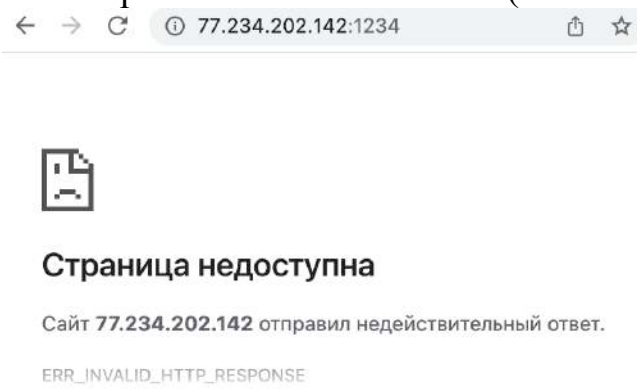


Рисунок 13. Ошибка обработки ответа браузером

В то же время curl, не являясь строгим веб-клиентом, распознаёт такой ответ:

```
-▸ curl 77.234.202.142:1234
hello%
-▸
```

Рисунок 14. Ответ, выданный Curl

> **Примечание**
> Если какая-либо версия curl не способна на это, можно добавить параметр, подавая запрос примерно так:
> curl 77.234.202.142:1234 --http0.9

Разумеется, «симметричный» клиент на Node.js без труда отправит запрос и отобразит ответ:

```
void net.connect(1234, '77.234.202.142').on('data', x =>
console.log(String(x))).end();
```

Чтобы превратить TCP-сервер в «настоящий» HTTP-сервер, достаточно добавить выдачу самой первой строки с версией протокола:

```
net.Server(sock => sock.end('HTTP/1.1 200\n\nhello')).listen(1234);
```

После этого любой браузер будет отображать минимально возможную веб-страницу, состоящую из простого (plain) текста. В простом тексте такие символы, как открывающая теговая угловая скобка (<, знак «меньше») не интерпретируются в качестве начала тега или иного специального фрагмента, а рассматриваются «как есть» наравне с цифрами или латинскими буквами.

☞ Мы использовали браузер, curl, telnet и программный вариант на JavaScript. В качестве эксперимента, если вы в достаточной мере владеете другим языком или языками программирования, напишите программный вариант на доступном вам языке.

Многим удобно отправлять запросы с помощью таких приложений как Postman или Insomnia. Также удобен клиент **httpie** (https://httpie.io/), которым легко пользоваться из командной строки:

```
-▶  http GET 77.234.202.142:1234
HTTP/1.1 200

hello
```

Рисунок 15. Выполнение команды httpie

Исторически функции сервера выполняли либо специально предназначенные для этого программы, либо сценарии на языках типа Perl. В книге Weaving the Web создатель протокола HTTP и языка HTML Тим Бернерс-Ли пишет, что разработал первый веб-сервер – «программное обеспечение, которое держит веб-страницы на компьютере и позволяет получать к ним доступ» (с. 29). Такой подход предполагал, что веб-страницы представляли собой файлы, содержащие как минимум простой текст, но в нормальной ситуации также и теги разметки, которые были заранее созданы и подлежали теперь обслуживанию со стороны веб-сервера. Это статический этап существования веба, и для многих авторов веб-ресурсов он растянулся на долгие годы.

**Примечание**

Тим Бернерс-Ли разместил первый веб-сайт для всеобщего доступа в августе 1991 г. Около 1999 г. многие веб-энтузиасты (и автор данного пособия) создавали веб-сайты по тому же принципу – сначала редактировали веб-страницы как отдельные файлы с помощью обычных текстовых редакторов, затем размещали их на статических веб-серверах типа GeoCites или Narod. В июне 2021 г. сэр Тим продал оригинальный код веба в качестве NFT за $ 5.4 млн.

Если отойти от статического подхода, то можно позволить скомпилированной программе либо сценарию генерировать разметку (и, возможно, другие ресурсы) «на лету», в процессе обработки запроса. При этом в роли «фронт-контроллера» может выступать статический веб-сервер, такой как Apache, а динамический код может передавать ему разметку для дальнейшей отдачи клиенту. Или динамический код может делать всё самостоятельно.

В первом случае для связи на стороне сервера были разработаны правила CGI (Common Gateway Interface) – соглашения о том, как должна считывать и выдавать данные программа, работающая под управлением веб-сервера. Это достаточно низкоуровневые соглашения. Есть более высокий уровень, например WSGI/ASGI в экосистеме Python, или сервлеты в экосистеме Java. Веб-сервер обычно помещает такие значения, как строку GET-запроса, в переменные окружения, доступные программе с помощью кода типа getenv("QUERY_STRING"). Данные в теле POST-запроса считываются из стандартного потока ввода с помощью кода типа fread(data, 1, num_bytes, stdin)

Второй случай как раз составляет предмет нашего изучения: приложение на Node само **является** сервером и получает данные непосредственно от модулей, реализующих стек протоколов интернета. Таким образом, серверное веб-приложение в контексте Node.js – это приложение, которое обеспечивает и реализацию HTTP-взаимодействия с клиентом, и всю бизнес-логику (решение содержательных задач предметной области), включая взаимодействие с данными.

Нужно отметить, что взаимодействия между приложениями, компонентами систем совершенно не обязаны осуществляться по протоколу TCP или UDP или HTTP и так далее. Транспорт сообщений может быть любым, как любым может быть и подход к организации сообщений.

**Упражнение 2**

Выполните шаги по работе с пакетом **dnode**

1. Используйте дистрибутив Ubuntu 20.04 и убедитесь что установлен python2.7
2. Перейдите к старой версии Node.js sudo n 10.0.0
3. В папке инициализируйте новый проект npm init -y
4. Установите зависимости PYTHON=/usr/bin/python2.7 npm i weak dnode
5. Создайте файлы server.js и client.js со следующим содержимым:

server.js

```
const dnode = require('dnode');
```

```
const server = dnode({
  mul(n, m, cb) {
    cb(n * m);
  }
});
server.listen(5555);
```

client.js

```
const dnode = require('dnode');

dnode
  .connect(5555,  (remote) => {
    remote.mul(10, 20,  n => console.log(`10 * 20 = ${n}`));
  });
```

Листинг 2. Код клиентской и серверной части приложения для dnode

Запустите сначала сервер, а затем в другом окне терминала клиент и убедитесь, что вы видите результат умножения двух чисел.

Здесь всё работает так, словно клиент вызывает функцию mul у объекта remote, и это было нормально, если бы мы перед этим импортировали объект remote из модуля, но «объект» явно берётся из сетевого соединения. Выглядит так, будто мы получили из соединения объект с функцией или некую ссылку на объект, который находится на сервере.

Это можно интерпретировать следующим образом: мы вызываем удалённую процедуру, передаём ей числа 10, 20 и коллбэк, который находится в коде клиента. И при вызове

```
cb(n * m);
```

… этот коллбэк вызывается «на другой стороне».

Стиль RPC (Remote Procedure Call) можно рассматривать как аналог подхода к проектированию систем, в которых главный компьютер выполняет вычисления, а подключающиеся к нему терминалы лишь передают ему свои запросы с кодом, который следует выполнить.

☞ Вернитесь к этой идее в Главе 6, при обсуждении REST и в Главе 8, при исследовании Socket.io

**Примечание**

При установке зависимостей в Упражнении 2 используется инструмент node-gyp – он компилирует аддоны (нативные модули, написанные на C или C++, которые должны быть скомпилированы на конкретной машине). Этот инструмент требует наличия в системе вспомогательных средств для компиляции, таких как Python, make,

gcc. Успешность выполнения Упражнения 2 протестирована на экземпляре машины с Ubuntu 20.04 после установки пакета python2.7 вручную; версия npm была 5.6.0.

Вот некоторые современные кейсы эффективного использования Node.js в области серверных веб-приложений:

- асинхронные API, не связанные с тяжёлыми вычислениями,
- посредничество между клиентом и «тяжёлым» бэкэндом на уровне BFF (Backend-for-frontend),
- Edge-сервера, для которых критично время отклика.

Отметим, что высокую эффективность в обработке запросов, не связанных с тяжёлыми вычислениями, платформе Node.js обеспечивает реализация «цикла событий» (алгоритм Event Loop), которая была представлена в движке V8. В то же время современные среды выполнения (runtime) могут полагаться и на другие движки. Более подробно о цикле событий, паттерне «Реактор» и оптимизации очереди см. книгу [4] и её более свежие издания.

## Глава 3. Fetch API

Платформа Node.js длительное время развивалась параллельно или альтернативно развитию JavaScript в браузерах, но начиная с какого-то этапа браузерная и внебраузерная платформы стали служить друг для друга стимулами к развитию. В некоторых случаях важные функции появлялись сперва на Node.js, а затем были реализованы браузерами, а в некоторых – наоборот. Fetch API является примером технологии, которая сначала была внедрена в браузере а затем на платформе Node.js.

Часто веб-платформой называют именно браузерные технологии. На странице https://developer.mozilla.org/en-US/docs/Web/API можно видеть обновляющийся перечень так называемых Web API, к числу которых относится и Fetch.

Переход к Web 2.0 был ознаменован с точки зрения пользователей развитием социальных сетей и появлением нового опыта взаимодействия с веб-страницами. Ранее после нажатия кнопки отправки формы, например при поиске, вся страница обновлялась целиком. Сервер рендерил новую страницу и отправлял её клиенту. С увеличением нагрузки это стало непозволительным разбазариванием трафика. Новый опыт заключался в том, что стали обновляться части страниц, а кроме того, например при вводе поисковой строки стали появляться варианты окончания:

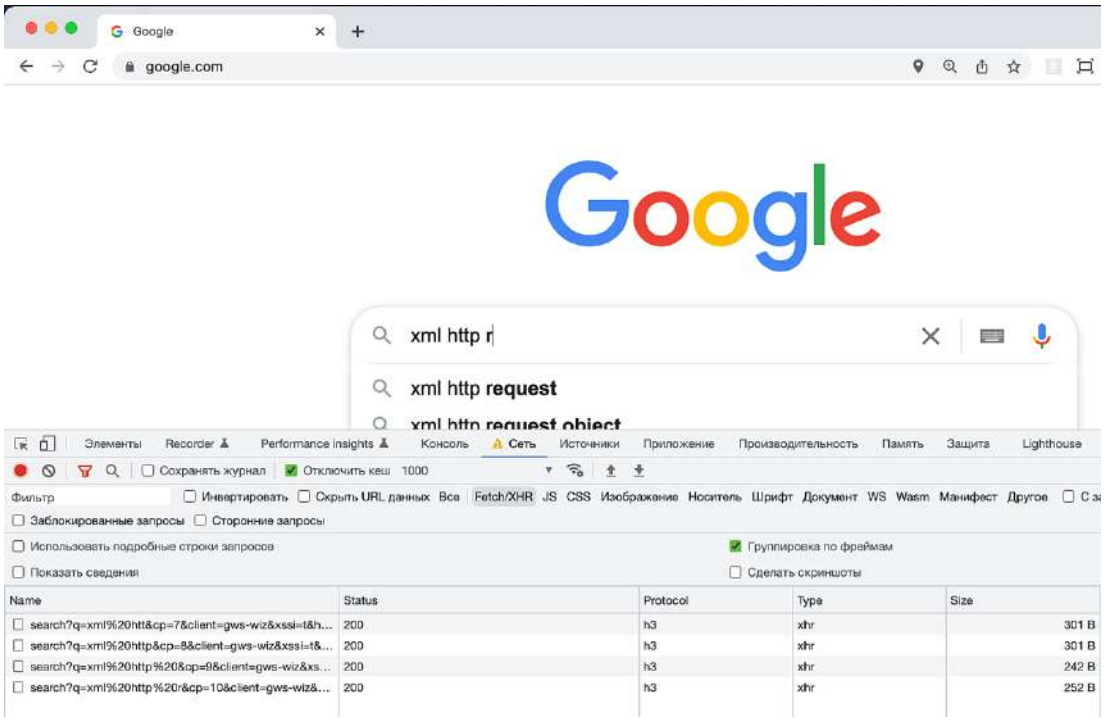


Рисунок 16. Фоновые запросы на странице поисковой системы

## Основа: AJAX

Технология программирования клиентских фоновых (асинхронных) запросов была впервые представлена как часть Trident в браузере Microsoft Internet Explorer 5.0 в 1999 г. Спустя примерно 6 лет был внедрён термин AJAX – Asynchronous JavaScript And XML. Впоследствии вместо XML для обмена данными стали использовать JSON, но название осталось прежним. AJAX – это не столько технология, сколько подход, который может быть реализован с помощью разных инструментов. В середине 2000-х таким инструментом был главным образом объект XMLHttpRequest.

```
var xhr = new XMLHttpRequest() || new
ActiveXObject('Microsoft.XMLHTTP');
xhr.open('GET', 'https://kodaktor.ru/g/ko_resp?cors', true);
xhr.send();
xhr.onreadystatechange = function(){
    if (this.readyState != 4) return;
    console.log(this.responseText);
    // вместо this можно и xhr
};
```

Листинг 3. Клиентский код, реализующий традиционный AJAX-запрос

Выражение в первой строке примера использует сокращённую схему вычислений: если в браузере присутствует объект XMLHttpRequest (т.е. это скорее всего браузер Mozilla), то использовать его, а иначе это Internet Explorer со своим ActiveX-элементом.

Ранний стиль подразумевал использование назначение функций DOM Level 0 (современный подход предполагает подключение слушателей событий из DOM Level 2) и использование внутреннего состояния readyState (как в объекте WebSocket).

Более современный вариант выглядит примерно так:

```
const xhr = new XMLHttpRequest();
xhr.addEventListener('load', e =>
  console.log(e.currentTarget.responseText)
);
xhr.open('GET', 'https://kodaktor.ru/g/ko_resp?cors');
xhr.send();
```

Листинг 4. Реализация традиционного AJAX-запрос в более современной форме

На Node в качестве устаревшего аналога AJAX можно рассматривать модуль request: https://kodaktor.ru/g/request

Однако уже достаточно давно веб-разработчики перешли к Fetch API. Как и у многих других API и у самого языка JavaScript, у Fetch есть спецификация: https://fetch.spec.whatwg.org/#fetch-api

Поэтому Fetch доступен в виде реализаций: в браузерах и вне браузеров.

До недавнего времени, чтобы fetch был доступен на Node.js, следовало запускать систему с ключом: `node --experimental-fetch` (версия 17.5, февраль 2022 г.). Однако в свежих версиях (например, 19.6.0), наоборот, доступен ключ `node --no-experimental-fetch` (хотя технология всё ещё в экспериментальном статусе).

Имплементация fetch на Node связана с проектом Undici (https://undici.nodejs.org/#/), в рамках которого для Node был написан с нуля HTTP1.1-клиент.

В платформу вводятся глобальные имена fetch, Request, Response, Headers. В отличие от XHR, fetch изначально промисифицирован (возвращает объект Promise). С учётом возможности использовать await на уровне модуля обращение к fetch становится особенно удобным.

Полифиллом для Node версий младше 17.5 является пакет node-fetch. Автор данного пособия собрал на его основе один файл, который можно скачать и импортировать из него метод fetch: https://kodaktor.ru/nf/fetch.mjs

Следующее обращение на период написания пособия является полностью изоморфным, т.е. будет совершенно одинаково работать в консоли браузера и в Node.js REPL:

```
await ((await fetch ('https://kodaktor.ru/g/ko_resp?cors')).text());
```

Несколько неуклюжая конструкция с двойным await в этом однострочнике объясняется тем, что fetch является источником двух последовательных промисов. Если разложить работу fetch по двум промисам, то получится так:

```
const pr1res = await fetch(myURL);
const pr2res = await pr1res.text();
```

pr1res будет экземпляром класса Response и в том числе содержит:

- методы, возвращающие промис (blob, arrayBuffer, text, json, formData);
- свойство типа «поток» (body)

Конструктором body является ReadableStream, и на этапе написания этого пособия существует разница в обращении с потоками «вебного» происхождения и «нативного» (Node.js). Здесь мы зафиксируем ситуацию на конец 2022 г.

### О потоках

Технология fetch содержит два пути работы с контентом: получение информации одной законченной порцией и считывание приходящих данных по мере поступления.

Потоки абстрагируют перемещение байтов по каналу. Наряду с асинхронностью и событийностью потоки представляют собой важнейшую основу программирования и разработки на Node.js. Мы уже сталкивались с этим понятием в Главе 2 при обсуждении сокета и модуля node:net.

Стандартные потоки представляют собой заранее определенные каналы передачи данных между приложением и средой: стандартный ввод (stdin), стандартный вывод (stdout) и стандартный поток ошибок (stderr). В приложении Node эти каналы обеспечивают взаимодействие между приложением Node и терминалом, своего рода механизм прямого «общения» с приложением. Потоки process.stdout и process.stderr отличаются от других потоков Node.js: их «под капотом» используют соответственно console.log() и console.error(), и запись в них может быть по обстоятельствам синхронной либо асинхронной.

Мы можем запустить приложение, перенаправив вывод ошибок в файл:

```
node index.js 2> error.log
```

Рассмотрим типичный кейс получения потока на основе файла:

```
import { createReadStream } from 'node:fs';
createReadStream('./package.json')
.on('data', d => console.log(d));
```

В результате вызова createReadStream создаётся объект с методом on. Это сокращенная форма записи для функции EventEmitter.addListener, получающей те же параметры; аналог addEventListener в Web API. Он позволяет привязать к событию многократный коллбэк, т.е. мы подписываемся на событие 'data' и устанавливаем, что всякий раз при его наступлении должна вызываться указанная нами функция.

Это так называемый непрерывный (flowing) режим, при котором данные проталкиваются, передаются (push) коллбэку-обработчику по мере их прибытия. Режим с событием data (и end) – классический, известный как **Streams1**. Событие data отправляется при получении нового фрагмента данных, готового к использованию, а событие end — при потреблении всех данных. Событие error, естественно, отправляется при возникновении ошибки.

На определённом этапе был введён интерфейс **Streams2**, позволяющий работать в более гибком дискретном режиме, non-flowing (по умолчанию чтение из потока заключается в подключении обработчика события readable, которое уведомляет о наличии новых данных, готовых для чтения. Затем в цикле выполняется чтение данных, пока не опустошится внутренний буфер. Метод read() обеспечивает синхронное чтение из буфера и возвращает объект Buffer или String, представляющий фрагмент данных. Получается, что мы опрашиваем данные и извлекаем (pull) их по требованию). Эта область продолжает эволюционировать, и нужно следить за обновлениями в API.

Важная характеристика потоков – конвейеризуемость. Вот два эквивалентные способа вывести данные из файла потоковым способом:

```
createReadStream ('./package.json')
.on('data', d => console.log(d));          // <Buffer  .... >
и
createReadStream ('./package.json')
.pipe(process.stdout);
```

Если в программе функции могут образовывать композицию, то сущности потокового происхождения могут образовывать конвейер:

- UNIX $ a | b | c | d
- Node.js > a.pipe(b).pipe(c).pipe(d)
- То же самое:
  - a.pipe(b)
  - b.pipe(c)
  - c.pipe(d)

Рисунок 17. Сравнение конвейеризации в разных средах

Объединение в конвейер родственно также цепочкам промисов и последовательности middleware.

**Примечание**

В начале 2023 г. на 2-й стадии рассмотрения находится предложение о внесении с стандарт ECMAScript оператора конвейера https://github.com/tc39/proposal-pipeline-operator с такой формой записи: a |> b |> c |> d

Метод pipe позволяет программисту не заботиться о контроле «скорости» передачи данных из одного потока в другой (превышение объема внутреннего буфера потока, обратное давление, backpressure). Если Writable поток работает на запись медленнее, чем их передает источник данных Readable, то передача данных «ставится на паузу» до тех пор, пока Writable «не просигнализирует» (очистка буфера), что он готов принимать следующую порцию данных.

```
источник.pipe(приёмник)
Readable.pipe(Writable)
Readable.pipe(Transform).pipe(Writable);
```

У потоков из Web API существует метод pipeTo, который требует родственного типа потока. В Node.js-потоках работает pipe(process.stdout), а с потоком «вебного» происхождения не получится выполнить pipeTo(process.stdout) – сперва нужно выполнить преобразование. Приведём конкретный пример, иллюстрирующий это:

```
import { Writable } from 'node:stream';

const webWritable = Writable.toWeb(process.stdout);
const pr1res = await fetch('http://kodaktor.ru/g/ko_resp?cors');
pr1res.body.pipeTo(webWritable);

(или в более коротком варианте)

const webWritable = new WritableStream({ write(x) { process.stdout.write(x);
} });
const pr1res = await fetch('http://kodaktor.ru/g/ko_resp?cors');
pr1res.body.pipeTo(webWritable);
```

Листинг 5. Объединение потоков в конвейер вне браузера

Если бы мы попытались выполнить pr1res.body.pipeTo(process.stdout), то получили бы сообщение об ошибке, что на вход pipeTo подан объект типа WriteStream, в то время как нужен WritableStream.

При этом, конечно, с точки зрения Node.js process.stdout – это именно WritableStream

process.stdout instanceof **Writable** === true

Несмотря на эту сложность, все эти виды потоков в конечном итоге совместимы и позволяют создавать почти изоморфный код (более подробно см. в статье Акселя Раушмайера https://2ality.com/2022/06/web-streams-nodejs.html#writing-to-writablestreams-via-writers)

**Упражнение 3**

По адресу https://kodaktor.ru/api2/net.txt клиент получает данные большого объёма. С помощью потока определите, сколько раз в получаемом массиве символов содержится символ с десятичным ASCII-кодом 48

**Упражнение 4**

I. Пусть дан файл 1.txt, который состоит из строки 1111111

Напишите код на Node.js, который открывает этот файл как поток, считывает символы побайтово (используйте highWaterMark) и увеличивает каждый символ на 1, после чего сохраняет файл. Т.е. после первого прогона вместо 1111111 в файле должно оказаться 2222222 и так далее.

II. Создайте трансформирующий поток, который осуществляет промежуточное преобразование содержимого файла, как в предыдущем пункте этого упражнения.

Для этого импортируйте (затребуйте) класс Transform из модуля node:stream

Создайте функцию преобразования:

```
const transformplus1 = function (c, enc, cb) {

  this.push(.....);

  cb();
};
```

Ваше приложение должно с точки зрения результата работать в точности как https://kodaktor.ru/plus1_demo

Также вы можете проверить его работу с помощью curl:

```
curl https://fork.kodaktor.ru/plus1 -d '3333333' -H 'Content-type:'
```

Результатом будет:

4444444

☞ Когда вы ознакомитесь со следующей главой, реализуйте эту функцию как маршрут серверного приложения: при ответе на запрос методом POST с помощью pipe прогоняйте поток запроса через этот новый поток:

```
const plus1 = new Transform({ transform: transformplus1 });
r.pipe(plus1).pipe(rs);
```

### Отправка запросов

Рассмотрим отправку данных методом POST с помощью fetch и получение информации в виде законченной порции:

```
const myURL = 'https://kodaktor.ru/api/posts';
const tit = '05.09.2022 23:38:15';
const headers = {"Content-Type" : "application/x-www-form-urlencoded"};
const method = 'POST';
const cont = Math.random();
const body = `title=${tit}&categories=work&content=${cont}`;
await (await fetch(myURL, { method, headers, body})).json();
```

Листинг 6. Отправка данных на сервер методом POST с помощью fetch-запроса

При выполнении такого запроса возвращается идентификатор созданного поста, допустим 160.

Теперь чтобы извлечь информацию о конкретном посте (например, с номером 160) нужно отправить гораздо более простой запрос

```
const { title } = await (await fetch('https://kodaktor.ru/api/posts/160')).json();
```

Что же касается считывания из пулл-потока, то здесь ситуация пока выглядит сложнее. Можно использовать две техники:

- Цикл, аналогично построчному чтению из БД или из файла в цикле
- Рекурсивный промис

Рассмотрим первый вариант с использованием наиболее «лобового» подхода, а именно цикла с выходом по break

```
const myURL = 'https://kodaktor.ru/api/chunks';
const { body } = await fetch(myURL);
```

```
const reader = body.getReader();
let result = '';

while (true) {
  const { done, value } = await reader.read();
  if (done) {
    console.log('Поток исчерпан');
    break;
  }
  result += new TextDecoder('utf-8').decode(value);
}
```

Листинг 7. Запрашивание данных с помощью пулл-потока в цикле

Здесь метод read будет вызван столько раз, сколько порций данных (чанков) выслал сервер (в следующей главе мы вернёмся к понятию чанка).

Рассмотрим второй вариант. В нём используются промисы и рекурсивный вызов функции-коллбэка.

```
reader.read().then(function process({done, value}) {
    if (done) {
      console.log(result);
      return;
    }
    result += new TextDecoder('utf-8').decode(value);
    reader.read().then(process);
 });
```

Этот вариант можно переписать с помощью await:

```
(async function process({done, value}) {
    if (done) {
      console.log(result);
      return;
    }
    result += new TextDecoder('utf-8').decode(value);
    process(await reader.read());
})(await reader.read());
```

Листинг 8. Запрашивание данных с помощью пулл-потока рекурсивным промисом

В связи с этим можно сформулировать вопрос (**https://kodaktor.ru/imagefetch**):

Дан элемент img и код

```
const myURL = 'https://kodaktor.ru/pattern.png';
const imresp = await fetch(myURL);
```

Как, не используя в коде далее myURL, вставить программно изображение из imresp в имеющийся элемент img?

Ответ:

```
const img = await imresp.blob();
const reader = new FileReader() ;
reader.readAsDataURL(img) ;
reader.onload = function(){
  document.querySelector('#i').src = this.result;
};
```

В этой короткой главе мы рассмотрели сущностные черты fetch и современные примеры с его использованием. Важно, что fetch стал изоморфным API, который реализуется на основе спецификации движком в браузере и в Node.js. В Deno он поддерживается «из коробки» – см. Главу 9.

## Глава 4. Модель приложения на Node.js

Зафиксируем, чем отличается приложение на Node.js от обычного веб-сервера, например такого как Apache. Обычный веб-сервер главным образом обслуживает файлы, то есть его основная задача – по запросам выдавать веб-страницы, такие как index.html. Помимо веб-страниц могут быть выданы картинки или музыкальные файлы, аудиофайлы и т.д. Обычный веб-сервер типа Apache требует наличия определённой конфигурации сайта, которая предполагает, что для хранения файлов сайта выделена некоторая отдельная папка, и в системе также установлены определённые разрешения, которые позволяют разрешать поступающие запросы и выдавать соответствующие файлы. Если для того, чтобы выдать файл требуется нечто большее, чем просто найти его в файловой системе, веб-сервер типа Apache передаёт управление соответствующим программам.

Также характеристикой обычного веб-сервера является то, что на запросы всеми http-методами веб-сервер типа Apache отвечает одинаково – в режиме, который можно условно назвать method-agnostic, т.е. «не замечая» указания метода (для веб-сервера Nginx характерно несколько иное поведение по умолчанию, но в любом случае всё определяется конфигурацией).

В отличие от обычного веб-сервера, для приложения на платформе Node.js лишь одним из кейсов является отдача файлов, так называемой статики. Приложение совмещает в себе логику веб-сервера и содержательной или предметной части, которая делает нечто полезное, выполняет вычисления, извлекает и обрабатывает данные и формирует ответ для клиента, в том числе для браузера.

Разрабатывая приложения на платформе Node.js, мы можем во всех деталях описать реакцию на запрос, кастомизируя код ответа, сообщение которое получает клиент и само содержание этого сообщения.

Теперь рассмотрим, чем веб-сервер на Node.js отличается от любого другого приложения. Когда мы создаём сценарий на языке JavaScript, размещаем его в файле и таким образом получаем модуль, который выполняется с помощью платформы Node, то фактически мы уже имеем приложение. Оно может отработать и завершиться, если решает какие-то задачи, не требующие постоянного присутствия в памяти. Приложение выполняется, пока в стеке или очереди заданий есть невыполненные задания, в том числе приложение может находиться в цикле обработки запросов по тем или иным протоколам. И если речь идёт о протоколе HTTP, приложение «слушает» некоторый характерный для этого протокола порт (80/443), отвечает на запросы, то мы говорим о веб-сервере.

Веб-сервер на Node.js, как правило, работает с объектами, которые обычно называются req или res (request/response), реализующими запрос и ответ. Такое приложение формирует заголовки и контент, в том числе считывая контент из уже имеющихся (статических) файлов.

Веб-сервер можно реализовать на основе существующих базовых модулей, например модуля **node:http**. Но этот ручной низкоуровневый подход в основном используется в демонстрационных или учебных целях, чтобы показать, как это в принципе делается, но для того чтобы решать сколь бы то ни было жизненные задачи, мы реализуем веб-сервер и приложения, которые включают в себя веб-сервер, с помощью тех или иных библиотек и фреймворков.

Возможно, мы привыкли к тому, чтобы считать веб-ресурсом просто файл, который доступен по определённому адресу, например обычную веб-страницу, которую можно открыть на нашем собственном локальном компьютере или можно предоставить внешнему миру через ту или иную сеть, но в действительности веб-ресурс – это нечто более широкое. Это совершенно необязательно файл как таковой, это скорее данные, которые в итоге оказываются доступными запросившему их клиенту. А то, каким именно образом они были сформированы, остаётся внутренним делом сервера. Веб-ресурс может, например, быть динамически сгенерирован на лету, в момент запроса.

Веб-приложение рендерит представления (View), то есть на основе имеющихся у него данных, возможно извлечённых из каких-то хранилищ, создаёт то, что в браузере выглядит как пользовательский интерфейс, как

текст, картинки, вёрстка, элементы управления, элементы форм, выпадающие списки и т.д., одним словом UI (user interface). На контрасте с этим, веб-сервис или API работает не столько для пользователей, не столько для браузера, сколько для автоматизированных сценариев, которые программным способом посылают запросы, и выдаёт в первую очередь не разметку, не веб-страницы, а данные в каком-нибудь удобном формате, наиболее распространённым из которых, наверное, на сегодняшний день продолжает оставаться JSON ( MIME-тип application/json).

Запросы могут при этом посылаться самыми различными методами, здесь нет абсолютно никаких приоритетов; если приоритетным методом для веб-приложений, для веб-ресурса является GET, то в случае API методы могут быть самые разные, и этим методам ставятся в соответствие определённые действия, совершаемые на стороне веб-сервиса.

Веб-сервис тоже склонен выдавать данные некоторой стандартной формы по определённой договорённости, которая существует между клиентской и серверной частью, то есть это не часть UI, не элемент представления, а данные которые нуждаются в определённой обработке на стороне клиента, возможно фрагментарные данные, которые используются для обновления частей пользовательского интерфейса, возможно для вычислений и т.д.

Итак, веб-приложение / веб-ресурс:

- работает для браузера в окне;
- выдаёт text/html в ответ на GET и POST запросы;
- рендерит представления / UI.

Веб-сервис / API:

- работает для сценариев;
- выдаёт application/json в ответ на запросы разными методами;
- посылает данные по некоторой стандартной схеме.

Рассмотрим теперь, какими именно способами мы можем обеспечить запуск приложения, в данном случае на платформе Node.js. Мы можем воспользоваться данным компьютером, если нам достаточно экспериментальных целей, целей локального тестирования. Если хочется предоставить доступ к тому, что происходит на локальном компьютере, внешнему миру, мы можем разрешить доступ из локальной сети, если же речь идёт об интернете, то можем воспользоваться различными сервисами, позволяющими пробрасывать сетевые запросы внешнему миру.

Мы можем воспользоваться специфическими сервисами, такими как runkit (см. пример: https://kodaktor.ru/g/runkit) для встраивания кусочков кода, с тем чтобы они выполнялись для тех пользователей, которые посетят страницы, куда эти кусочки кода были встроены, и увидели результат работы, возможно даже вступили в некоторое взаимодействие с этим выполняющимся кодом.

В последнее время получила распространение технология **serverless**, так называемые бессерверные сценарии, название которое некоторым образом

вводит в заблуждение, потому что на самом деле сервер, разумеется, присутствует и выполняет свою функциональность, но можно абстрагироваться от его настроек (т.к. инфраструктура обеспечивает всё необходимое, включая масштабирование). От программиста, который работает с serverless, требуется написать только содержательный код, который что-то делает, то есть производит предметные, содержательные действия с данными, которые поступили по запросу. Как правило этот содержательный код оформляется в виде функций, экспортируемых из пользовательского модуля, и возвращающих тот ответ, который вышестоящая инфраструктура должна передать конечному потребителю. Их часто называют λ–функциями или просто лямбдами (см. Главу 6).

Если нам нужно взаимодействие, если мы хотим заниматься кодированием совместно с другими разработчиками, видеть результаты друг друга, возможно одновременно кодировать и исправлять действия друг друга или дополнять их, то подойдёт такая среда как repl.it.

Если же речь идёт о production, о предоставлении доступа работающему приложению конечным пользователям, то можем выбрать Platform as a Service (PaaS – платформа как сервис), такое решение как Яндекс.Облако, Vercel, Netlify, Heroku.

Или же мы можем создать экземпляр виртуального хостинга в широком диапазоне предоставляемых решений, и в самом мощном варианте мы можем арендовать физический сервер, с которым производить любые желательные для нас действия.

Перейдём теперь к запуску сервера на localhost. Вообще localhost – это доменное имя локального компьютера, данного компьютера, на котором работает программист, обучающийся, разработчик, к которому можно обратиться по адресу: 127.0.0.1, находясь на этом самом компьютере. Аналог этого адреса в случае IPv6: [::1], что является сокращением 0:0:0:0:0:0:0:1

Для предварительной разработки бывает удобно запускать приложение на локальном сервере. Если мы просто дважды щёлкнем по HTML-странице, то она может открыться в браузере, мы увидим в адресной строке схему протокола file://, это позволяет посмотреть отдельные статические страницы, но для сколь-либо реалистичного поведения современного приложения, нам необходимо чтобы в браузере документы открывались по схеме протокола http://, чтобы их браузеру выдавал именно веб-сервер. И мы можем написать простой веб-сервер, использовать Apache, Nginx и т.д., либо использовать встроенные в веб-сервер популярные языковые экосистемы. Часто достаточно бывает именно файлового сёрвинга, то есть выдачи файлов, например из текущей папки, и чтобы развернуть локальный сервер в виде приложений, мы можем запрограммировать его или воспользоваться готовым решением. В случае Python – это модуль http.server.

Рассмотрим команду

```
python3 -m http.server 1234
```

Она запускает веб-сервер на 0.0.0.0:1234 – при этом используется модель чёрного списка (сервер по умолчанию *открыт* для внешнего доступа и для закрытия требуется особый параметр); чтобы добавить ограничение его можно прикрепить к localhost таким образом:

```
python3 -m http.server --bind localhost 1234
```

Другой вариант:

```
python -m SimpleHTTPServer 4321
```

При этом по умолчанию поддерживается только GET (иначе выдаётся сообщение 501 Unsupported method)

В Node.js роль development-сервера может выполнять сочетание команды npx и глобального npm-пакета типа serve, http-server или live-server.

Для запуска простого сервера веб-статики можно использовать команду:

```
npx static-server -p 4321
```

При этом также работает модель чёрного списка и по умолчанию используется только GET (иначе ошибка 405 Method Not Allowed).

Убедитесь, что после запуска все нижеследующие команды вернут один и тот же результат:

- curl [::]:4321
- curl [::1]:4321
- curl '[0:0:0:0:0:0:0:0]:4321'
- curl '[0:0:0:0:0:0:0:1]:4321'
- curl '[0:0:0:0:0:ffff:7f00:1]:4321'
- curl localhost:4321
- curl 127.0.0.1:4321

Запустим серверное приложение на платформе Node.js для начала на localhost, то есть на нашем собственном компьютере.

В самом минимальном варианте веб-сервер только «слушает» порт и даже ничего не отвечает на запросы. Напомним, что порт – это часть сокета, то есть оконечной точки соединения. Порт может быть двухбайтовым целым числом, которое фактически служит идентификатором конкретной программы, выполняющейся под управлением операционной системы. Исторически сложилось, что веб-сервер работает на порте 80 по умолчанию для протокола HTTP, и на порте 443 – для протокола HTTPS.

Это значит, что, когда приходит сетевой запрос, в нём указан номер порта. Обрабатывая запрос, операционная система ищет программу, которая заняла этот порт, и если порт не был занят, а программа его заняла, другая программа уже не может занять тот же порт.

Программы, которые что-то «слушают», то есть постоянно находятся в памяти и выполняют по сути бесконечный цикл, часто называют сервисами или службами. В терминологии Unix/Linux – это демоны, фоновые

программы. Веб-сервер – это как раз пример сервиса. Он постоянно работает и слушает порт.

> **Примечание**
> Веб-сервер – это сервис в «линуксовом» понимании. Веб (World Wide Web) часто называют сервисом Интернета примерно по этой причине – множество серверов, которые круглосуточно работают в Интернете и слушают запросы определённого типа. Но не следует смешивать это с веб-сервисом (мы рассмотрели это понятие ранее).

Рассмотрим сценарий из одной строки. Этот сценарий можно запустить в REPL, то есть в оболочке команд Node.js. Для этого нужно запустить Node, и выполнить эту команду:

```
require('node:http').Server().listen(4321);
```

… после чего программа начнёт выполняться.

В другом терминале выполнив команду обращения к localhost и соответствующему порту, мы не увидим сообщение об ошибке о том, что соединение невозможно. Вместо этого курсор замирает, и никакого ответа не получается. В написанной нами программе не предусмотрено никакой реакции на событие прихода запроса, а по умолчанию модуль node:http не реагирует на запросы самостоятельно.

Если теперь в REPL нажать Ctrl+c, то есть выйти из REPL, то это остановит выполнение нашего серверного приложения.

> **Примечание**
> В Linux мы также можем узнать PID созданного процесса по порту: `fuser -v 4321/tcp` либо `lsof -i :4321 | grep LISTEN` и после этого уничтожить процесс командой kill.

Вернувшись к терминалу, в котором мы подали запрос, мы увидим появившийся ответ, пустой (Empty reply from server). И если теперь (после остановки сервера) снова выполнить обращение к серверу, мы получим сообщение о том, что соединение невозможно (Failed to connect).

Итак, мы рассмотрели следующие два варианта:

1) порт свободен, его никто не слушает, соединение невозможно;

2) порт занят, его кто-то слушает, однако никто не отвечает.

> **Примечание**
> Разумеется, сервер можно остановить не только нажатием Ctrl+c (или Ctrl+d, или вводом директивы .exit)
> Вот программный вариант (важно, что экземпляр сохраняется в переменной):
> ```
> const s = require('node:http').Server().listen(4321);
> s.close();
> ```

Далее мы будем работать с кодом в основном не в REPL, а в виде файлов сценариев, причём по возможности с использованием нативной модульности ECMAScript. Для редактирования исходного кода рекомендуется использовать текстовый редактор Visual Studio Code (Когда мы вносим изменения в код, то сохраняем их и нам требуется снова выполнить команду запуска. Это хочется автоматизировать, и до появления версии 19 задачу наблюдения за изменениями исходного кода и перезапуска выполнял пакет nodemon. Однако начиная с версии 19 можно использовать параметр --watch команды node.)

Модуль **node:http** предоставляет нам фабричную функцию Server (или createServer), которая, будучи вызвана, возвращает объект класса http.Server (https://nodejs.org/api/http.html#http_class_http_server)

Кроме того, этот объект реализует интерфейс EventEmitter.

```
import { Server } from 'node:http';
import { EventEmitter } from 'events';

const s = Server();
console.log(s instanceof EventEmitter); // true

s.listen(4321);
```

Листинг 9. Демонстрация того факта что http-сервер есть экземпляр EventEmitter

Последний факт означает, что мы можем подписать сервер s на событие request, которое возникает, когда клиент обращается с HTTP-запросом. Кроме того, этот факт означает, что мы можем без участия клиента сгенерировать (emit) такое событие, что повлечёт за собой запуск подписанной функции – это используется при тестировании.

Добавим обработку запроса. Начиная с этого момента мы работаем в файле (назовём его server.mjs и запускаем node --watch server.mjs)

```
import { Server } from 'node:http';

const s = Server();
s.on('request', (req, res) => res.end('OK\n'));
s.listen(4321);
```

Теперь при обращении клиента или инициировании события request вручную мы будем получать ответ:

```
curl localhost:4321 -i
HTTP/1.1 200 OK
Date: Sun, 01 Aug 2022 15:28:47 GMT
Connection: keep-alive
Content-Length: 3

OK
```

Итак, веб-сервер с базовой функциональностью по умолчанию никак не реагирует на запросы – эти реакции нужно определять самостоятельно. Здесь усматривается логика JavaScript – если создать на веб-странице кнопку и не определить слушатель события click, кнопка тоже не будет реагировать на щелчки.

По умолчанию сервер выдал ответ с кодом 200. Это всегда можно переопределить.

Кроме того, мы видим, что по умолчанию отсылается минимальный набор HTTP-заголовков. Среди них – заголовок Content-Length, сообщающий число переданных байт. В нашем случае это следующие байты:

```
4f
4b
0a
```

Первые два – ASCII-коды английских букв O и K, а третий – символ переноса строки \n.

Метод listen, который инициирует процесс слушания соединений, также принадлежит серверу. Если указать номер порта, который уже используется, то получим неотлавливаемую фатальную асинхронную ошибку (Error: listen EADDRINUSE).

Если же в вызове метода listen убрать номер порта, то он будет назначен автоматически из числа неиспользуемых в данный момент. Кстати, клиенту порт назначается автоматически всегда. По умолчанию при таком способе запуска сервер будет доступен другим компьютерам локальной сети. В методе listen мы можем указать, что прослушиваем только локальный сетевой

интерфейс (прибайндить), и этим отключить возможность посылать запросы локальному веб-серверу с других компьютеров локальной сети.

```
.listen(4321, '::1' ...)
```

Ниже приведён пример сценария с сервером, который не знает заранее, какой порт будет слушать. Он выдаёт номер после запуска. Вы можете получить сценарий командой `wget https://kodaktor.ru/j/port.mjs` и запустить командой `node port.mjs`

```
import { Server } from 'node:http';
const s = Server();
s.on('request', (req, res) => res.end('OK\n'));
s.listen(() => console.log(`Порт: ${s.address().port}`));
```

Вы получите номер порта вида 54321. Выполните обращение к localhost:54321
(подставьте номер порта, полученный в результате работы сценария)

Такие пакеты, как portfinder, находят свободный порт более «явным» образом. При этом когда приложение запускается на PaaS типа Heroku, среда предоставляет порт, и в методе listen указывают выражение вида process.env.PORT || 4321 либо process.env.PORT ?? 4321. Выражение с оператором || реализует сокращённую схему вычислений: если каким-либо способом была установлена переменная окружения PORT, оператор || использует её, иначе вычисление будет продолжено и будет использован правый операнд с числом. Это удобно для запуска приложения в разных средах. На сегодняшний день есть смысл перейти от оператора || к оператору ?? (Nullish coalescing) из ECMAScript 2020, семантика которого идеально соответствует рассматриваемой ситуации.

Так как событие request самое важное, его обработчик может быть передан в вызов функции Server в качестве слушателя события по умолчанию. В тех примерах, которые отличаются от нижеприведённого только содержанием этого обработчика, мы будем приводить лишь его код.

```
import { Server } from 'node:http';

const s = Server((req, res) => res.end('OK\n'));
s.listen(4321);
```

Эта функция – **асинхронный коллбэк**. Данный факт очень важен для понимания того, как основанный на Node.js веб-сервер обходится с запросами и объясняет его высокую эффективность при обработке большого числа одновременно поступающих запросов. Мы ещё вернёмся к этому, а сейчас

рассмотрим, как мы можем контролировать выполнение ответа в режиме пошагового выполнения.

## Отладка

Для отслеживания хода выполнения сценария используется инструмент отладки (debugging). Естественно, мы можем получить всю полноту информации о том, что используется в программе из документации, но также полезные сведения можно извлечь в ходе отладки. Мы можем использовать различные инструменты отладки. Базовым инструментом является команда, когда мы запускаем Node с ключом inspect-brk и указываем имя сценария.

В нашем случае это будет index.js или index.mjs. При запуске такой команды создаётся подключение, которым пользуется клиентская часть отладчика, и в её роли может выступать браузер Chrome и его DevTools.

Существует также инструмент, который называется NDM от Google, но поскольку мы уже начали работу в редакторе кода Visual Studio Code, логично будет воспользоваться встроенным инструментом отладки, который представлен в этом редакторе, он достаточно удобен. Сейчас у нас имеется приложение, которое откликается на запросы поступающие на localhost и на порт 4321. Мы просто получаем ответ и ничего более. Теперь задача состоит в том, чтобы этот ответ посылать  не в программе, чтобы мы могли его вручную отправить в режиме отладки.

Для этого предназначены кнопка Run and Debug и команда **debugger**, которая играет роль триггера точки останова.

Рисунок 18. Команда debugger

Мы должны будем естественно остановить работу предыдущего сеанса, открытого в терминале, после этого нажать на кнопку Run and Debug и выбрать вариант Node.js.

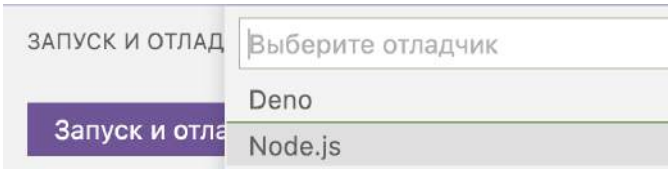


Рисунок 19. Выбор отладчика в Visual Studio Code

Теперь наше приложение работает в режиме отладки, и, если обратиться к приложению, то есть послать клиентский запрос, то сработает эта функция, обработчик запроса и сработает точка останова.

Для удобства откроем ещё один терминал и напишем команду curl localhost:4321, и после нажатия Enter будем наблюдать как сработает точка останова.

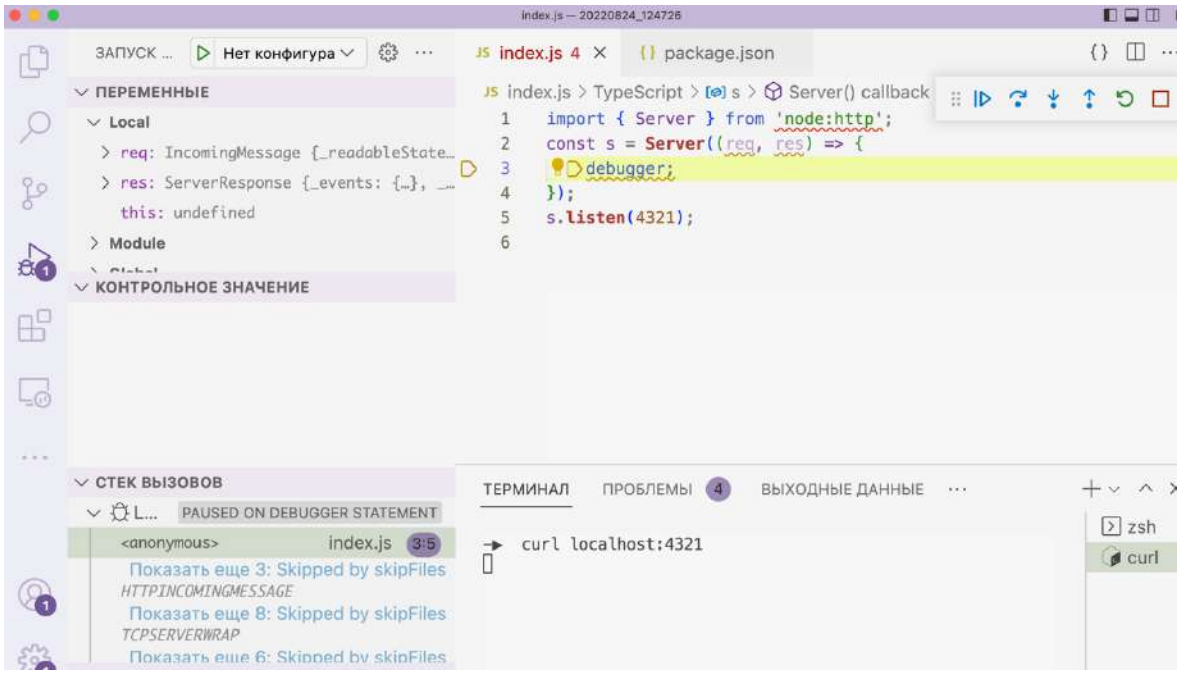


Рисунок 20. Срабатывание точки останова

После срабатывания точки остановка в Debug console мы видим сообщение о том, что отладчик запустился и можем вручную выдать ответ ждущему его клиенту…

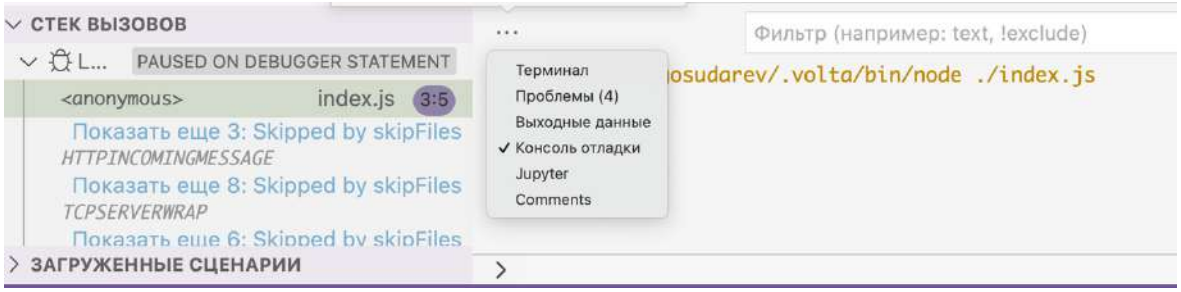


Рисунок 21. Консоль отладки

… то есть написать res.end и предоставить какой-нибудь простой ответ.

Рисунок 22. Ручная отправка ответа сервера в консоли отладки

После появления ответа, для того чтобы завершить этот цикл обработки запроса, мы нажимаем на кнопку **Continue**, то есть продолжить.

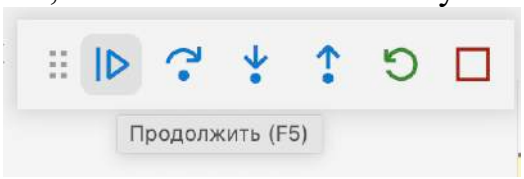


Рисунок 23. Кнопки управления отладкой

После этого всё можно повторять заново. Но давайте теперь немного поисследуем то что происходит. Например, посмотрим чем является **req**, который указан в качестве одного из параметров обработчика события запроса.

Мы здесь видим, что req это экземпляр класса IncomingMessage. В свою очередь res – это экземпляр класса ServerResponse. И IncomingMessage и ServerResponse являются классами, которые доступны во встроенном модуле HTTP. Поскольку у нас есть доступ к переменной req, мы можем исследовать свойства, которые доступны в req. Одним из таких свойств является url. Оно показывает запрашиваемый маршрут. Свойство req.method отображает метод, которым был отправлен запрос клиента, сейчас это GET.

Кстати, через ссылку req.client доступен TCP-сокет, который используется в данном HTTP-соединении. После отправки запроса и срабатывания отладчика мы можем в консоли отладки написать

req.client.**write**('one');

req.client.**end**('two');

и нажать Continue

И тогда curl с параметром -i покажет только onetwo, без всяких заголовков HTTP, потому что мы отправили данные в сыром низкоуровневом виде, как если бы использовали низкоуровневый модуль node:net.

И ещё отправим точно такой же запрос, но методом POST (curl localhost:4321 -XPOST). Мы можем увидеть, что свойство req.method теперь

содержит POST. Это показывает, что серверное приложение в той форме, которую мы сейчас перед собой видим, характеризуется свойством которое мы ранее обозначили как method-agnostic. То есть оно делает одно и тоже, независимо от того каким методом был послан запрос (впрочем, то же самое касается и маршрута или пути, который указывается после имени хоста в комбинации с номером порта).

Построенное нами приложение не зависит ни от маршрута, ни от метода. Если мы хотим, чтобы оно по-разному реагировало на запросы, приходящие на разные маршруты и разными методами, мы должны оперировать свойствами req.url и req.method для того чтобы дифференцировать по этим параметрам.

Если в ответ на запрос выполнять сначала res.write и только потом res.end, то ответ будет состоять из совокупности посланных данных, после того как мы завершим цикл обработки запроса. Обратите внимание на заголовок ответа Transfer-Encoding: chunked. Информация при обработке запроса поступала в ответ порциями – чанками.

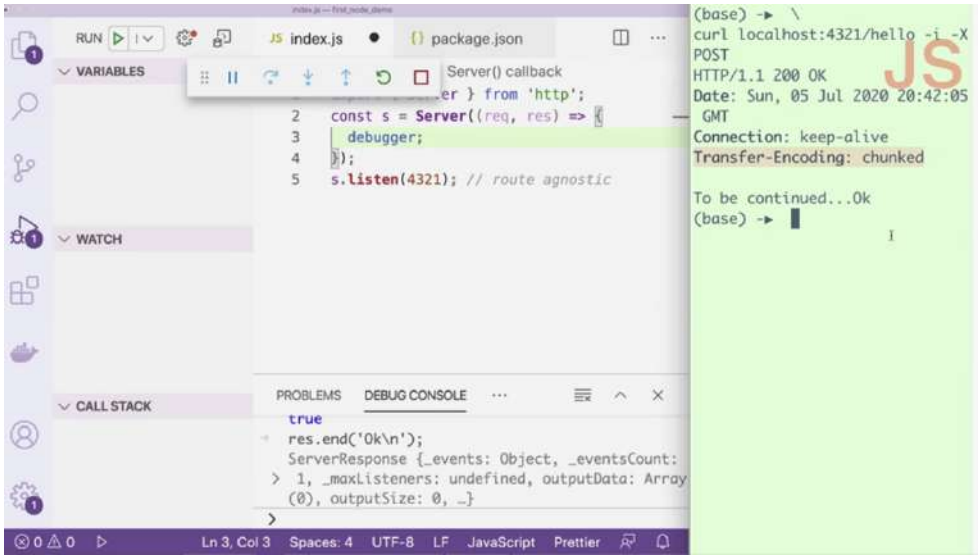

Рисунок 24. Ответ типа Transfer-Encoding: chunked

В этом случае не происходит автоматического вычисления общей длины сообщения и браузеру предлагается принимать информацию порциями, а потом выдать эти порции как единый информационный блок. Это возможно благодаря тому, что методы write и end являются на самом деле методами потоков.

☞ Подробное освещение темы потоков не входит в задачи нашего пособия – мы изложили лишь некоторые сведения в Главе 2, но вы можете самостоятельно проделать следующий эксперимент: импортировать класс Stream из стандартного модуля node:stream и проверить тот факт, что объект res является его инстансом (или экземпляром). Вы должны будете получить в консоли значение true.

Рисунок 25. Объект res как инстанс потока

Далее, обратившись к документации по классу Stream, вы увидите, что экземпляры этого класса поддерживают методы write и end. Используя метод write можно посылать в поток порции информации. В данном случае это поток, поддерживающий запись. А с помощью метода end можно завершить отправку данных. Вызов метода end обязателен для завершения обработки запроса. После его вызова уже нельзя вызвать write, а последующие вызовы end игнорируются.

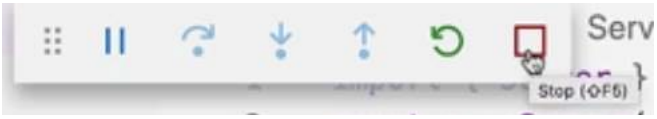


Рисунок 26. Выход из режима отладки

Для завершения отладки нажмём на Stop.

Итак, когда событие request возникает, система вызывает функцию-обработчик и передаёт ей объекты, описывающие запрос и ответ. Они являются экземплярами классов, соответственно, http.IncomingMessage и http.ServerResponse. Соответствующие формальные параметры часто называются req и res.

```
import http from 'node:http';
import { EventEmitter } from 'node:events';
import { Readable, Stream } from 'node:stream';

const s = http
.Server((req, res) => {
  console.log(req instanceof http.IncomingMessage); // true
  console.log(req instanceof EventEmitter);         // true
  console.log(req instanceof Readable);             // true
  console.log(res instanceof http.ServerResponse);  // true
  console.log(res instanceof EventEmitter);         // true
  console.log(res instanceof Stream);               // true
  res.end('ok');
})
.listen(() => console.log(s.address().port));
```

Листинг 10. Демонстрация того факта что req и res суть экземпляры EventEmitter и потоков

Как было показано в Главе 2, при использовании модуля **node:net** мы оперируем одним объектом (TCP-сокетом), который представляет собой полнодуплексный поток, служащий и для отправки данных серверу, и для ответа клиенту. При использовании **node:http** мы абстрагируемся от сокета и работаем с двумя более высокоуровневыми сущностями, двумя потоками, и через первый посылаем запрос серверу, а через второй получаем ответ.

Итак, мы отправляем данные, при необходимости разбивая их на порции-чанки, для чего сначала сколько угодно раз вызываем метод write, затем в конце один раз метод end:

```
const s = Server((req, res) => {
  res.write('OK');
  res.end('\n');
});
```

> **Примечание**
> В цепочку их объединить не получится: res.end возвращает ServerResponce, а res.write возвращает true или false (true, если все данные были успешно буферизованы, false если все или часть данных были поставлены в очередь в памяти пользователя, событие 'drain' испускается, когда буфер снова свободен).

Выдавать можно какое угодно содержимое, в этом случае проблемой клиента будет способ его обработки. Например, выдадим байты.

```
res.write(Buffer.from([1, 2, 33]));
res.end();
```

Это явная работа с двоичными данными, можно указать неявно `'\u0002\u0001\u0021'`

Также можно использовать изоморфный вариант с типизированным массивом:

```
const bytes = new Uint8Array(3);
bytes[0] = 2; bytes[1] = 1; bytes[2]=34;
res.write(bytes);
```

Чтобы получить эти данные, выполняем: `curl localhost:4321 -o file.dat` и смотрим содержимое файла в HEX-режиме.

Создать байты можно и напрямую из массива:

```
new Uint8Array([2, 1, 0x21]);
```

Если браузер получает содержимое, которое невозможно отобразить в окне, то предлагает его скачать. Мы можем явно указать такое поведение с помощью заголовка **Content-Disposition**, например 'Content-Disposition': 'attachment; filename="file.dat"'

Мы можем судить о чанках, если исследуем сервер на Node как он есть, без промежуточных серверов типа NGINX. Настройка chunked_transfer_encoding off; отключает порционную выдачу. Кроме того, curl собирает в одну строку всё что приходит в виде чанков. Поэтому нужно использовать клиент на Node и отслеживать событие data.

**Упражнение 5**

Сравните работу кода на PHP и на JavaScript/Node.js, запустив сервер и посетив маршрут в браузере

| | Браузер ждёт всю страницу (pending) | Браузер выводит по частям |
|---|---|---|
| PHP | <?php<br>echo '<h1>1</h2>';<br><br><br>sleep(5);<br>echo '<h2>2</h2>'; | <?php<br>echo '<h1>1</h2>';<br>ob_flush();<br>flush();<br>sleep(5);<br>echo '<h2>2</h2>'; |
| NODE | Server((req, res) => {<br>res.write('<h1>1</h1>');<br>((y, x = Date.now()) => { while (Date.now() - x < y) ;})(5000);<br>res.end('<h2>2</h2>');<br>})<br>.listen(4321); | Server((req, res) => {<br>res.write('<h1>h1</h1>');<br>setTimeout(() => {<br>res.end('<h2>h2</h2>');<br>}, 5000);<br>})<br>.listen(4321); |

Вернёмся к предыдущему примеру и заменим английское OK на русское Да

```
const s = Server((req, res) => {
```

```
    res.write('Да');
    res.end('\n');
});
```

Сервер отошлёт уже не 3, а 5 байтов (d0, 94, d0, b0, 0a), так как русские буквы здесь закодированы двумя байтами каждая.

Открыв соответствующий адрес в браузере, мы с высокой вероятностью увидим другие символы

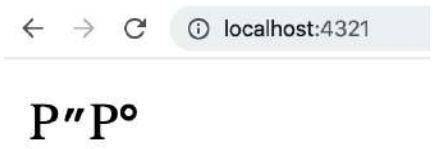


Рисунок 27. Неверное отображение кириллической буквы в браузере

Это происходит вследствие того, что браузер интерпретирует присланные байты не как UTF-8. Ему следует явно указать это делать. Для этого следует использовать метод writeHead:

```
import http from 'node:http';

http.Server((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });
    res.write('Да');
    res.end('\n');
})
.listen(4321);
```

Листинг 11. Сервер, отсылающий заголовок Content-Type и Transfer-Encoding

Мы указываем здесь MIME-тип text/plain и собственно кодировку. Вызывая этот метод, мы также указываем и код статуса ответа. Ничто не мешает указать код 404.

Поскольку при работе с чанками веб-сервер не отсылает вычисленное количество байтов ответа (отсылается заголовок Transfer-Encoding: chunked), мы можем явно указать клиенту, какое количество байт будет отослано.

```
const s = Server((req, res) => {
    res.writeHead(200, {
        'Content-Type': 'text/plain; charset=utf-8',
        'Content-Length': 5
    });
```

```
    res.write('Да');
    res.end('\n');
});
```

Наш веб-сервер отвечает одинаково на запросы, переданные любыми методами (method agnostic / «стиль Apache»). Если мы хотим по-разному обрабатывать запросы, отправленные разными методами, то должны прочитать значение свойства req.method и использовать структуру ветвления. Впрочем, считывание информации от клиента организовано по-разному. Данные передаются в заголовках, в виде глагола, как часть URL и в теле сообщения (Глава 1). Чтобы прочитать данные в виде заголовков, достаточно свойства req.headers:

Выполните запрос
curl https://fork.kodaktor.ru/getheaders -H "User-Agent:" -H "Accept:"

… и сервер вернёт заголовки, которые отослал клиент curl (в количестве двух: host и connection). Затем перейдите по https://fork.kodaktor.ru/getheaders в браузере и сравните результат (посмотрев его во вкладке Network).

Итак, чтобы отправить заголовки в ответе сервера, можно использовать метод объекта ответа (res) writeHead. В сочетании с оператором rest/spread мы можем удобно соединять вместе несколько определённых ранее заголовков (представлены как объекты):

```
const hu = {
    'Content-Type': 'text/html; charset=utf-8'
};

const cors = {
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Allow-Methods': 'GET,POST,PUT,DELETE,OPTIONS',
    'Access-Control-Allow-Headers': 'Content-Type,Accept,Access-Control-
Allow-Headers'
};

res.writeHead(200, { ...hu, ...cors });
```

В значении поля Content-Type мы указываем MIME-тип и кодировку (стандартом де-факто является utf-8).

| Расширение файла | Тип данных |
|---|---|
| css | application/x-pointplus |
| html | text/html |
| jpg | image/jpeg |
| js | text/javascript |
| png | image/png |
| txt | text/plain |
| zip | application/zip |

Рисунок 28. Различные MIME-типы

Cross-origin resource sharing (CORS; с англ. — «совместное использование ресурсов между разными источниками») — технология современных браузеров, которая позволяет предоставить веб-странице доступ к ресурсам другого домена. В примере выше указана звёздочка (**наименее** безопасный путь) – подключение разрешаются клиентскому сценарию, находящемуся на любом хосте.

Адрес https://kodaktor.ru/g/ko_resp не отдаёт заголовки CORS без дополнительного указания. Поэтому обратиться к нему с другого сервера клиентским сценарием невозможно без ошибки.

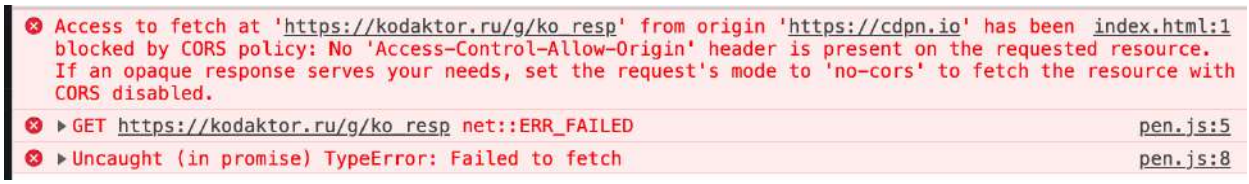


Рисунок 29. Ошибка при использовании политики CORS

Однако если использовать промежуточный сервер, который отдаёт эти заголовки и работает как прокси, то это ограничение можно обойти (https://cors-anywhere.herokuapp.com/https://kodaktor.ru/g/ko_resp).

### Простейшая маршрутизация

Наш веб-сервер также одинаково отвечает на все запросы к данному хосту независимо от структуры пути (route-agnostic). Если мы хотим учитывать информацию, которая находится в адресной строке, то должны анализировать свойство req.url (куда включается и строка запроса после вопросительного знака). Иными словами, модуль node:http не интерпретирует то, что выглядит как путь к файлу, в таком качестве, т.е. не поддерживает никакого встроенного механизма маршрутизации. Аналогично node:http не поддерживает интерпретацию QueryString.

Чтобы по адресу http://localhost:4321/doc/new/vasya-new.htm наше серверное приложение выдавало что-то содержательное, совершенно не нужно, чтобы существовал каталог doc и/или каталог new и тем более файл vasya-new.htm – это очень отличает серверное веб-приложение от обычного веб-сервера в стиле Apache, обслуживающего файлы. Нам всего лишь нужно предусмотреть реакцию на соответствующий маршрут:

```
if (req.url === '/ru') {
   res.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8'});
   res.write('<h1><i>Да</i></h1>\n');
} else if (req.url === '/doc/new/vasya-new.htm') {
   res.write('<h1><i>Vasya Web Page</i></h1>\n');
} else {
   res.write('OK\n');
}
res.end();
```

Листинг 12. Простейшая маршрутизация на основе анализа req.url

Для экспериментирования, для создания условий чуть более близких к реальности, чем localhost, можно использовать среду repl.it, которая объединяет онлайн-редактор, позволяющий осуществлять совместную работу над кодом, и сервер. Вышеприведённый пример доступен по адресу https://kodaktor.ru/g/first_node_demo

### Блокирующий код

При поступлении запроса коллбэк (req, res) => {… } помещается в очередь, и если для обработки запроса не нужно обращаться к внешним ресурсам, коллбэк будет выполнен максимально быстро. Разумеется, если в коде этого коллбэка будет нечто блокирующее вроде цикла while с огромным количеством итераций, то эффективность будет крайне низкой. Вся очередь будет вынуждена ждать, пока не выполнится блокирующий код. А это означает, что такой веб-сервер не будет обрабатывать запросы в период блокирования.

Пример приложения, иллюстрирующий обработку блокирующих запросов: https://kodaktor.ru/g/second_node_demo

Маршрут /timer? предполагает указание времени в миллисекундах после знака вопроса и выдаёт ответ спустя это время с помощью метода setTimeout.

```
1  const { Server } = require('http');
2
3  const CORS = {
4    'Access-Control-Allow-Origin': '*',
5    'Access-Control-Allow-Methods': 'GET,POST,DELETE',
6    'Access-Control-Allow-Headers': 'Content-Type, Access-Control-Allow-Headers'
7  };
8  const s = Server((req, res) => {
9    if (req.url === '/ru') {
10     res.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8', ...CORS });
11     res.write('<h1><i>Да</i></h1>\n');
12   } else if (req.url === '/doc/new/vasya-new.htm') {
13     res.write('<h1><i>Vasya Web Page</i></h1>\n');
14   } else if (req.url.startsWith('/timer')) {
15     const secs = req.url.substring(1 + req.url.indexOf('?'));
16     return setTimeout(() => res.end(`<h1><i>Seconds: ${secs}</i></h1>\n`), secs);
17   } else if (req.url.startsWith('/blocking')) {
18     const secs = req.url.substring(1 + req.url.indexOf('?'));
19     ((y = secs, x = Date.now()) => { while (Date.now() - x < y); })();
20     return res.end(`<h1><i>Seconds: ${secs}</i></h1>\n`);
21   } else if (req.url === '/download') {
22     res.writeHead(200, {…});
25     res.write('File\n');
26   } else {
27     res.write('OK\n');
28   }
29   res.end();
30 });
31 s.listen(4321);
```

Рисунок 30. Добавление маршрутов

После запуска приложения мы можем обратиться к этому маршруту: https://secondnodedemo.igossoudarev.repl.co/timer?5000

Функция setTimeout работает асинхронно, она откладывает выполнение своего коллбэка в очередь. На рисунке ниже подчёркнута инструкция return: если её удалить, то после завершения ветвления (синхронной структуры) управление перейдёт к 29-й строке, которая завершит работу потока. В этом случае асинхронная операция из 16-й строки не успеет выполниться.

```
14   } else if (req.url.startsWith('/timer')) {
15     const secs = req.url.substring(1 + req.url.indexOf('?'));
16     return setTimeout(() => res.end(`<h1><i>Seconds: ${secs}</i></h1>\n`), secs);
17   } else if (req.url.startsWith('/blocking')) {…} else if (req.url === '/download') {…}
     else {
27     res.write('OK\n');
28   }
29   res.end();
```

Рисунок 31. Предотвращение ошибки записи после окончания потока

Проведём эксперимент. В одном браузере отложим выполнение запроса на 5 секунд. И запустим также ещё один браузер, в котором тоже перейдём по тому же самому маршруту, но, с несколько другой задержкой, например, 8 секунд. Мы должны будем сначала увидеть результат выполнения запроса в первом браузере, где написано 5 секунд. Запускаем запрос в одном браузере, и почти сразу в другом. В обоих браузерах у нас происходит то, что выглядит как параллельная обработка запросов.

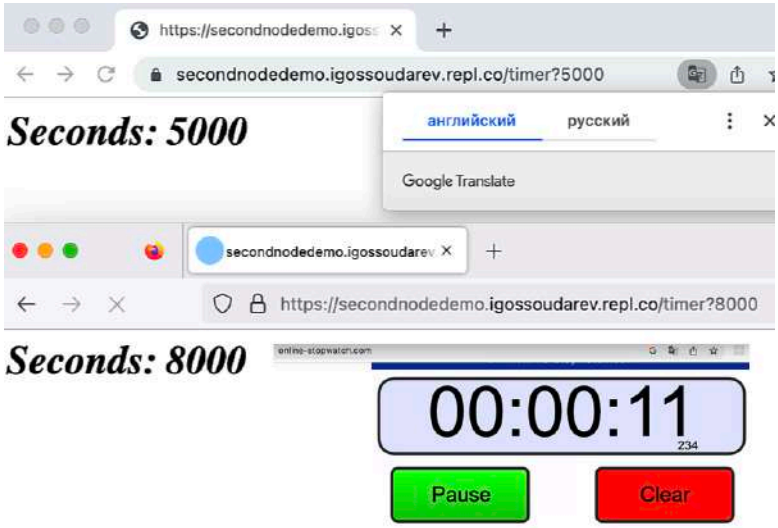


Рисунок 32. Маршрут с обработкой в очереди

Запрос, который занял меньшее количество времени, ожидаемо выполнился раньше, и это выглядело так, как если бы происходила параллельная обработка запросов. Такое возможно именно благодаря тому, что код работает асинхронно и работа функций, запущенных в ответ на поступление этого запроса не блокирует основной цикл событий. Теперь продемонстрируем противоположную ситуацию, которую следует избегать. По адресу https://kodaktor.ru/blocking находится пример кода, который блокирует цикл событий.

```
((y = 10000, x = Date.now()) => { while (Date.now() - x < y) ;})();
```

Если запустить такой код в браузере, то пользовательский интерфейс теряет отзывчивость. Например, при первом посещении страницы пользователь может выделить фрагмент этого кода, а после нажатия на кнопку уже нет: пользовательский интерфейс заблокирован.

JavaScript выполняется в одном потоке. И пока не закончился блокирующий код, ничто не может быть обработано, в том числе такие события, как выделение или щелчки и т.д. Воспользуемся блокирующим кодом, для того чтобы показать что происходит с серверным приложением в такой нежелательной ситуации.

Поместим блокирующий код в маршрут /blocking:

```
} else if (req.url.startsWith('/blocking')) {
  const secs = req.url.substring(1 + req.url.indexOf('?'));
  ((y = secs, x = Date.now()) => { while (Date.now() - x < y); })();
  return res.end(`<h1><i>Seconds: ${secs}</i></h1>\n`);
```

Рисунок 33. Блокирующий код в обработчике маршрута

Выдача ответа будет происходить только после выполнения этого блокирующего кода. Это будет означать, что если мы сделаем один блокирующий запрос, то пока он не выполнится уже невозможно будет обработать даже неблокирующий запрос.

Снова запустим запросы в двух браузерах. В одном блокирующий запрос на 10 секунд, а в другом неблокирующий запрос на 5 секунд. Если наши предположения верны, то с момента нажатия на Enter здесь и до момента завершения обработки запроса мы в сумме получим 15 секунд.

Постараемся максимально быстро начать оба запроса. В разделе Сеть инструментов разработчика мы видим сообщение «pending». После 10 секунд на обработку первого запроса ещё 5 секунд будет затрачено на второй запрос.

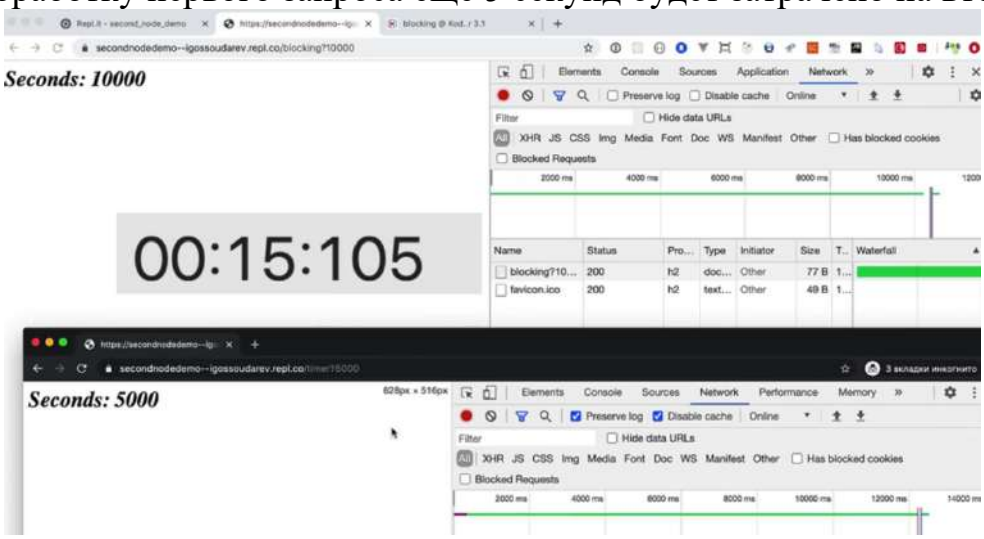


Рисунок 34. Последовательная обработка запросов

И теперь ещё раз здесь запустим неблокирующий запрос на 10 секунд, и во втором браузере опять неблокирующий запрос на 5 секунд, снова стараясь сделать это ближе к одновременности.

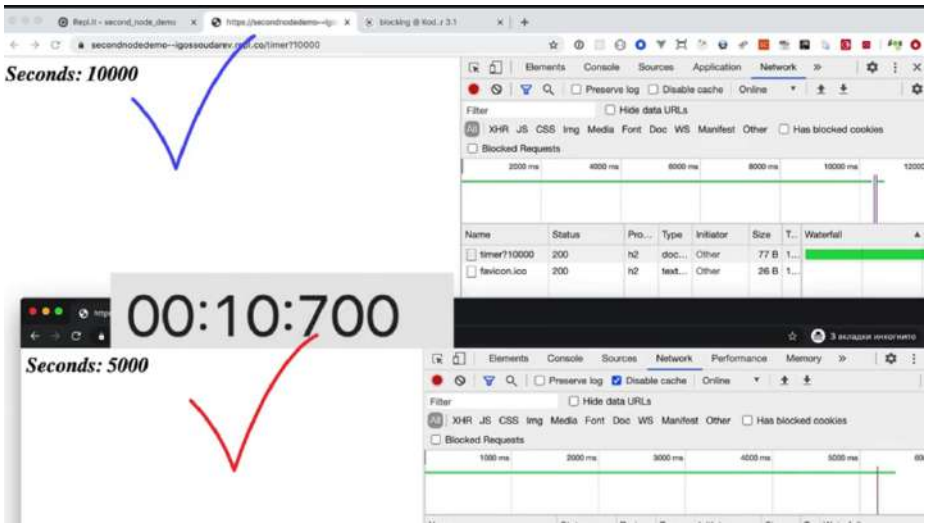

Рисунок 35. Иллюзия параллельной обработки запросов

Запрос на 5 секунд завершился до завершения запроса на 10 секунд. Если всё же необходимо выполнять действия, которые блокируют цикл событий, например, тяжёлые вычисления, то следует вынести такие действия в отдельный поток. На платформе Node.js для создания отдельных потоков существует модуль **node:worker_threads**. В браузере для этого используются веб-воркеры.

☞ Сделаем главный вывод: насколько это возможно, **не блокировать цикл событий в основном потоке исполнения**. В финале следующей главы мы вернёмся к теме эффективного запуска Node-приложения с использованием модуля cluster.

### Выдача статики

Мы уже увидели, что переход от режима route-agnostic к режиму дифференциации по маршруту связан с перебором условий по req.url. Для выдачи статических файлов тоже придётся «ветвить» код. Это осложняется тем, что нам могут быть заранее неизвестны имена файлов, представляющих собой статику. Для нативного модуля node:http существуют сторонние npm-пакеты, реализующие работу со статикой, например serve-static. Общая схема выглядит так: сперва мы явно определяем реакцию веб-сервера на конкретные маршруты типа /about/author. Если ни один из этих маршрутов не «сработал», т.е. клиент запрашивает нечто другое, то включается обработчик статики. Обработчик статики реагирует на маршруты, которые указывают на реально существующие статические файлы типа /public/license.txt. Если же запрос не может быть удовлетворён описанными способами, то должен включиться Последний Обработчик (finalhandler), который фактически выводит правильно оформленную 404-страницу.

```
import finalhandler from 'finalhandler';
```

Итак, finalhandler – это код, который срабатывает, когда ни один из предусмотренных вариантов затребуемого клиентом маршрута не соответствует реально затребованному. Пусть это функция, которая принимает на вход аргумент err. Если err ложностен (falsy = false, null, undefined и т.д.), то такая функция должна сработать как default case, т.е. выдать 404.

Например, написать **finalhandler**(req, res)() означает передать Последнему Обработчику undefined, т.е. ложностное значение, т.е. выдать 404.

Поэтому такой сервер всегда будет выдавать 404:

```
Server((req, res) => finalhandler(req, res)())
```

Пусть мы хотим, чтобы по запросу /hey осуществлялось приветствие, а во всех остальных случаях выдавалось 404.Тогда сервер будет выглядеть так:

```
Server((req, res) => {
  if (req.url === '/hey') res.end('hello!');
  finalhandler(req, res)();
})
```

Здесь нужно уточнить, что понимается под статическими страницами и статическим сайтом. В традиционном понимании это просто файлы HTML – разметка, даже без JavaScript; кроме того, можно говорить о страницах, которые полностью решают свою задачу будучи открытыми на локальном компьютере как file:///

Такие хостинги как GitHub Pages выдают статику, и, конечно же, эту задачу отлично решают CDN. В расширенном смысле статика – это и файлы JavaScript, и CSS, и шрифты и т.п. Для выдачи этой статики достаточно файл-сервинга уровня Apache/NGINX, нет нужды в шаблонизации, подключении к базе данных. При этом обеспечивается максимальная скорость получения информации, она фактически зависит только от Интернет-соединения.

Но следует отметить, что в современном понимании – более узком – статичность – это характеристика веб-страницы с точки зрения клиента или сервера. Если сервер считывает из файла или генерирует разметку и отправляет клиенту, тогда с точки зрения опыта клиента браузер получил статическую страницу – нужно лишь «выполнить» теги.

Но в SPA (Single Page Application) сервер генерирует «сырые» исходные данные, а браузер генерирует по ним разметку, и с точки зрения клиента это уже не статические, а динамические страницы.

Чтобы решать обусловленные динамической сущностью SPA проблемы, а также для ускорения загрузки, используются две основные техники:

- Server-Side Rendering (SSR): на каждый запрос сервер отсылает отрендеренную страницу, выполнив обращение к API или БД, так что браузеру остаётся только «выполнить» теги;
- Statice-Site Generation (SSG): осуществляется своего рода однократное построение, в результате чего создаётся множество страниц, для каждой из которых выполняются нужные запросы, и затем эти страницы с сервера будут в неизменном виде отдаваться как готовые файлы – до следующего момента построения.

В рамках этой главы мы говорим о статике в расширенном понимании – о ресурсах, подготовленных на стороне сервера и не меняющихся по ходу его работы, таких как HTML-файлы, таблицы стилей, изображения или шрифты.

Воссоздавать выдачу статики самостоятельно с нуля непродуктивно (можно в учебных целях наметить решение, например https://kodaktor.ru/serve.vm), лучше воспользоваться готовыми решениями. Компактный пример такого готового решения см. на https://kodaktor.ru/serve.static.vm – в этом варианте не используется пакет finalhandler.

```
import srv from 'serve-static';
const serve = srv('public', {
 'index': ['index.html', 'index.htm']
});

 Server((req, res) => {
  if (req.url === '/hello') res.writeHead(200) || res.end('Hello!');
  serve(req, res, () => res.writeHead(404) || res.end('not found'));
})
```

Выражение

```
  serve(req, res, () => res.writeHead(404) || res.end('not found'));
```

эквивалентно

```
  serve(req, res, finalhandler(req, res));
```

Функция serve – это функция от точки монтирования (например public) и некоторых настроек, включая индексный документ. В дальнейшем при работе этой функции то, что находится в req.url трактуется как путь относительно этой указанной точки монтирования.

Например, мы указали public – и у нас в папке приложения есть папка public, а в ней папка docs, в которой файл author.pdf. Тогда путь /docs/author.pdf как раз будет отсчитываться от этой папки-как-корня.

Кроме того, функция serve – это **middleware**, которая принимает объекты req, res и функцию next. Т.е. если файл не найден, она вызывается, и тем самым управление передаётся дальше по middleware-стеку. Более обстоятельно к middleware мы вернёмся в Главе 5.

**Упражнение 6**

| Выполняется в Linux или MacOS | |
|---|---|
| Команда | Пояснение |
| curl localhost:3336<br>yarn init -y<br>yarn add serve-static<br>curl -SL 'https://kodaktor.ru/j/serve.static.vm' -o 'index.js'<br>mkdir public<br>curl -SL 'https://kodaktor.ru/g/static.html' -o 'public/index.html'<br><br>node . &<br>curl localhost:3336<br><br>fg<br>И нажмите Ctrl C для остановки | Убедитесь что порт свободен<br>Инициализируйте проект<br>Добавьте зависимость<br>Скачайте сценарий<br>Создайте папку для статики<br>Скачайте в неё пример страницы<br><br>Запустите сценарий в фоне<br>Убедитесь в работе сервера<br><br>Перейдите в передний план<br>Остановите сценарий<br>Убедитесь в доступности порта |

Код примера доступен по адресу https://kodaktor.ru/g/third_node_demo

### Приём запроса с телом сообщения

Как мы уже убедились, «заголовочная» часть HTTP-запроса легко доступна через свойства req.headers, req.method и req.url. Рассмотрим чтение и отправку данных в теле сообщения.

Чтобы **прочитать** информацию из **тела сообщения** (т.е. части, отделённой от заголовков пустой строкой), потребуется больше усилий. Объект класса IncomingMessage, известный в нашем контексте как req, является push-потоком. Данные, которые передаются после пустой строки, должны считываться в рамках реагирования на событие data этого потока:

```
Server((req, res, b = []) => {
  req
  .on('data', d => b.push(d))
  .on('end', () => {
    res.writeHead(200, {
```

```
            'Content-Type': 'text/plain; charset=utf-8',
            'Content-Length': 3
        });
        res.write(Buffer.concat(b))
        res.end('\n');
    })

});
```

Модуль node:http активизирует поток req и генерирует событие data только если после пустой строки (т.е. двух переводов строк) что-то есть. Это что-то и будет телом запроса.

Чтобы **отправить** запрос с телом сообщения в curl используется параметр -d, при этом по умолчанию используется глагол POST. Можно указать и другой глагол, например -XPUT

```
curl localhost:4321 -d 'OK' -i
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 3
Date: Tue, 03 Mar 2020 01:53:17 GMT
Connection: keep-alive

OK
```

В соответствии с соглашениями HTTP, методом GET не следует отправлять запросы с телом, хотя технически ничто не мешает это сделать. У методов HTTP есть изначально закреплённая за ними семантика. Подход, при котором веб-сервер совершает различные действия в зависимости от глагола запроса (при этом каждый запрос полностью определяет то состояние, в которое должно прийти серверное приложение), называется REST.

Почему же первые три пути отправки данных оформлены как свойства, а четвёртый как поток? Очевидно, потому что заголовки, глагол, URL имеют ограниченный (небольшой) размер, а тело сообщения может оказаться очень объёмным.

Модуль node:http позволяет не только отвечать на запросы, но и посылать их. Вернёмся к серверу, который создавался нами ранее – см. Листинг 11.

Чтобы получить ровно те 5 байт, которые вышеприведённый сервер отсылает, в «чистом» виде, мы можем написать свой веб-клиент, который работает аналогично curl или браузеру.

```
import { get } from 'node:http';

get('http://localhost:4321', (r, b = []) => r
    .on('data', d => b.push(d))
    .on('end', () => console.log(Buffer.concat(b)))
);
```

Здесь используется метод get модуля node:http, отправляющий запрос одноимённым HTTP-глаголом. Он принимает адрес приложения и коллбэк, в который передаётся объект, реализующий потоковый интерфейс и интерфейс EventEmitter. Его можно подписать на событие data, возникающее при отправке чанка данных, и событие end, возникающее, когда поток передачи данных завершается. Данные накапливаются в массиве. Затем они соединяются в один буфер, и тогда выводятся. Результат: <Buffer d0 94 d0 b0 0a>

Итак, в этом коде r является экземпляром Readable из node:stream и, конечно, r является экземпляром EventEmitter из node:events; конструктором этого r является IncomingMessage

Использование в чистом виде http.get приводит к отправке на сервер следующих минимальных данных:

```
GET / HTTP/1.1
Host: 127.0.0.1:4321
Connection: close
```

Завершаются данные байтами 13, 10, 13, 10 – и в этом отличие отправки сообщения с телом, которое последует за двумя переводами строки. Чтобы «подглядеть», что именно отправляет наш клиент, можно запустить клиент на модуле node:net вместо node:http

```
import { Server } from 'node:net';

Server(sock => sock.on('data', d => console.dir(d)).end('HTTP/1.1 200\n\nДа'))
    .listen(4321);
```

Чтобы отправить сообщение с телом с помощью встроенного модуля net:http, нужно воспользоваться не методом get, а методом request

```
import { request } from 'node:http';

const options = {
  hostname: '77.234.202.142',
```

```
  port: 12345,
  path: '/',
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length': 9,
  },
};

request(options, res => res.on('data', d => console.log(d)))
.end('name=Ivan');
```

Листинг 13. Отправка серверу данных с телом сообщения методом http.request

Вот что будет отправлено на сервер:

```
POST / HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 9
Host: 127.0.0.1:4321
Connection: close

name=Ivan
```

Т.е. отправляется всё то же что было при GET, но
1. Добавляются заголовки запроса Content-Type и Content-Length
2. После байтов 13, 10, 13, 10 указывается сообщение, длина которого была ранее указана в Content-Length

Наконец, приведём сервер, который считает тело сообщения и отправляет его обратно

```
Server((req, res, b = []) => req
   .on('data', d => b.push(d))
   .on('end', () => {
      const sum = Buffer.concat(b);
      res.writeHead(200, {
         'Content-Type': 'text/plain; charset=utf-8',
         'Content-Length': 1 + sum.length
      });
      res.write(sum);
      res.end('\n');
```

```
    });
});
```

Листинг 14. Эхо-сервер, отправляющий клиенту полученное от него сообщение

Если клиент отправит меньше байт, чем указал в Content-Length, то событие **data** возникнет, а событие **end** не сможет быть обработано, запрос «повиснет». Если клиент отправит больше байт, то «лишние» байты будут проигнорированы.

Простой пример сервера, считывающего данные из тела запроса и показывающего их в своём ответе см. по адресу https://kodaktor.ru/g/fourth_node_demo или https://replit.com/@igossoudarev/fourthnodedemo#index.js

Обращайтесь к работающему приложению:

curl https://fourthnodedemo.igossoudarev.repl.co -d name=ilya

или с помощью httpie

http post https://fourthnodedemo.igossoudarev.repl.co name=ilya

**Упражнение 7**

Определите, сколько раз срабатывает событие data (т.е. сколько чанков данных придёт) при получении ответа на POST-запрос по адресу https://kodaktor.ru/api/chunks с передачей вашего имени (латиницей) в качестве значения параметра login

Мы не можем не отметить, что, во-первых, код отправки POST-запроса сложнее чем код отправки GET-запроса, а во-вторых, что этот код очень близок к ситуации Callback Hell из-за нескольких вложенных обратных вызовов. Долгое время с этим ничего нельзя было поделать, но после того как нам позволили использовать await на уровне модуля и после того как мы получили цикл for await of, можно осуществить небольшой рефакторинг этого кода в сторону большей линейности (хотя при этом он становится более императивным).

Промисифицируем метод http.get, учитывая тот факт, что его коллбэк *не следует соглашению Error First*

```
import { promisify } from 'node:util';
import { get } from 'node:https';

export default class HTTPS {
   static async get(myURL) {
```

```
    get[promisify.custom]= x => new Promise((res, rej) => get(x, res, rej));
    const g = promisify(get);
    const r = await g(myURL);

    let result = '';
    for await (const d of r) result += d;
    return result;
  }
}
```

Листинг 15. Промисификация потока (pro_get.mjs)

Теперь можно импортировать метод get и использовать его

```
import HTTPS from './pro_get.mjs';

console.log(await HTTPS.get('https://kodaktor.ru/api/chunks'));
```

Обратим внимание на то, что после промисификации уже нет необходимости подписываться на событие data. Цикл for await of делает всю работу по прочитыванию данных за нас. Его тело в режиме асинхронной итерации выполнится столько раз, сколько чанков было послано сервером.

Существует как минимум ещё один способ работать с потоком, не утруждая себя навешиванием коллбэков. Это, конечно же, конвейерное перенаправление данных из потока для чтения в поток для записи:

```
import { get } from 'node:https';

get('https://kodaktor.ru/api/chunks', r => r.pipe(process.stdout));
```

Использование встроенного модуля node:http может быть полезным при обучении, но на практике большинство разработчиков предпочитают использовать уже готовые проверенные решения. Мы обычно говорим о **библиотеке**, если имеем в виду полезный код (функции, классы, константы), который можем подключать к своему собственному проекту, при этом библиотека не предъявляет особых требований к стилю кодирования или структуре проекта. **Фреймворк** – это обычно реализация некоего паттерна типа MVC или MVVM, которая задаёт каркас – начальную структуру проекта и в предельных случаях превращается в фактическую заготовку кода с некоторыми оставленными промежутками, куда следует вписать свой конкретный содержательный код. Фреймворк может предоставлять инструмент генерации кода по некоторому формальному описанию. Может быть и так, что фреймворк «вырастает» из библиотек и определённого

«бойлерплейтного» кода (т.е. трафаретного кода, который приходится повторять из проекта в проект в почти неизменном виде).

В случае стандартного модуля node:http у нас есть функция Server или createServer, которую мы запускаем чтобы получить экземпляр сервера. Тем или иным способом с сервером связывается обработчик события request.

Но по мере продвижения вверх по абстракции мы хотим, чтобы программный код отражал сущности «реального мира», в том числе приложение. Т.е. если у нас есть приложение, то должен быть объект Application, сокращённо app, который содержит соответствующую функциональность, и в первую очередь маршруты. Ручная реализация маршрутизации подразумевает написание ветвления по req.url, мы же хотим быть избавленными от такой низкоуровневости. В экосистеме Node.js одной из первых попыток воплотить вышесказанное в жизнь был модуль connect, который можно рассматривать как своего рода протобиблиотеку.

### Модуль connect

В connect была реализована идея подключения middleware, о котором мы уже упоминали ранее и которое подробнее раскроем в следующей главе.

Вот пример минимального приложения, использующего connect:

```
import connect from 'connect';

const app = connect();
app
 .use((req, res, next) => res.end('Hello, world!'))
 .listen(4321);
```

Оно отталкивается от объекта app, который содержит метод use, с помощью которого к app подключаются функции с зафиксированной сигнатурой (req, res, next). Как мы помним, у обработчика события request модуля net:http сигнатура включает первые две сущности. Здесь добавляется ссылка next, которая символизирует следующую такую подключаемую функцию, и таким образом приложение собирается из последовательности подключаемых функций.

### Деплой приложения на Next.js/vercel

Такие модули, как connect, редко используются сами по себе – они входят в состав более сложных проектов. Рассмотрим в обзорном режиме очень большой и популярный проект – фреймворк **Next.js**.

Next.js создан как обёртка вокруг самой популярной фронтэнд-библиотеки для построения SPA-приложений из повторно используемых компонентов – React. Next подразумевает использование определённой структуры файлов и папок, это, как говорят, упрямый (opinionated) фреймворк. Наградой за согласие следовать его требованиям является очень простое

создание клиентской маршрутизации. Мы более подробно обсудим маршрутизацию в следующей главе.

Пока же поставим задачу: быстро создать и развернуть серверное приложение с маршрутом вида /api/mul/{N1}/{N2} – например https://kodaktor.ru/api/mul/11/3

> **Примечание**
> В данном случае {N1} и {N2} – выражения метаязыка, которые используются чтобы обозначить подстановочный характер величин N1 и N2. В шаблонизации фигурные скобки часто используются для этой цели, но возможны и другие варианты, например с двоеточием (/api/mul/:N1/:N2) в так называемом параметризованном маршруте, см. Глава 5

Для того, чтобы создать подобное приложение, нам нужно воспользоваться командой

npx create-next-app.

Или как вариант yarn create next-app.

В ответ на приглашение вводим имя проекта, после чего начинается загрузка зависимости, здесь перечислены react, react-dom. И собственно next.

Рисунок 36. Главные зависимости create-next-app

После завершения установки перед нами заготовка приложения.

Рисунок 37. Созданный проект next

В структуре есть папка зависимостей, а также папка pages.

Итак, по маршруту /login должен выдаваться логин в системе, и по серверному динамическому, так называемому, маршруту.

например /api/mul/3/11

…должен выдаваться JSON с результатом вычисления.

Чтобы реализовать выдачу login-а перейдём в pages и создадим здесь папку login, в ней файл index.js. Это *страница* в терминологии next, результат рендеринга компонента

Если бы мы написали export default () => 'admin';

то при этом создавалась бы лишняя разметка.

Нам же нужно, чтобы выдавалось только и исключительно одно слово, чтобы это был простой текст — поэтому придётся реализовать функцию getServerSideProps, которая при каждом запросе (каждом обращении к маршруту /login в нашем случае) выполняет указанные действия на стороне сервера, после чего страница со сформированным контентом отдаётся браузеру (это Server Side Rendering в понимании Next.js).

```
const plain = () => null;

export const getServerSideProps = async ({ res }) => {
  res.setHeader('Content-Type', 'text/plain');
  res.setHeader('author', 'admin');
  res.end('admin');
  return { props: {} };
}

export default plain;
```

Листинг 16 можно найти по адресу https://kodaktor.ru/vercel.next.js

Реализуем API. Next.js – это фуллстэк-фреймворк, он позволяет создавать и клиентскую и серверную части.

В разделе pages/api создадим папку mul, а в ней (это типичный прием создания маршрутов в Next) файл с неким специфическим именем, которое представляет собой основу для параметризации (здесь важны многоточие и квадратные скобки).

[...params].js

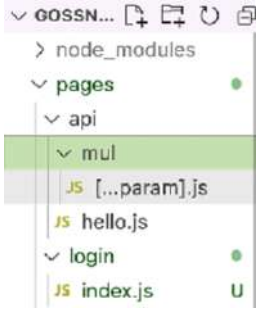


Рисунок 38. Папки и документы, превращаемые в маршруты

Тем, кто размещал лямбда-функции, например, на Яндекс-облаке или еще в каком-то подобном сервисе, это должно напомнить, конечно же создание обработчиков запроса (а тем, кто не размещал, будет предоставлена инструкция в Главе 6). Но в лямбдах, как правило, используется синхронный возврат значения для ответа (return), здесь же, как и в обычном http.Server мы используем res.end

```
const handler = (req, res) => {
   res.statusCode = 200;
   res.setHeader('Content-Type', 'application/json');
   // req url  /api/mul/11/3  => {result: 33}
   const requrl = req.url.split('/');
   res.end(JSON.stringify({ result: requrl[3] * requrl[4] }));
};

 export default handler;
```

Листинг 17 также находится по адресу https://kodaktor.ru/vercel.next.js

Так как Next использует библиотеку connect, для расширения функциональности таких обработчиков можно подключать middleware.

После создания двух вышеописанных файлов нужно будет запустить систему с помощью `npx next` или `yarn next`.

Это создаст локальный сервер с портом 3000 по умолчанию.

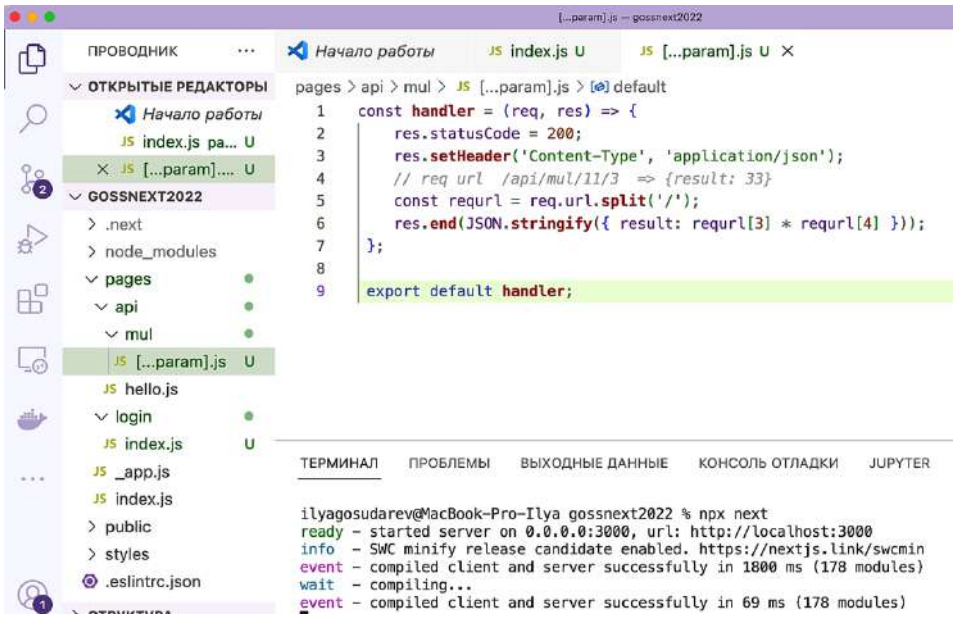


Рисунок 39. Запуск девелоперского сервера

При этом также создаётся папка .next, куда помещаются в числе прочего результаты рендеринга.

На локальном сервере будут доступны маршруты http://localhost:3000/login и http://localhost:3000/api/mul/3/11

Фреймворк Next.js был создан компанией Vercel, и она поддерживает хостинг таких приложений. Т.е. этот хостинг обеспечивает наличие Node.js, на каковой платформе выполняется серверный код. Нам нужно создать Git-репозиторий и поместить туда всё из проекта кроме node_modules.

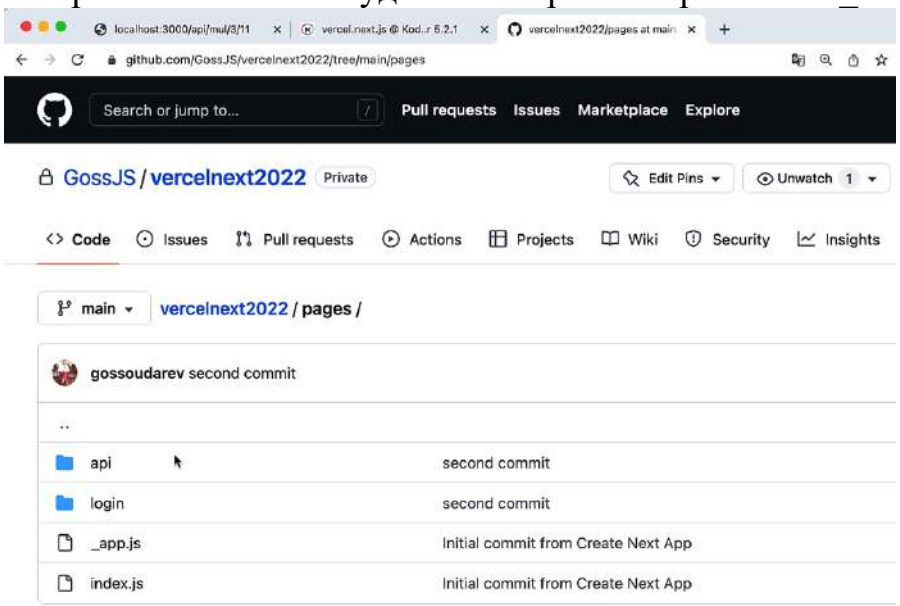


Рисунок 40. Репозиторий для импорта в vercel

Затем мы переходим на сайт vercel.com, выполняем sign-up (создаём новый аккаунт)

Оформляем авторизацию с помощью GitHub: нам нужно импортировать репозиторий с нашим проектом, выдав разрешение приложению vercel.com

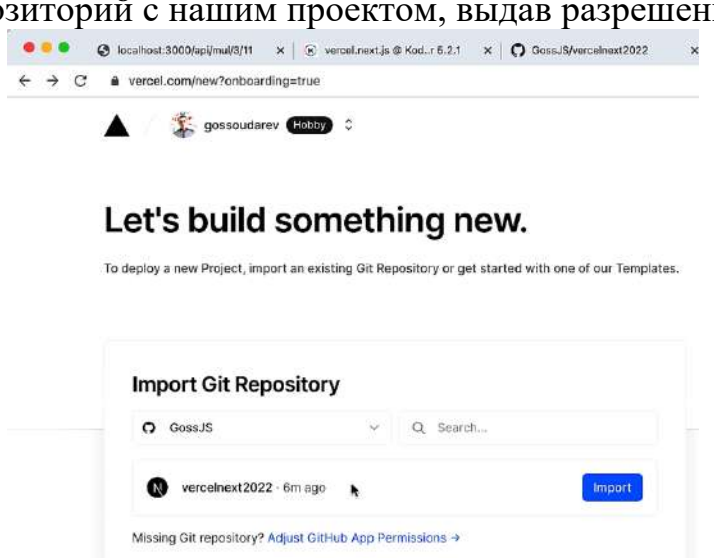


Рисунок 41. Процесс импорта репозитория в vercel

После импорта репозитория и нажатия кнопки Deploy происходит построение сайта, которое завершается выдачей URL

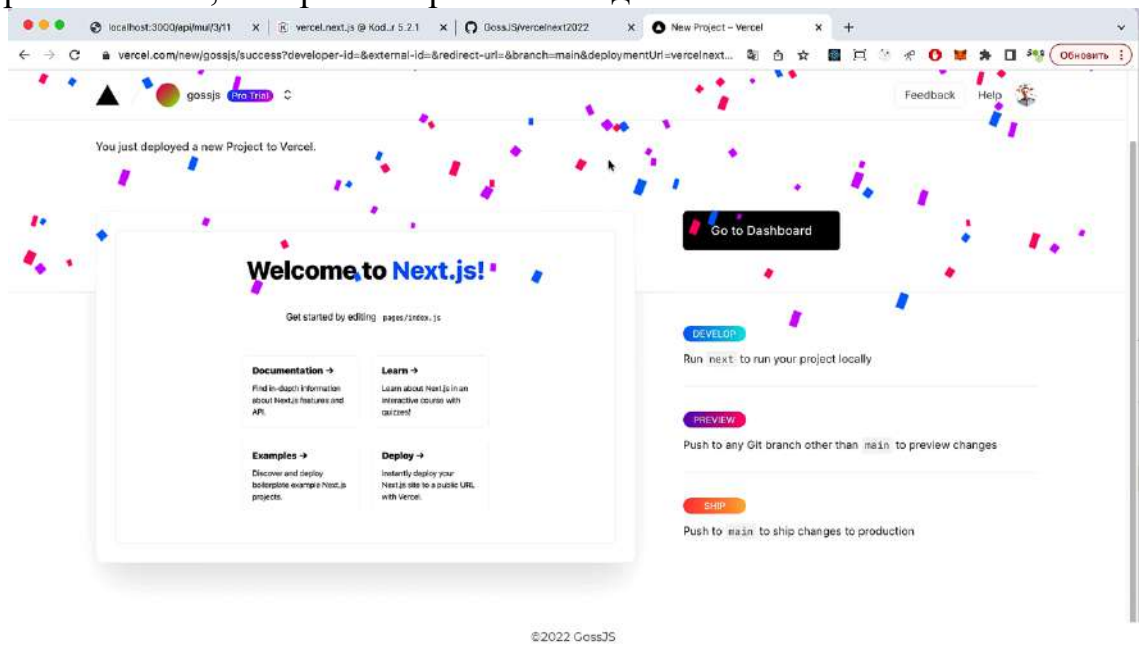


Рисунок 42. Успешный деплой проекта в vercel

Наконец, перейдя по выданному адресу, мы можем убедиться в работоспособности маршрутов:

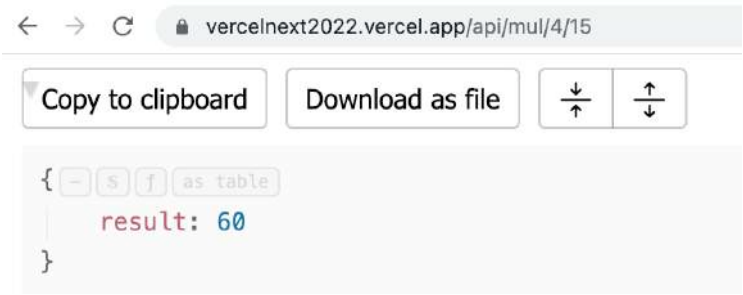


Рисунок 43. Успешная обработка маршрута приложением в vercel

**Примечание**

Глагол deploy передаётся на русском языке как «развернуть», «разместить» и даже «раскатать». Но разработчики часто говорят просто «задеплоить».

Таким образом мы рассмотрели способ создать приложение с размещением его на (пока) бесплатном ресурсе vercel.com. Эту инструкцию в видеоформате вы можете посмотреть по адресу https://kodaktor.ru/next.js.2022.mp4

### Использование ngrok

Если нет нужды в постоянном размещении проекта в публичном Интернете, то можно предоставить его в доступ со своего компьютера. Этот компьютер может иметь постоянный IP-адрес, но «светить» его без необходимости было нарушением требований безопасности. Удобнее скрыть свой реальный адрес за сервисом, который осуществляет проброс запросов и ответов через глобальный Интернет к локальному компьютеру и обратно; примером такого сервиса является ngrok. Его базовое использование крайне просто.

Сначала необходимо создать аккаунт на ngrok.com, затем загрузить и установить инструменты командной строки (CLI):

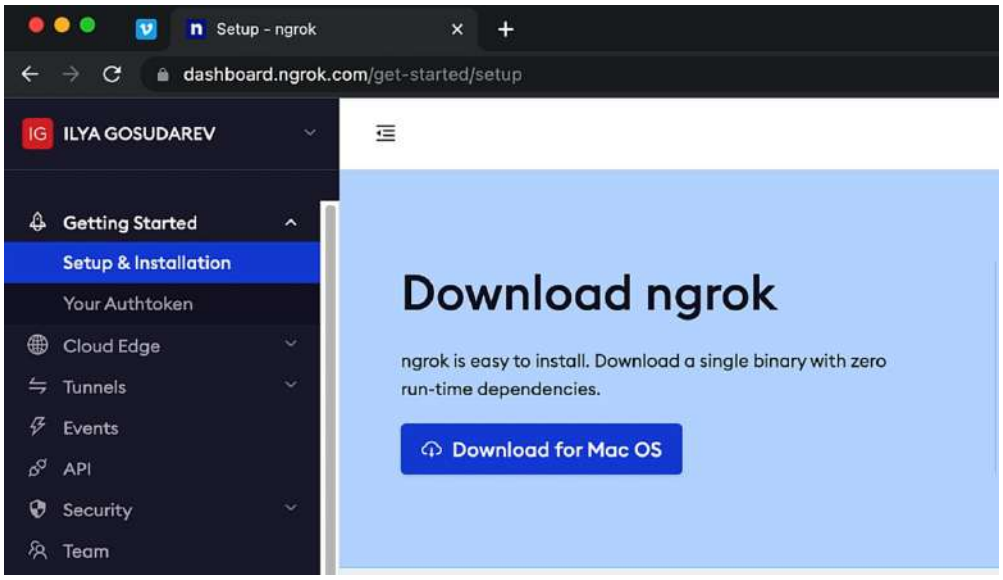


Рисунок 44. Страница загрузки ngrok

Затем после установки на свой компьютер нужно выполнить команду добавления конфигурации с токеном, который предложен новому пользователю в личном кабинете:

Рисунок 45. Ключ для конфигурации ngrok

И наконец, после того как мы запустили локальное приложение на 80 или например 443 порту, осталось выполнить команду вида ngrok http 80

Результатом этого станет через некоторое время генерация доменного имени с суффиксом .ngrok.io – бонусом будет автоматический доступ по https.

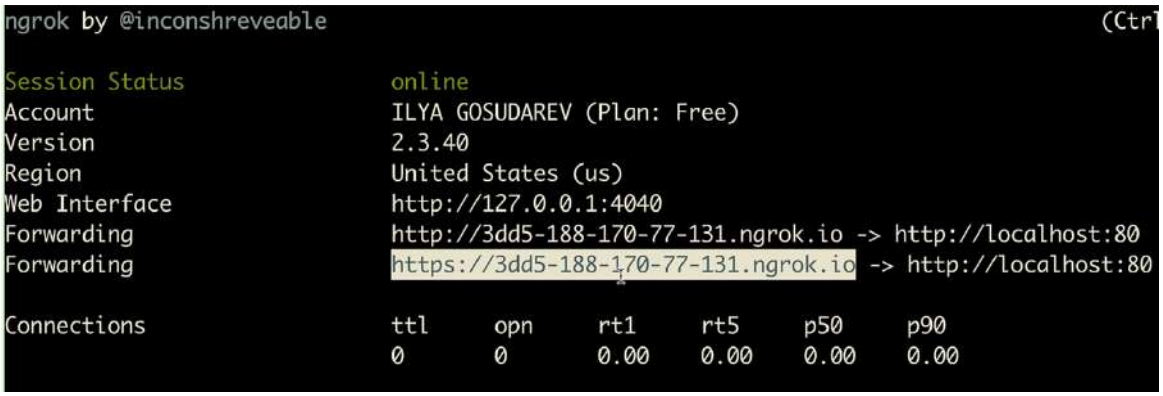


Рисунок 46. Результат запуска и пробрасывания приложения

Этот способ годится для того, чтобы запустить приложение на короткое время чтобы, например, сдать результат выполнения задания по онлайн-курсу.

Мы остановились в конце предыдущей главы на различиях между библиотеками и фреймворками. В этом смысле Express.js – нечто среднее между библиотекой и фреймворком, потому что с одной стороны у приложений, которые используют Express, наблюдается повторяющаяся структура, они тяготеют к монолитности, но с другой стороны ничто не мешает использовать Express как подключаемый компонент кода, с помощью которого создавать лямбды, т.е. даже не микросервисы, а наносервисы.

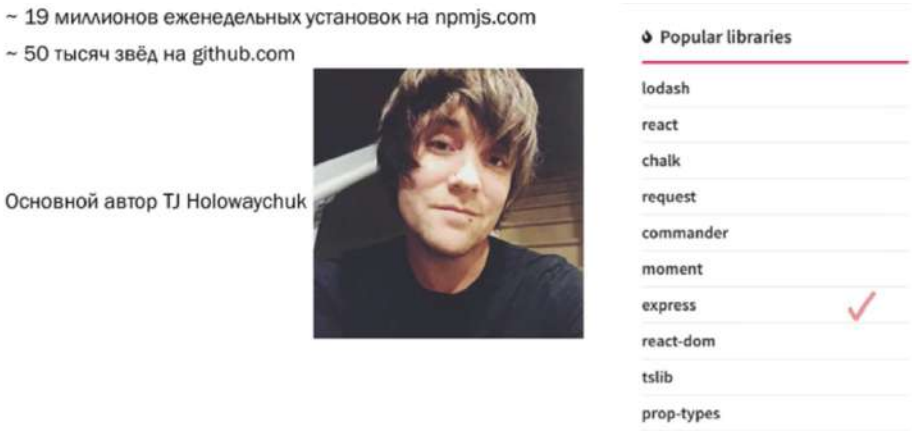


Рисунок 47. Создатель Express и характеристики Express

Выбор Express продиктован тем, что это один из самых популярных npm-пакетов за всю историю, стабильная и хорошо себя зарекомендовавшая библиотека. На основе Express созданы многие фреймворки и такие инструменты как webpack-dev-server. Нельзя не упомянуть библиотеку Koa, созданную авторами Express и реализующую похожую функциональность на основе усовершенствованного синтаксиса и улучшенной организации кода.

Вот некоторые примеры фреймворков, построенных на Express:

- Kraken
- LoopBack
- Sails
- Hydra-Express
- NestJS

Express сам по себе подходит для создания неспециализированных серверных приложений. Используя express, можно вручную реализовать и подобие NGINX, и REST API, и простую CMS, и микросервис. Однако для создания более «настоящих» микросервисов лучше использовать Hydra и Seneca, для REST API лучше использовать LoopBack, а для промышленного бэкэнда – NestJS.

На основе Express работают серверные веб-приложения, входящие в состав самых разных решений. Например, чтобы на платформе Яндекс.Облако – Функции создать лямбду со встроенной маршрутизацией, можно использовать Express – именно это мы и сделаем в Главе 6.

В целом в контексте обучения Express можно рассматривать как удачную **референсную** технологию: изучить её с максимальной детальностью, а затем использовать как базу для сравнения с другими технологиями и инструментами.

Для Express свойственен минимализм: в составе пакета всего 16 файлов, располагающихся в 3 папках. Общий объём собственно файлов с программным кодом составляет порядка 90 килобайт. Express имеет относительно небольшое число зависимостей, около 30. Кроме того:

- Express распространяется по лицензии MIT License
- Официальный сайт https://expressjs.com/
- Вдохновлён библиотекой маршрутизации Sinatra для Ruby

Важнейшей характеристикой Express является использование идеи промежуточного программного обеспечения или middleware. Также следует упомянуть, что для Express характерен функциональный стиль, и очень многое делается с помощью коллбэков. В частности, обработчики маршрутов являются именно коллбэками.

## Установка Express

Теперь рассмотрим создание приложения на основе библиотеки Express несколькими различными способами. Первый из них будет – создание с нуля. Откроем новое окно терминала на локальном компьютере и создадим новую папку. Инициализируем рабочее пространство для работы с JavaScript проектом, это выразится в создании файла package.json

Добавим в наш проект в качестве зависимости Express – `npm i express` или `yarn add express`.

Результатом этого действия окажется создание папки node_modules с загруженной библиотекой Express и появление записи в разделе dependencies. (Мы уже начали это делать в Главе 1 – см. рисунки 4 и 5.)

В папке node_modules располагаются файлы, которые составляют основу библиотеки Express, и самое главное, что здесь находится – это в папке lib.

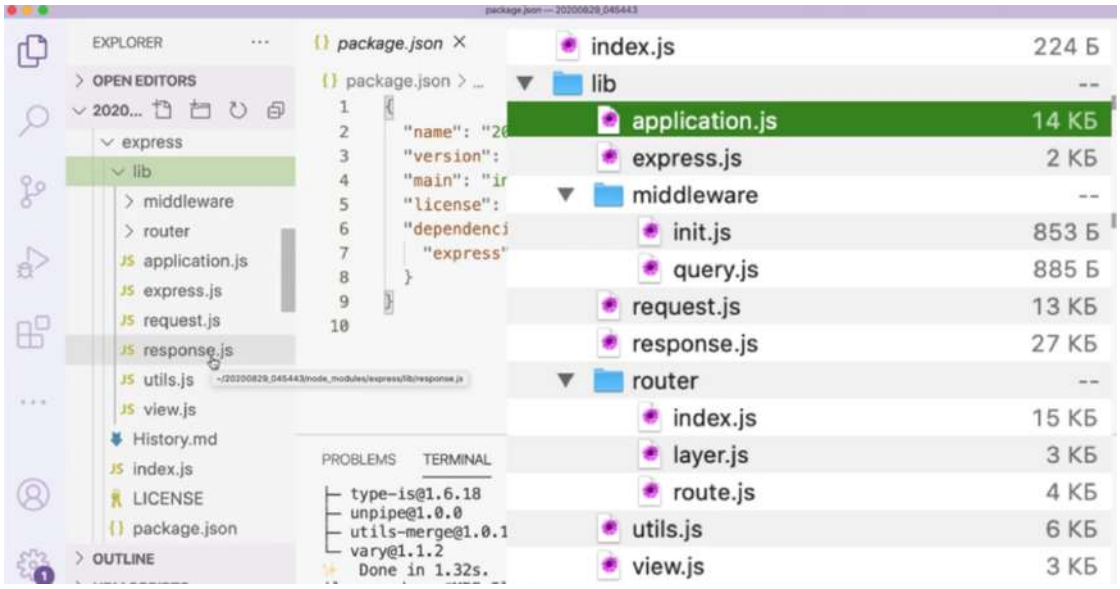


Рисунок 48. Файлы в составе Express

☞ Вы также можете загрузить весь код в виде архива:
wget kodaktor.ru/express/x.zip
Распаковав его, вы получите папку express. Всегда можно использовать этот код локально, не обращаясь к npm.

Когда мы будем затребовать библиотеку к себе в проект, то будет загружаться файл lib/express, и мы можем увидеть, что он экспортирует функцию, которая называется createApplication.

Рисунок 49. Экспорт функции createApplication

Результат работы этой функции будет Application, то есть express-приложение и он будет являться в первую очередь реализатором интерфейса EventEmitter.

Рисунок 50. Состав функции createApplication

Создадим файл index.js и в самом начале нашего файла поместим команду для импорта зависимости, которая экспортирует express. В результате вызова этой функции будет создаваться приложение, которое мы и обозначим как app. На всякий случай ещё раз отметим, что при желании работать с модулями ECMAScript, нам нужно в package.json создать ключ type и указать значение module. И в этом случае мы вместо require будем использовать import.

Рисунок 51. Варианты затребования / импорта Express

При использовании локального кода мы просто поменяем 'express' на './express'.

Итак, в результате выполнения функции x, которую, впрочем, можно было бы назвать как угодно по-другому, например просто express, мы получаем в app объект Applicaton, и в общем в простейшем случае построение приложения на основе express сводится к тому, чтобы всячески манипулировать этим объектом, в том числе с его помощью создавая маршруты.

Например, если мы хотим, чтобы наше приложение реагировало на корневой маршрут, откликаясь при этом на запрос, отправленный методом get, то мы должны вызвать одноименный метод приложения app и зарегистрировать коллбэк, который будет вызываться при наступлении события request с заданными параметрами, то есть по корневому маршруту и методом get. Этот коллбэк в простейшем случае от двух аргументов.

Здесь всё совпадает с работой, которую мы проделывали ранее со стандартным модулем node:http. Различия начнут проявляться чуть позже, в

частности для отправки запроса при помощи объекта response или res, в express обычно используется метод send, но можно использовать и метод end с тем же успехом.

Попробуем с помощью метода end отправить строку, которая выглядит как сериализация объекта. То есть на корневой маршрут наше приложение будет отвечать порцией данных в формате JSON.

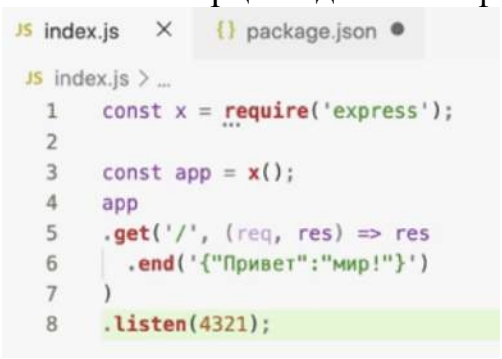


Рисунок 52. Первый маршрут

Если мы теперь перейдём в браузере по этому адресу, то увидим ответ, но он отображается не совсем так, как, по-видимому, нам бы хотелось, поскольку текст на кириллице браузер интерпретировал по-своему.

В ответных заголовках наше приложение, как выясняется, не указало кодировку.

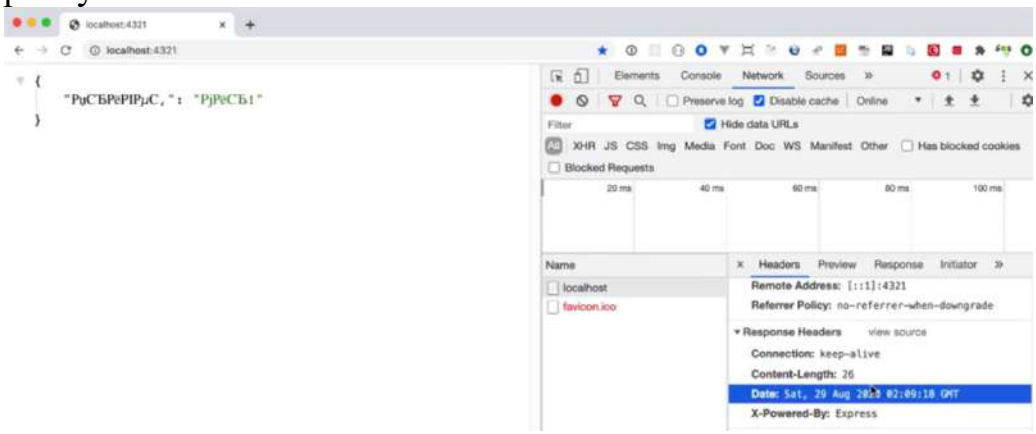


Рисунок 53. Результат res.end в браузере

Попробуем теперь заменить end на send. Обновив страницу, мы увидим, что русские буквы теперь отображаются корректно, но метод send автоматически послал для нас данные с MIME-типом text/html. Используем теперь метод **json** вместо end или send. Обновим страницу снова и увидим, что произошло экранирование кавычек, которые являются существенной частью формата json. Это опять помешало расширению браузера верно интерпретировать ответ. В данном случае дело в том, что мы посылаем строку. Уберём кавычки, обозначающие строковое значение и оставим только двойные кавычки внутри фигурных скобок:

```
.json({"Привет":"мир!"})
```

Обновив, увидим правильный результат.

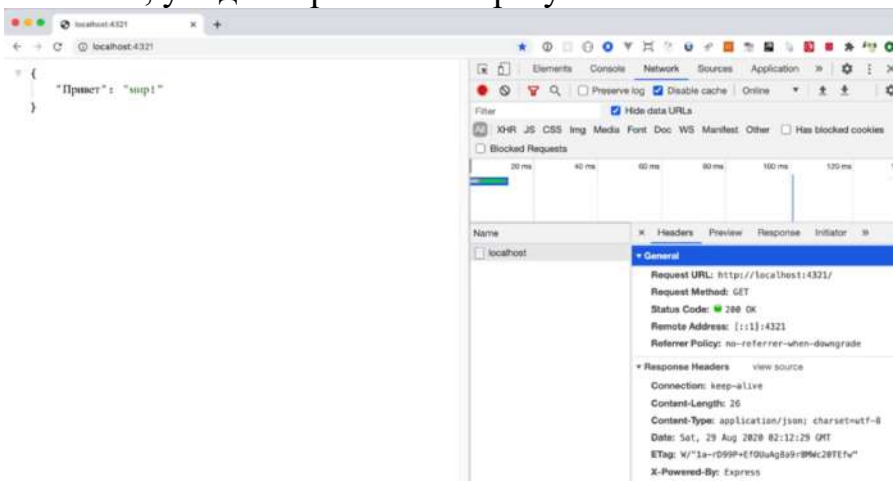


Рисунок 54. Результат res.send в браузере

Код, получающийся при создании приложения с нуля, можно найти по адресу: https://kodaktor.ru/x0. По этому адресу также приведён пример однострочника, то есть кода всего этого приложения, уложенного в одну строку, которую легко можно запустить в режиме REPL.

Эквивалентные однострочники для работы в REPL для быстрого запуска

| | |
|---|---|
| http | void http.**Server**((req, res) => res.**end**(req.url)).**listen**(4321); |
| express | void **require**('express')().**all**('*', r => r.res.**send**(r.url)).**listen**(4321); |

Ниже приведены выражения, которые приводят к созданию method-route-agnostic приложения:

- app.**all**('*', (req, res) => {})
- app.**use**((req, res) => {})
- http.**Server**((req, res) => {})

Поскольку в Главе 4 были приведены примеры кода на repl.it, то чтобы продолжить эту логику, преобразуем первый пример с http.Server в пример с Express:

https://replit.com/@igossoudarev/firstnodedemo#index.js

↓↓↓

https://replit.com/@igossoudarev/firstexpressdemo#index.js

(https://kodaktor.ru/g/first_express_demo)

Вместо writeHead у нас здесь будет set и с помощью вызова метода status мы установим код 200. Вместо сочетания методов write и end мы можем использовать send.

Переход по маршруту, который не был предусмотрен среди обрабатываемых, приводит к появлению кастомизируемого сообщения:

https://firstexpressdemo.igossoudarev.repl.co/xxx

***Не найдено! (/xxx)***

Рисунок 55. Ошибка пользователя

Это достигается способом реализации finalhandler, который является типичным для Express:

```
.use((req, res) => {
  res
  .status(404)
  .set({ 'Content-Type': 'text/html; charset=utf-8', ...CORS })
  .send(`<h1><i>Не найдено! </i>(${req.url})</h1>\n`);
})
```

Таким образом с помощью метода use подключается middleware-обработчик, который отвечает за ошибку пользователя (404).

Отметим также, что за загрузку файла (т.е. за выдачу заголовка Content-Disposition) в Express отвечает отдельный convenience-метод sendFile.

В конце предыдущей главы мы также рассматривали различные способы запуска приложения – локальный, в том числе с пробросом в Интернет с помощью ngrok, деплой на vercel.com. Здесь можно, помимо repl.it, упомянуть еще:

- Размещение небольших «демок» на runkit

- Запуск в составе Jupyter NoteBooks:

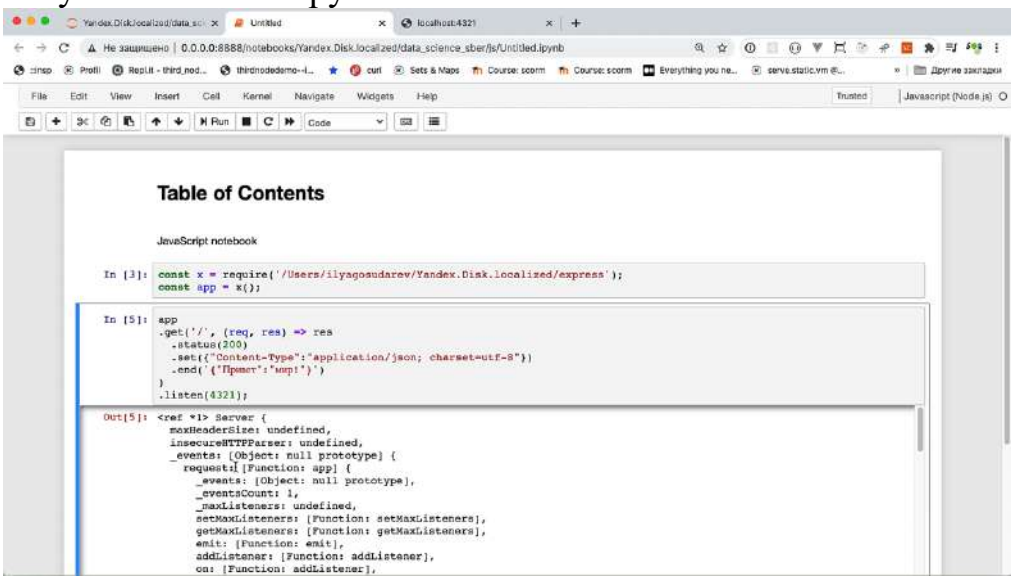


Рисунок 56. Jupyter NoteBooks с Node

- Использование Netlify – вы можете посмотреть инструкцию в видеоформате по адресу https://kodaktor.ru/netlify-x-2022.mp4
- Использование облака и serverless (см. Главу 6)

## Понятие middleware. Виды middleware и их стек

Трафаретный код базового приложения Express доступен по адресу: kodaktor.ru/x. Также для удобства обучающихся был создан сервер https://demoapp.kodaktor.ru на котором это приложение с некоторыми небольшими дополнениями (см. далее middleware и маршрутизацию) запущено.

index.js > ...

```
1   const [{ Server: h1 }, x] = [require('http'), require('express')];
2
3   const Router = x.Router();
4   const PORT = 4321;
5   const { log } = console;
6   const hu = { 'Content-Type': 'text/html; charset=utf-8' };
7   const app = x();
8   Router
9     .route('/')
10    .get(r => r.res.end('Привет мир!'));
11  app
12    .use((r, rs, n) => rs.status(200).set(hu) && n())
```

```
13    .use(x.static('.'))
14    .use('/', Router)
15    .use(({ res: r }) => r.status(404).set(hu).send('Пока нет!'))
16    .use((e, r, rs, n) => rs.status(500).set(hu).send(`Ошибка: ${e}`))
17    /* .set('view engine', 'pug') */
18    .set('x-powered-by', false);
19  module.exports = h1(app)
20    .listen(process.env.PORT || PORT, () => log(process.pid));
```

Рисунок 57. Базовый код Express-приложения (kodaktor.ru/x)

☞ Сервер https://demoapp.kodaktor.ru/ доступен если запущен, а запустить его может любой перейдя по адресу https://kodaktor.ru/api/4321/on

В первой строке подключаются модуль node:http и библиотека express. При этом функция Server оперативно переименуется здесь в модуле node:http и превращается в h1. В 19-й строке App, то есть результат вызова того что импортируется из Express, передаётся этой самой функции h1. Это говорит о том, что App является функцией, которая обрабатывает запросы. При том, что это ещё и объект у которого есть определённые методы, он может выполнять роль особой промежуточной функции – middleware.

App играет роль слушателя, который может прослушивать порты и таким образом выглядит как сервер, и отсюда мы можем сделать вывод, что функциональность этого объекта App или Application очень разносторонняя.

В дальнейшем такая передача App на вход функции, которую мы извлекаем из модуля node:http понадобится для того, чтобы подключать сертификаты SSL и реализовывать взаимодействие по протоколу HTTPS (Глава 7). Мы видим в трафаретном коде много вхождений слова use. При этом в 15-й строке находится уже знакомый обработчик ошибки пользователя (Ненайденный маршрут). Отметим пока, что в следующей строке подключается обработчик ошибки приложения, функция от 4-х аргументов – таково соглашение Express.

**Упражнение 8**

Запуск базового приложения на основе express на локальном компьютере

1. Создайте новую папку и перейдите в неё
2. Выполните в ней команду
   `npm init -y`
3. Установите библиотеку express командой
   `npm i express`
4. Скачайте базовый код командой
   `wget kodaktor.ru/g/x.mjs`
5. Запустите сценарий на выполнение командой

```
node --watch x.mjs
```

6. Убедитесь, что сервер отвечает на клиентский запрос:
```
curl localhost:4321
```

Завершив упражнение, вы должны будете увидеть ответ «Привет, мир!»

Мы уже упоминали термин middleware в связи с понятием finalhandler и модулем connect. В Express подключение middleware радикально расширяет функциональность app.

Рисунок 58. Последовательное подключение middleware

С помощью метода **use** мы подключаем middleware, к числу которых относится и роутер-маршрутизатор, вынесенный в отдельную функцию.

Рисунок 59. Части обработчика маршрута с точки зрения middleware

Понятие middleware не является чем-то особенно новым, оно используется в сфере веб-технологий достаточно давно. В русском языке используются самые разные слова для того чтобы обозначить это понятие:

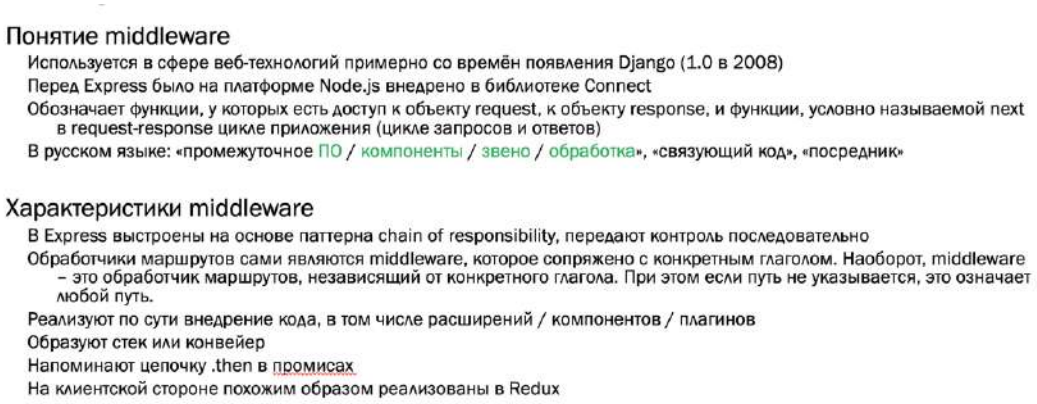


Рисунок 60. Характеристики понятия middleware

Выше уже упоминался middleware-обработчик ошибки пользователя. Если ни один из подключенных обработчиков возможных маршрутов, предусмотренных в приложении, не мог сработать, то управление передаётся именно ему.

Рассмотрим пример: маршрут /first. Мы можем, находясь в обработчике маршрута, вызвать ошибочную ситуацию и передать управление обработчику. Это может быть реализовано, если мы добавим в число аргументов обработчика маршрута next-ссылку.

Если вызвать это next, то управление будет передано (в чём и состоит смысл слова next), **следующему** middleware в так называемом стеке, то есть в последовательности подключённых функций промежуточного звена.

```
.get('/first', (req, res, next) => {
  if (req.query.error == 'yes') next();
  res.send('Работает!');
})
.use(({ res: r }) => r.status(404).set(hu).send('Пока нет!'))
.use((e, r, rs, n) => rs.status(500).set(hu).send(`Ошибка: ${e}`))
```

Рисунок 61. Передача управление следующему middleware

Если вызову next передаётся, например, как на рисунке выше, undefined, синонимом чего является непередача вообще какого-либо значения, то вызывается Последний Обработчик (ошибки пользователя). А если передать какую-то кастомную ошибку, то вызван будет обработчик внутренней ошибки приложения.

Это middleware-коллбэк, который принимает по соглашению 4 аргумента: 1) описание ошибки 2) запрос 3) ответ и 4) формальная ссылка на функцию промежуточного звена, которой уже нигде дальше не может быть, потому что это Самый Последний обработчик.

Рисунок 62. Уточнение поведения обработчика ошибки

Так что, если обратиться к маршруту /first на рисунке выше, мы получим соответствующее сообщение. Аналогичного эффекта можно было бы добиться, просто написав слово throw. И именно такой подход используется, например, во фреймворке Koa, где архитектура промежуточного звена была подвергнута сильному пересмотру и в целом выглядит очень целостно и более очевидно, чем в Express. Тем не менее, эти соглашения в рамках Express следует соблюдать.

Мы можем получать сообщение об ошибке, о том что невозможно послать заголовки, после того как они посланы (Error [ERR_HTTP_HEADERS_SENT]: Cannot set headers after they are sent to the client). Это может происходить, если забыть написать return в функции, которая работает с res, или если вслед за res.write попытаться выполнить res.send.

По маршруту /first в версии приложения по адресу demoapp.kodaktor.ru выводится стек подключенных middleware – это сделано следующим образом. Через объект request доступно приложение: req.app. У него есть служебное свойство router. И в свою очередь у него имеется свойство stack, которое хранит массив подключённых middleware. Поскольку это массив, то мы можем в цикле для каждой функции посмотреть её имя в консоли. req.app._router.stack

Вызвав маршрут увидим список функций. Здесь в том числе есть и router и servestatic, которые мы подключаем для того, чтобы обслуживать статические файлы.

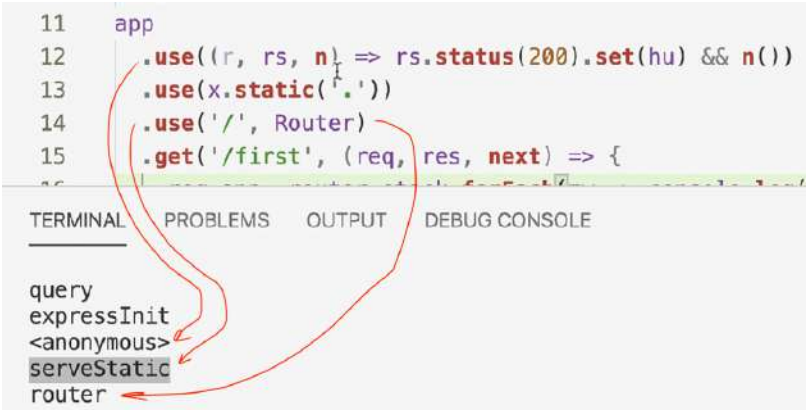


Рисунок 63. Стек подключённых middleware

Что касается анонимных middleware, то функции, у которых нет имени, соответствующим образом и обозначаются в этом списке (<anonymous>). Если мы хотим обрести больший контроль над происходящим, то мы можем именовать эти функции.

Например, мы можем вынести функцию mw0 отдельно и теперь при переходе по этому маршруту получим именно её. Таким образом, мы можем сделать вывод о том, в каком порядке они были подключены. Сперва некое встроенное middleware, которое позволяет обрабатывать query, middleware которое инициализирует express и инициализирует App, затем наше middleware mw0, которое мы только что создали, затем serve static, обслуживание статических файлов.

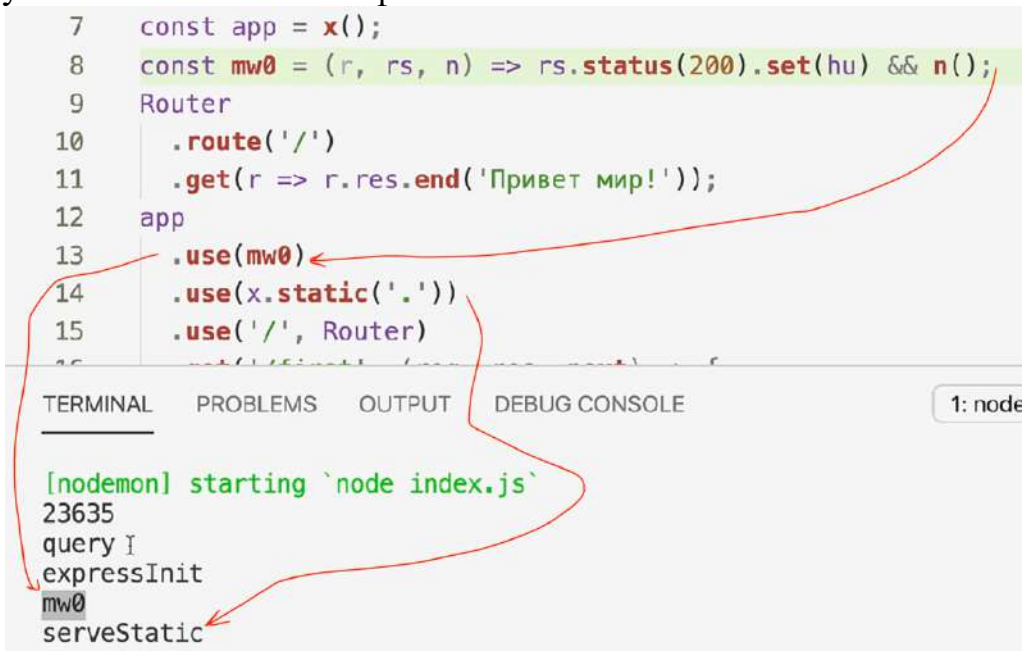


Рисунок 64. Именованная функция middleware

Обычно говорится, что middleware реализует паттерн Chain of Responsibility (цепочка ответственностей). Они последовательно передают контроль над процессом обработки запроса. Обработчики маршрутов, которые подключаются с помощью таких методов как get, post и других, соответствующих методам HTTP, сами являются middleware, и наоборот. Middleware можно рассматривать как обобщённый обработчик маршрутов, не зависящий от конкретного глагола. Если при этом ещё и не указывается путь, то это означает любой путь.

```
app.all('*', (req, res) => {});

app.use((req, res) => {});
```

Рисунок 65. Эквивалентные универсальные обработчики

Также можно сказать, что middleware реализует внедрение кода. И в роли этого кода могут выступать расширения, компоненты, плагины. Например,

обслуживание статики – это как раз средство расширения базовой функциональности Express.

Похожим образом обрабатываемый объект передаётся сверху вниз, позволяя создавать некое общее информационное пространство. Продемонстрируем это ещё одним примером.

Создадим middleware, которое определяет свойство req.working. Напишем use (function workingSetter), который будет создавать соответствующее значение и передавать управление дальше по конвейеру обработчиков промежуточного звена.

```
.use(mw0)
.use(function workingSetter(req, res, next) {req.working = 'Работает, ура!'; next();})
.use(x.static('.'))
.use('/', Router)
.get('/first', (req, res, next) => {
  req.app._router.stack.forEach(mw => console.log(mw.name))
  if (req.query.error == 'yes') return next('кастомная ошибка');
  res.send(req.working);
```

Рисунок 66. Передача данных через middleware

Если забыть вызвать обработчик next и обратиться к маршруту, то браузер будет ожидать ответа (pending) – это pending будет «висеть» теперь до тех пор, пока не истечёт время ожидания при запросе и браузер напишет о том, что это время превышено.

## Маршрутизация. Создание express.Router и дерева маршрутов

Раскроем более подробно понятие маршрутизации. **Маршрут** (рут, **route**) – это метод HTTP, сопряженный с шаблоном соответствия URL-адресам; каждый маршрут связан с блоком кода.

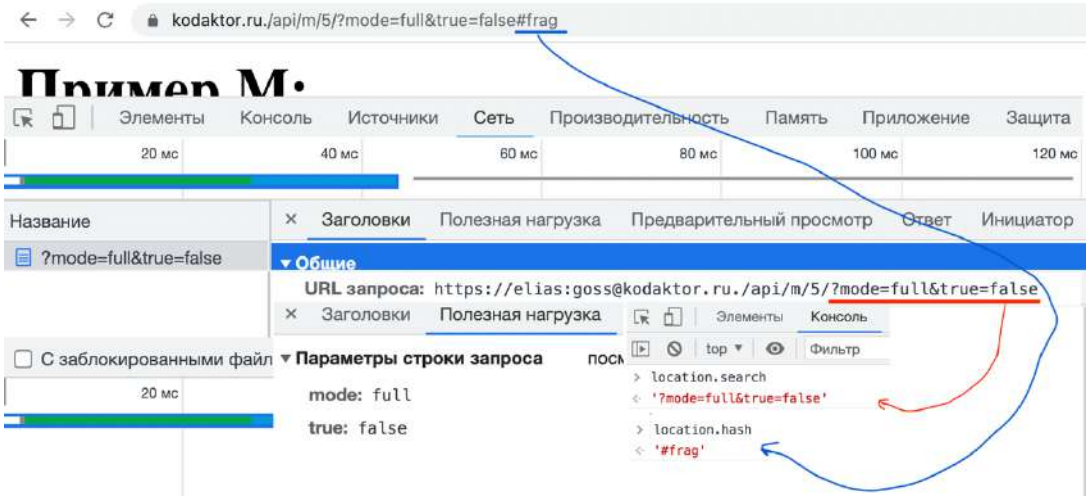


Рисунок 67. URL запроса

Близкими терминами являются путь или path, endpoint, но соотношение между маршрутом и endpoint это скорее соотношение общего и частного, поскольку конкретным случаем маршрута является endpoint, то есть такой вариант маршрута, который уже никуда не ветвится и представляет собой

окончательный вариант маршрута. С понятием маршрута также может связываться URL в узком смысле, то есть та часть, которая указана после всего относящегося к хосту: первый слэш и далее ещё какие-то части. Кроме того, с понятием маршрута также связывается понятие контроллера.

Маршруты можно задавать различными способами, например на Рисунке 68 представлен вариант, который вдохновлял создателей библиотеки Express. Описание маршрутов очень минималистичное и имеющее декларативный оттенок.

```
 GNU nano 2.5.3                File: myapp.rb

# myapp.rb
require 'sinatra'

get '/' do
  'Hello world!'
end

get '/lala' do
  'sing with me!'
end
```

Рисунок 68. Декларативная маршрутизация в Sinatra

Мы ранее встречались с подчёркнуто императивным вариантом в модуле node:http, когда приходилось писать ветвления для того чтобы дифференцировать одни URL от других, в Express этот способ более функциональный.

Маршрутизация может быть также и клиентской, и тем, кто работает на фронтэнде, известно о различных вариантах реализации клиентского роутера, который сопоставляет маршрутам компоненты. Ниже на рисунке представлен пример из библиотеки React, это браузерный роутер который основывается на History API.

```
import { render as r } from 'react-dom';
import { BrowserRouter } from 'react-router-dom';

r(
  <BrowserRouter>
    ...
  </BrowserRouter>,
  document.querySelector('.cont')
);
```

```
<Route exact path={HOME_PAGE_ROUTE} component={Home} />
<Route exact path={MYCOMP_PAGE_ROUTE} render={() => <MyComp title="hello" text="hey" />} />
```

Рисунок 69. Декларативная маршрутизация в React Router

При запуске в режиме разработки такого приложения файлы обслуживает webpack DevServer, который построен на Express. Когда приложение деплоится, можно использовать любое серверное программное обеспечение, которое будет обслуживать нужные файлы в статике, JavaScript сценарии, каскадные стили и, конечно, страницу index html, которая подгружается по практически всем возможным маршрутам, независимо от того, что запрашивается:

```
app
.use(x.static('./build'))
.get('*', r => r.res.sendFile('public/index.html'), { root: '.'})
```

В качестве альтернативы можно использовать обычные традиционные веб-сервера, выдающие статику с прописанными правилами рерайтинга URL, что позволяет достигать такого же эффекта.

В то же время если на хостинге (например, на GitHub pages) нет возможности запустить серверное приложение, то нам помогает HashRouter, который позволяет имитировать эту деятельность, и когда мы открываем такую страницу, то мы можем менять часть после хэша. На это будет уже реагировать чисто клиентский сценарий и запроса к серверу не происходит. При этом эксплуатируется тот факт, что фрагмент, часть URL после хэша, клиенты http по соглашению не отправляют на сервер (https://www.rfc-editor.org/rfc/rfc3986#section-3.5 ).

По нижеприведённым ссылкам вы можете увидеть базовую реализацию клиентского роутера на HashRouter:

https://kodaktor.ru/react_router

https://kodaktor.ru/g/react_router#/first

Обычно существенной задачей при разработке серверного приложения является проектирование и разработка дерева маршрутов.

Рисунок 70. Дерево маршрутов

Например, когда мы пишем localhost в адресной строке и нажимаем Enter или kodaktor.ru и нажимаем Enter, то мы переходим по корневому маршруту. Здесь представлена схема маршрутов, которую мы в качестве демонстрации реализуем в пределах этого урока. Мы будем использовать встроенный в Express роутер, который является одним из видов middleware.

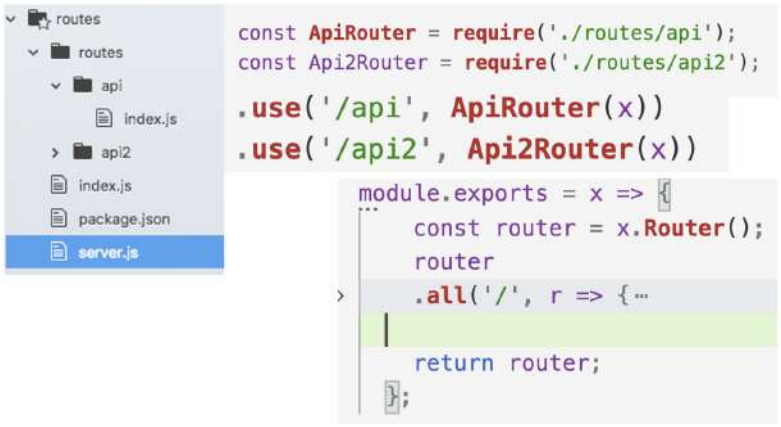


Рисунок 71. Маршрутизатор в отдельном файле

Задача заключается в том, чтобы в папках, в основной папке приложения, создать сценарии которые реализуют эти отдельные маршрутизаторы, или если угодно контроллеры, а затем затребовать/импортировать их и подключить к основному приложению App с помощью метода use. Каждый отдельный файл отвечающий за создание маршрутизатора будет строиться примерно по одной и той же схеме, которая представлена здесь на слайде.

Мы можем группировать похожие маршруты в пределах роутера, а также, разумеется, мы можем группировать вариации одного и того же маршрута, которые отличаются теми или иными конкретными данными, числами, строками и т.д., используя для этого параметризацию. В маршруте конкретные значения заменяются на места для подстановки или плейсхолдеры (мы уже сталкивались с этим в Главе 4).

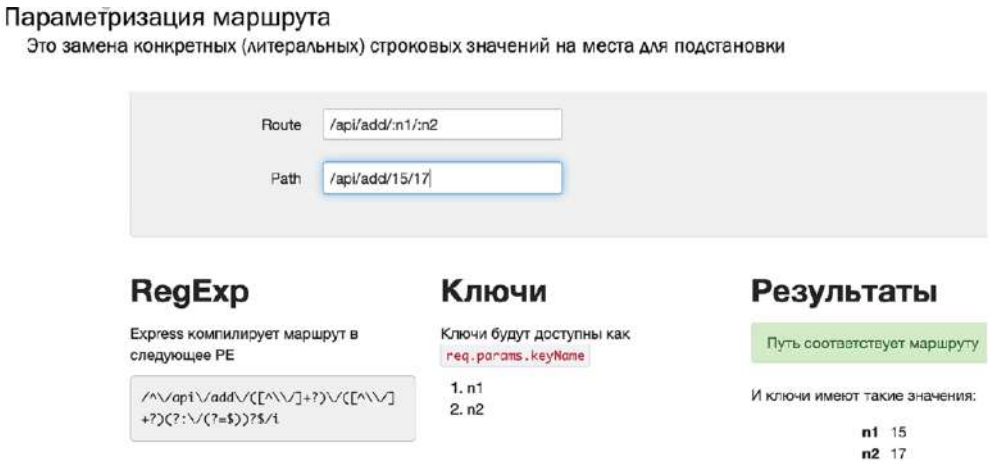


Рисунок 72. Регулярное выражение для компиляции маршрутов

Это своего рода встроенная шаблонизация маршрутов, и мы получаем единое выражение, которое компилируется в регулярное выражение библиотекой Express, и на соответствие которому подставляются уже конкретные варианты маршрутов, приходящие в запросах.

Создавая дерево маршрутов, следует иметь в виду, что в общем случае не очень хорошей практикой является писать всё, что связано с маршрутами, в основном файле приложения. Основной файл должен импортировать функциональность, которая реализована в отдельных модулях. Мы создадим папки api и api2. В папке api создадим файл под названием index.js.

И экспортируем в нём, используя способ, который сочетается с использованием require, функцию, которая по аргументу x будет возвращать экземпляр роутера. Роутер это в действительности один из вариантов middleware, который можно рассматривать как некоторую упрощённую сокращённую версию основного приложения. Класс Router в виде функции с соответствующим именем доступен через имя express, или как в нашем случае x.

Когда мы вызываем эту функцию, мы получаем этот экземпляр роутера, и после всех манипуляций его необходимо вернуть.

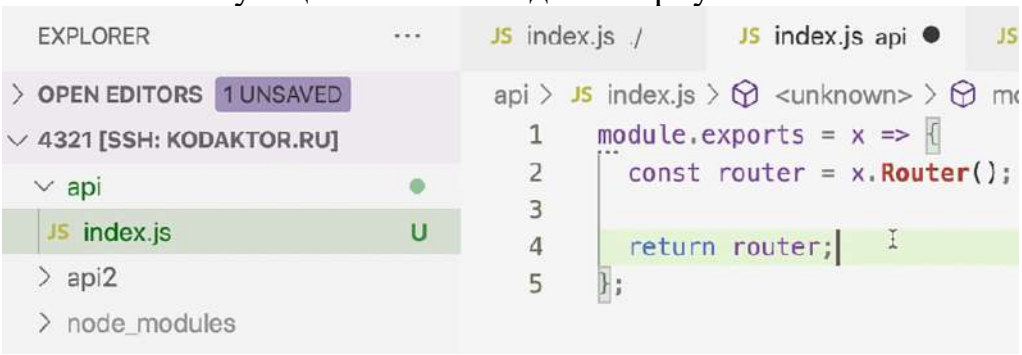


Рисунок 73. Возврат роутера в app

Таким образом, подход к выстраиванию взаимоотношений между основным приложением и роутерами которые обслуживают маршруты организованы так. В основном коде мы импортируем эти функции, передаём в их аргумент, инжектируя в качестве зависимости express. Мы передаём, функция запускается и возвращает экземпляр роутера, который представляет собой middleware, который необходимо будет подключить к основному конвейеру, используя метод use.

Так выглядит общая схема действий. С помощью созданного роутера мы можем обрабатывать маршруты, как бы примонтированные к корню дерева, за который этот маршрутизатор отвечает. Когда мы пишем здесь слэш, это означает что мы обрабатываем корень дерева маршрутов привязанного к api

```
router
.route('/')
```

То есть то, что в основном приложении выглядит как хост, слэш, api, слэш и т.д., здесь будет выглядеть как просто слэш.

И далее мы можем воспользоваться уже методами, соответствующими методам http. Например, мы можем написать здесь r.res.send (Арифметика: add, mpy). И это будет простым описанием того, что сможет делать этот маршрутизатор.

Далее мы возьмём субмаршрут add, и будем исходить из того, что он работает с двумя числами. Обозначим их: n1 и n2 – части параметризованного маршрута. Мы ответим на запрос по этому маршруту, выдав сумму значений первого параметра r.params.n1 и второго параметра r.params.n2

Express берёт на себя распознавание частей url, которые подпадают под этот шаблон: слово add, слэш, число, слэш, ещё число. URL-ы не подпадающие под этот шаблон, разумеется, не будут обрабатываться этим роутером.

```
router
.route('/')
.get(r => r.res.send('Арифметика: add, mpy'));

router
.route('/add/:n1/:n2')
.get(r => r.res.send('Got: ' + Number(r.params.n1) + Number(r.par
```

Рисунок 74. Обработчики маршрутов в маршрутизаторе

Для того чтобы теперь увидеть это в действии, нам нужно подключить этот маршрутизатор к основному коду приложения.

У нас здесь уже есть один такой маршрутизатор:

```
const Router = x.Router();
```

Рисунок 75. Создание объекта маршрутизатора

В базовом трафаретном коде он используется чисто для демонстрационных целей, для того чтобы показать саму возможность объединять в группы маршруты, у которых есть общие характеристики.

Прежде всего, затребуем маршрутизаторы из соответствующих файлов. Назовём первый ApiRouter, он находится в папке api, и поскольку представлен файлом index.js, то можно указать только папку. Поскольку мы экспортировали функцию, то в ApiRouter будет находиться именно она.

```
const ApiRouter = require('./api');
      const ApiRouter: (x: any) => any
```

Рисунок 76. Затребование маршрутизатора

Эта функция возвратит нам роутер тогда, когда мы вызовем её, передав ей в качестве параметра express (x).

```
.use('/', Router)
.use('/api', ApiRouter(x))
```

Рисунок 77. Подключение затребованного маршрутизатора как middleware

Это значение должно будет попасть на место аргумента x.

Рисунок 78. Подстановка / внедрение объекта app в аргумент маршрутизатора

Попробуем перейти на этот маршрут.

```
curl -k https://demoapp.kodaktor.ru/api/add/15/17
<h2>Got: 32</h2>%
```

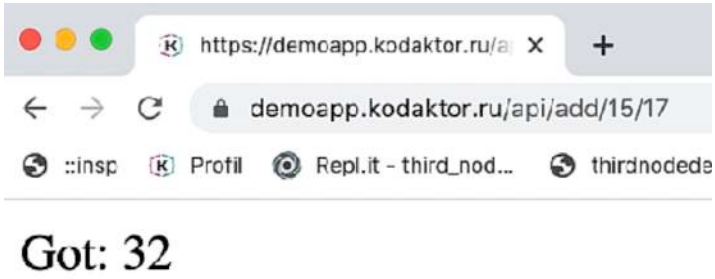


Рисунок 79. Обработка маршрута

Для того чтобы обрабатывать также запросы методом POST, мы продублируем обработчик и в нём заменим слово GET на одноимённое, а в ответе напишем Posted.

```
router
.route('/add/:n1/:n2')
.get(r => r.res.send('Got: ' + (Number(r.params.n1) + Number(
.post(r => r.res.send('Posted: ' + (Number(r.params.n1) + Numb
```

Рисунок 80. Обработка обращения методом POST

Снова отправив запрос, обнаружим уже скорректированный ответ.

```
-▶  curl https://demoapp.kodaktor.ru/api/add/10/13 -XPOST -i
HTTP/1.1 200 OK
X-Powered-By: Express
content-type: text/html; charset=utf-8
content-length: 10
etag: W/"a-Ioe9tNngIPpHz21p32cBKpIlJ0s"
date: Fri, 26 Aug 2022 00:16:20 GMT
connection: close

Posted: 23%
```

Рисунок 81. Ответ на запрос методом POST (параметры в URL)

Итак, отправляя запрос методом POST мы получаем один вариант ответа, отправляя методом GET получаем другой вариант ответа. Наше приложение уже не method-agnostic.

Чтобы добавлять дополнительные маршруты, создадим папку и назовём её routes и, соответственно, переместим и api, и api2 в эту папку. Visual Studio Code выступает в качестве интеллектуальной IDE и предоставляет нам возможность включить автоматическое обновление импортов.

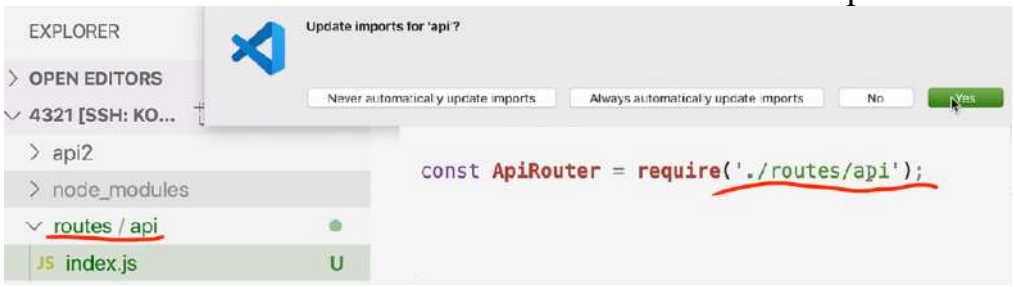


Рисунок 82. Обновление путей в Visual Studio Code

Таким образом, мы можем теперь подключить второй маршрутизатор и подмонтировать его в конвейер middleware, используя глагол use. Мы видим, что маршрутизатор выступает в роли управляющего кода, который маршрутизирует, то есть буквально «разруливает» запросы по тем блокам кода, которые реализуют нужную функциональность.

Вариант кода демонстрационного приложения с реализованными маршрутами доступен по адресу https://kodaktor.ru/demoapp.vm – осталось привести текст файла index.js из папки routes/api (об используемом здесь методе res.format сказано в следующем разделе):

```
module.exports = x => {
  const router = x.Router();
  router
  .route('/')
  .get(r => r.res.send('Арифметика: add, mpy'));
```

```
router
.route('/add/:n1/:n2')
.get(r => {
  const sum = Number(r.params.n1) + Number(r.params.n2);
  r.res.format({
    'text/html': () => r.res.send(`<h2>Got: ${sum}</h2>`),
    'application/json': () => r.res.json({"Got": sum})
  });
})
.post(r => r.res.send('Posted: ' + (Number(r.params.n1) + Number(r.params.n2))));

router
.route('/mpy/:n1/:n2')
.get(r => r.res.send('Got: ' + (r.params.n1 * r.params.n2)))
.post(r => r.res.send('Posted: ' + (r.params.n1 * r.params.n2)));


return router;
};
```

Листинг 18. Код подключаемого маршрутизатора в отдельном файле

Параметризованные маршруты подразумевают, что выделенные части URL (в первую очередь с помощью слэшей) являются чем-то вроде фактических параметров, передаваемых функциями при вызове. Но это относительно неклассический способ передачи серверу таких данных. Традиционным способом является использование QueryString (то, что браузер позволяет прочитать как location.search). Как мы уже говорили в предыдущей главе, node:http не распознаёт знак вопроса как нечто, выделяющее особую часть URL. Но Express содержит эту функциональность. Свойство объекта response под названием query содержит ссылку на объект, который состоит из пар «имя:значение».

Если обратиться к маршрутам вида ?a или ?a= или ?a=b, то мы получим в объекте query свойство a, которое имеет строковое значение, т.е. `typeof r.query.a == 'string'`

При этом в случаях ?a и ?a= значением query.a будет пустая строка. Это позволяет использовать такие свойства в качестве флагов, т.е. переход по маршруту ?a должен будет работать так же, как переход по маршруту ?a=true.

### Шаблонизация

В приложениях с достаточно чётким разделением полномочий по компонентам компонент представления (View) отвечает за формирование

окончательный разметки, которая передается в браузер, и традиционно в основе такой разметки лежит шаблон. Шаблон (template) получается в результате замены в обычном тексте определенных мест, на которых в разных ситуациях находится разное содержимое, на контейнеры-плейсхолдеры (placeholder, wildcard). В современном JavaScript шаблонные строки хорошо выражают суть происходящего: контейнеры переменного содержимого обозначаются знаком доллара и фигурными скобками, а весь текст ограничивается обратными косыми апострофами. По большому счету после появления шаблонных срок крайней необходимости в использовании отдельного инструмента шаблонизации на платформе Node.js уже нет, но, тем не менее, шаблонизаторы являются привычной частью серверных приложений и содержат не только синтаксис шаблонов, но некую систему управления ими.

Мартин Фаулер в своей книге «Шаблоны корпоративных приложений» (2016) классифицирует шаблонизаторы следующим образом:

- Template View
- Transform View

Вариант Template View подразумевает создание статических HTML-страниц. При этом происходит использование определенных маркеров, которые помечают гнёзда или плейсхолдеры, т.е. места для подстановки динамически генерируемого контента в процессе обработки запроса Эти помеченные места заменяются динамическим контентом и результат обработки запроса отправляется клиенту.

Template View в случае веб-приложений, в том числе серверных приложений, отвечает за V в структуре приложений, которые следуют шаблону MVC.

В тексте выше слово шаблон уже по факту использовано в разных смыслах. Например, имеют место паттерн проектирования и шаблон проектирования – в этом контексте паттерн и шаблон синонимичны, они описывают повторяющуюся структуру приложения – примерами выступают MVC и в более частной ситуации Template View.

Слово «паттерн» применяется при разговоре, например, о регулярных выражениях, поэтому стоит зафиксировать, что у слова паттерн есть смысловой оттенок схемы, образца, который задаёт класс объектов, которые ему соответствуют; с паттерном сверяются для того, чтобы выяснить, соответствует ли конкретный объект этому паттерну.

В то же время слово template используется для обозначения заготовки, и можно легко представить себе процесс возникновения такой заготовки, например, взять готовый текст и заменить в нем некоторые слова или последовательности слов последовательностями, которые начинаются с `${` и заканчиваются `}`.

И обратно, для того, чтобы по такому шаблону получить результирующий документ, нужно подставить на место этих областей или плейсхолдеров или

гнёзд конкретные данные. Это – шаблонизационная интерполяция (на контрасте с конкатенационной интерполяцией вида a + '<' + b).

Близким значением обладает слово boilerplate, за тем исключением, что оно иногда употребляется в несколько негативном смысле. Бойлерплейтный или трафаретный код возникает, когда код начинает повторяться, и возникает потребность написать его один раз, с тем, чтобы в дальнейшем только ссылаться на него. Когда мы видим, что какой-то код можно вынести в отдельную функцию, мы фактически избавляемся от его бойлерплейтности. Так мы получаем повторное использование, реюзабельность компонентов, модулей. Но в ряде случаев бойлерплейт необходим – как, например, при фиксации базового кода приложения.

Вид шаблонов Transform View представлен языком XSLT. В его основе трансформирующие узлы, которые получают на вход одну разметку (модель), а выдают другую, которую можно отрендерить.

Различаются и стили шаблонизации. При декларативном подходе объявляется, заявляется, каким должен быть результат. Порядок применения составных частей шаблонизатором может быть разным, результат от этого не меняется. Преобразующие узлы могут быть использованы столько раз, сколько нужно, то есть мы один раз описываем преобразующий узел и далее, сколько угодно раз применяем для имеющегося количества элементов, подлежащих преобразованию. При императивном подходе процесс преобразования описывается по пунктам, какие действия и в каком порядке нужно проделать, чтобы добиться результата. Для императивного подхода характерны такие структуры, как циклы for, условия if, и от перестановки мест таких конструкций результат вполне может измениться.

С точки зрения программирования шаблонизатор – это функция, которая часто называется словом render или даже template, которую возвращает компилятор шаблона, и которая сама возвращает разметку или представление. Символ «фигурная скобка» в стилизованной форме напоминает усы или велосипедные ручки, поэтому возникают такие названия шаблонизаторов как Mustache или Handlebars. Количество повторяющихся фигурных скобок обозначает уровень шаблонизации, например, уровень макета или меташаблона (шаблона шаблонов).

Рассмотрим в качестве примера изоморфный шаблонизатор **pug** (ранее известный как jade; у него тот же автор, что и у ExpressJS, поэтому эти два решения изначально тесно связаны).

Пример его подключения на клиентской стороне: https://kodaktor.ru/client_pug2

Для удобного создания шаблонов можно использовать сайт http://html2jade.org/ .

Для быстрого приобретения опыта работы с шаблоном сделайте следующее

**Упражнение 9**

1. С помощью curl "kodaktor.ru/j/users" получите ответ в виде JSON-строки

{"users":[{"login":"student@mail.com","password":"tneduts"},{"login":"myuserlogin@my.msk.ru","password":"mypas"},{"login":"prof@univer.msk.ru","password":"fessor"},{"login":"ego@yandex.ru","password":"selfish"}]}

2. В правой части на сайте html2jade вставьте код (https://kodaktor.ru/pug.js):

```
- result = JSON.parse('...')
- users = result.users
table
  tbody
    each user in users
      tr
        td= user.login
        td= user.password
```

3. … и полученную на первом шаге строку вставьте вместо многоточия.

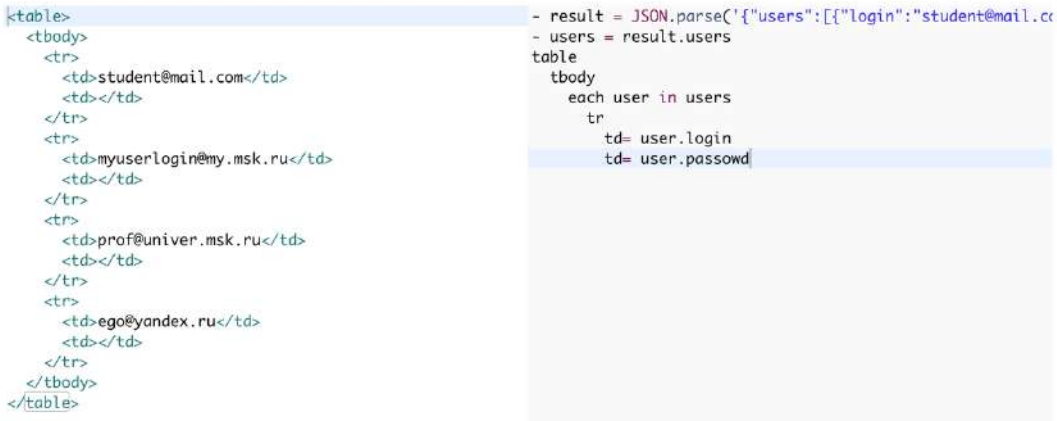


Рисунок 83. Сопоставление разметки и кода на шаблонизаторе

В левой части будет сгенерирована разметка по шаблону. Синтаксис выглядит достаточно декларативно и достаточно сильно абстрагируются от тегов. Места для подстановок обозначаются знаком равенства после тегов, например td=

Дефисом в синтаксисе pug обозначаются инструкции JavaScript. Мы в данном случае захардкодили данные, которые в нормальной ситуации приходят в шаблон извне при его вызове. Отступ, как в Python, имеет определяющее значение.

В Express чтобы отрендерить шаблон в разметку, нужно сделать следующее:

1. Создать в проекте папку views и в ней файл с кодом шаблона

(в качестве которого будет контент https://kodaktor.ru/pug.js *без первых двух строк*)

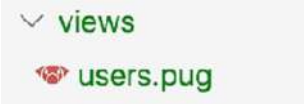


Рисунок 84. Расположение шаблона в папке views

2. Добавить pug в число зависимостей (npm i pug или yarn add pug)

3. Добавить в основном коде приложения (вероятно, после последних двух use с обработкой ошибок) инструкцию

```
app.set('view engine', 'pug')
```

В коде базового приложения kodaktor.ru/x эта строка имеет номер 17, и она закомментирована. Благодаря этому подключению Express будет искать представление именно в папке views.

Чтобы выдать по маршруту /api/users отрендеренную таблицу с логинами и паролями пользователей, мы каким-то образом получим данные о пользователях (из базы данных или с помощью fetch как в примере ниже) и передадим методу res.render два параметра:

1. Имя шаблона из папки views
2. Объект, в котором используемой в шаблоне переменной (в нашем случае users) сопоставляется конкретное значение

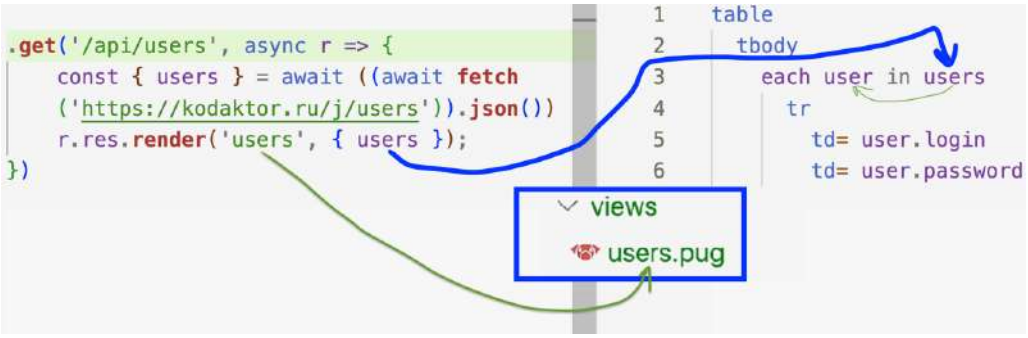


Рисунок 85. Подстановка литерального объекта в шаблон

Эту передачу значений для подстановки в гнёзда шаблона можно осуществлять не только в вызове метода render, но ещё и императивно – создавая свойства в объекте res.locals

Express поддерживает на уровне convenience-метода res.format то, что называется Content Negotiation, т.е. согласование контента. Если клиент посылает заголовок Accept с указанием mime-типа контента, серверное приложение может выдать контент именно в такой форме.

Мы можем отдать контент в сыром виде JSON-объекта для API-потребителя, либо в отрендеренном виде для пользователя, глядящего на экран браузера.

Достигается это сопоставлением методом res.render или res.json mime-типам, которые могут быть присланы в заголовке Accept (см. Листинг 18).

Мы отправляем информацию для метода res.format в заголовке Accept:

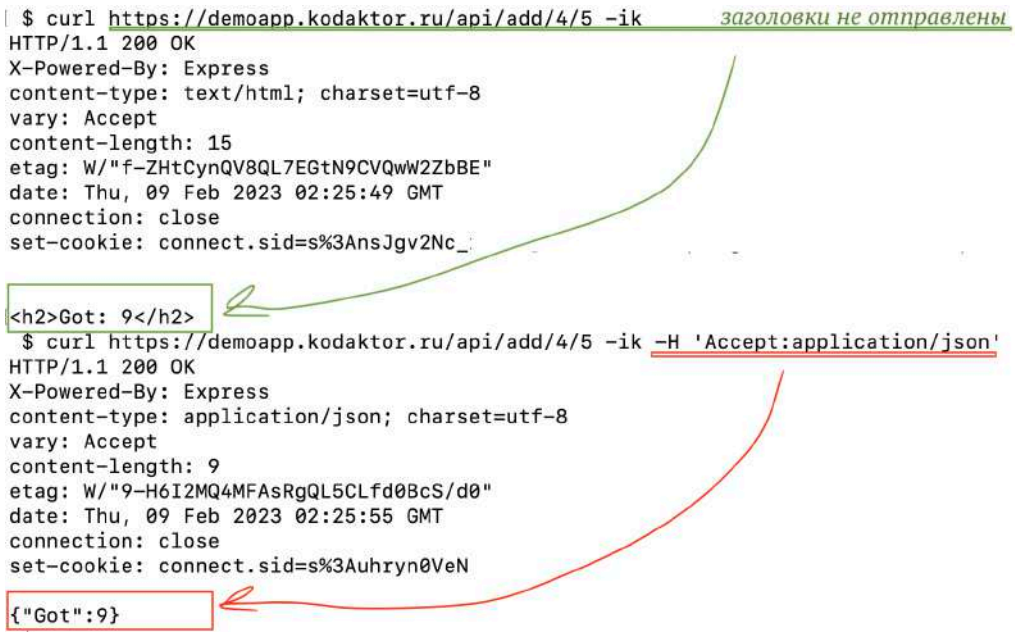


Рисунок 86. Заголовок Accept для res.format

Чтобы реализовать такое поведение для рассмотренного выше примера (Рис. 85), вы можете начать со следующего простого вызова res.format:

```
r.res.format({
  'application/json': () => r.res.json(users),
  'text/html': () => r.res.render('users', { users })
});
```

### Приём запроса с телом

В главе 4 мы научились принимать тело запроса с помощью node:http. В Express, как ни странно, не предусмотрено лёгкого штатного способа обращаться к телу запроса.

Рассмотрим четыре кейса:

1. Использование req как потока для перенаправления
2. Использование низкоуровневых зависимостей Express
3. Использование bodyParser
4. Считывание multipart-form-data (тело запроса в виде файла).

*Первый* вариант просто использует тот факт, что объект req наследуется из http:

```
import x from 'express';
import { createWriteStream as c } from 'node:fs';

const PORT = 4321;
const app = x();
app
  .all(/.*/, req => req.pipe(c('./exxxx-out.txt')))
  .listen(PORT);
```

Листинг 19. Перенаправление из тела сообщения с помощью pipe

*Второй* вариант связан с пакетами content-type и raw-body, которые есть среди зависимостей, которые загружаются при установке Express

```
import x from 'express';
import contentType from 'content-type';
import getRawBody from 'raw-body';

const PORT = 4321;
const app = x();

app.use((req, res, next) => {
  getRawBody(req, {
    length: req.headers['content-length'],
    limit: '1mb',
    encoding: contentType.parse(req).parameters.charset
  },  (err, string) => {
    if (err) return next(err);
    req.text = string;
    next();
  })
})
  .all(/.*/, (req, res) => {
    console.log(req.method);
    res.end(req.text);
  })
  .listen(PORT);
```

Листинг 20. Использование пакетов content-type и raw-body

Хотя технически можно отправить тело сообщения и с GET-запросом, http-клиентам не следует этого делать:

```
curl localhost:4321/abc -dname=ilya -HContent-Length=9 -XGET
```

*Третий* вариант создаёт с помощью middleware свойство body в объекте req, которое можно использовать в нижестоящих обработчиках запросов

```
import x from 'express';
import bodyParser from 'body-parser';

const PORT = 4321;
const app = x();

app.use(bodyParser.json())
  .all(/.*/, (req, res) => {
    console.log(req.method);
    res.send(req.body);
  })

  .listen(PORT);
```

Листинг 21. Использование пакета body-parser

```
-▸  http post localhost:4321 name=elias
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 16
Content-Type: application/json; charset=utf-8
Date: Mon, 29 Aug 2022 23:58:43 GMT
ETag: W/"10-M3rSuwZm4Zb2ewknna3pcNvcnhI"
Keep-Alive: timeout=5
X-Powered-By: Express

{
    "name": "elias"
}
```

Рисунок 87. Обработка POST-запроса с телом сообщения

Т.е. нужно добавить middleware bodyParser *до* маршрута, использующего post

Метод bodyParser.json() нужно подключать, чтобы middleware обрабатывало тело в виде JSON (нужно отсылать заголовок Content-Type!), а bodyParser.urlencoded({ extended: true }) для обычных пар вида name=value, причём в обоих случаях req.body является объектом с соответствующими свойствами и значениями.

Чтобы получить исходные данные до парсинга:

```
.use(bodyParser.urlencoded({ extended: true,
  verify(req, res, buf, encoding) {
    req.rawBody = buf.toString();
  }
}))
```

Если, например, отправить curl localhost:4321 -d=name
то «официальное» req.body станет пустым объектом, а rawBody – значением =name

**Упражнение 10**

1. Скопируйте и проанализируйте код со страницы https://kodaktor.ru/g/uploader1
2. Разместите код на своём компьютере в файле index.js в некоторой папке, предположим upload_demo
3. Запустите его, предположим, на порту 4321
   PORT=4321 node .
4. Перейдите в браузере по адресу
   http://kodaktor.ru/g/unsafe_uploader1?4321
5. Выберите на своём компьютере картинку (в формате PNG)
6. Нажмите на кнопку отправки
7. Убедитесь, что эта картинка с именем im.png находится теперь в папке upload_demo

*Четвёртый* вариант: обратите внимание, bodyParser не создан для того, чтобы принимать файлы, обрабатывая multipart form data, для этого лучше использовать модули типа busboy.

**Упражнение 11**

1. Создайте новый проект в отдельной папке
2. Установите express и busboy yarn add express busboy
3. В файле (назовём его bb.mjs) поместите код:

```
import x from 'express';
import Busboy from 'busboy';
import { createWriteStream as c } from 'node:fs';

const app = x();

app.use(x.static('.'));
app.post('/', r => {
```

```
    const busboy = Busboy({ headers: r.headers });
    console.log(r.headers);
    busboy
    .on('file', (_, file) => {
        console.log(file);
        const tmp = `${Math.random()}.png`;
        file
        .pipe(c(tmp))
        .on('finish', () => {
          r.res.send(`<a href="${tmp}">Ваш файл</a>`)
        });
    })
    .on('error', e => rs.end('ERROR boy'));
    r.pipe(busboy);
});

app.listen(4321);
```

Листинг 22. Использование пакета busboy

4. Запустите node bb.mjs
5. Перейдите в браузере по адресу http://kodaktor.ru/g/unsafe_uploader2?4321

6. Выберите PNG-картинку с компьютера и нажмите на кнопку отправки
7. Перейдя по полученной ссылке, убедитесь, что это та самая картинка, которую вы отправили

### Запуск приложения в формате сервиса

Если запустить веб-сервер в REPL, то при выходе из REPL он завершит свою работу. Если подключиться к виртуальному серверу с помощью SSH и запустить веб-сервер с помощью node, то, аналогично, после выхода из SSH-сеанса он завершит свою работу. Чтобы веб-сервер стал «настоящим», т.е. вёл себя как сервис и работал постоянно независимо от подключения администратора, его и нужно запустить в формате сервиса (в понимании Linux – как демона, процесса в фоновом режиме). Для этого можно использовать systemd: ставший стандартным диспетчер процессов. Команда service или systemctl позволяет запускать и останавливать процессы. Теперь systemd включает такие инструменты, как менеджер журналов (journald), менеджер сети (networkd) и менеджер устройств (udevd). Здесь d означает «демон». Чтобы демонизировать веб-сервер, мы можем описать путь до его сценария в файле с расширением service и поместить в /etc/systemd/system, а затем включить его автозагрузку с помощью systemctl enable.

Эти действия могут быть автоматизированы с помощью пакета **pm2**, который не только создаёт файл сервиса, но и ведёт учёт запущенных процессов, позволяя осуществлять базовый мониторинг.

Есть ещё один любопытный способ сделать так, что приложение не остановится после выхода пользователя из SSH-сеанса. А именно, нам нужно запустить веб-сервер от имени другого, заранее созданного (adduser demouser), пользователя.

После входа на сервер от root, выполните в SSH-терминале команду:

```
su demouser -c 'node -e "http.Server((r, rs) => rs.end(\"Hello \" + process.pid)).listen(12345);"' &
```

Выйдя из SSH-сеанса, через пару минут обратитесь к серверу по порту 12345 – вы должны будете увидеть, что веб-сервер выдаёт Hello с идентификатором node-процесс. Этот идентификатор далее можно использовать для слежения за процессом и его уничтожения.

В этом примере используется порт 12345 – в данном случае важно, что не 80 и не 443.

Если веб-сервер должен работать на 80 или 443 порту, то здесь возникнет трудность: с этими портами должен работать суперпользователь. Но существует решение authbind, которое позволяет осуществить привязку к таким портам от имени пользователя, отличного от root.

**Упражнение 12**

1. Создайте виртуальный сервер с Ubuntu 20.04 (можно попробовать и старшие версии) и node, добавьте нового пользователя (demouser)
2. Установите authbind:
   ```
   apt update -y && apt install authbind
   ```
3. Настройте authbind для работы с портом 80
4. touch /etc/authbind/byport/80 && chmod 500 /etc/authbind/byport/80 && chown demouser /etc/authbind/byport/80
5. Выполните команду запуска процесса node:
   ```
   su - demouser -c 'authbind --deep node -e "require(\"http\").Server((r, rs) => rs.end(\"Hello\")).listen(80);"' &
   ```

Разумеется, при использовании высокоуровневых решений, таких как Kubernetes, всё это берёт на себя инфраструктура.

### Использование ядер и потоков

Для размещения приложения, пригодного к использованию в production, нужно использовать либо PaaS с поддержкой платформы Node (включая размещение в виде контейнера), либо собственный виртуальный сервер. В последнем случае придётся продумывать решения всех непростых вопросов,

включая расход памяти и исполнение на нескольких ядрах для достижения горизонтального масштабирования (поскольку веб-сервер на node ассоциируется с одной виртуальной машиной, в данном случае это и вертикальное масштабирование). При наличии в доступе нескольких ядер мы можем создать кластер (модуль cluster), который является встроенным и который реализует размножение экземпляров серверов, использующих одну и ту же основу кода, прослушивающего заданный порт. Схема работы с ним выглядит следующим образом:

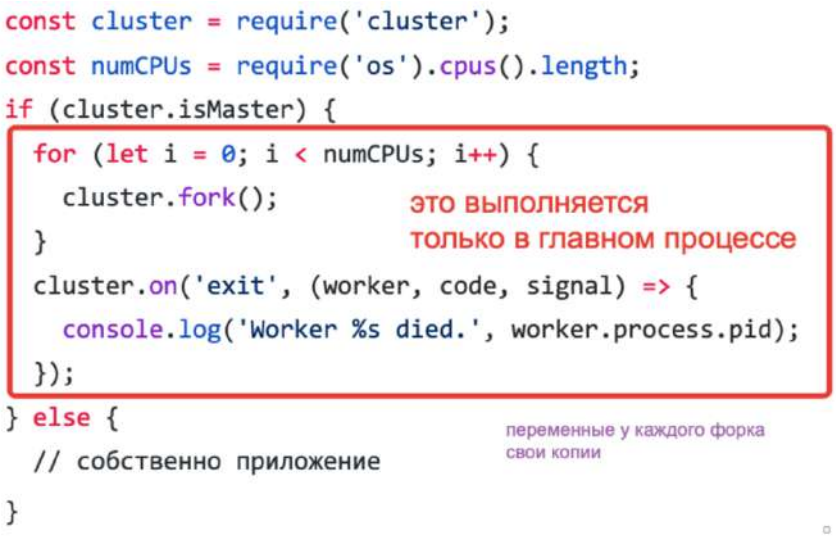


Рисунок 88. Оборачивание приложения в кластер

> **Примечание**
> Свойство isMaster объекта cluster стало называться isPrimary начиная с Node 16.0.0

**Упражнение 13**

1. Используйте виртуальный сервер как в предыдущем упражнении, добавьте нового пользователя (demouser)
2. Создайте ещё одно ssh-подключение в новом окне, чтобы перед глазами было два терминала, подключённых к серверу
3. В первом выполните команду `top -d 1 -u demouser` и убедитесь что процессов нет
4. Во втором запустите REPL от имени нового пользователя: `su demouser -c "node"`
5. Убедитесь что в первом окне появится процесс node
6. Во втором в REPL выполните инструкцию child_process.exec('node -e "setTimeout(() => console.log(10000), 10000)"', (e, s) => console.log(s));
7. Пронаблюдайте, как в первом окне появятся ещё два процесса (sh и node) и исчезнут через 10 секунд

8. Выйдите из REPL и убедитесь что в первом окне последний оставшийся процесс тоже исчез.

Это упражнение показывает, как можно форкануть (ответвить) процесс Node.js – используя такой метод как exec из модуля node:child_process и как можно использовать команду top для базового мониторинга процессов.

По адресу kodaktor.ru/cluster_x находится код уже кластеризованного приложения на базе Express. С этим кодом можно проводить следующие эксперименты:

- На виртуальном сервере создать пользователя (demouser)
- wget kodaktor.ru/j/cluster_x -O cluster_x.js
- Запустить приложение от имени пользователя su - demouser -c 'node cluster_x.js'
- Начать мониторить загрузку процессора top -u demouser
- Получить блокирующий код в виде модуля const pause = require('goss_pause')
- Задействовать его в каком-нибудь маршруте, перейти по нему и смотреть как меняется загрузка ЦП

Если начать с инстанса с 1 ядром, а затем добавлять в конфигурацию ядра, то можно будет наблюдать повышение отзывчивости приложения. Также для получения статистики полезно использовать инструменты бенчмаркинга типа Apache **ab**.

Когда есть необходимость в использовании блокирующего кода, следует использовать воркеры, которые упоминались в Главе 4. Они были внедрены в Node.js начиная с версии 10.5.0 (2018 г.).

**Упражнение 14**

1. См. первый шаг предыдущего упражнения
2. Перейдите в папку нового пользователя cd /home/demouser
3. Загрузите блокирующий код в отдельный файл wget kodaktor.ru/cluster0/pause2.js
4. В первом окне выполните команду top -u demouser и нажмите H (Shift H) чтобы показывать потоки
5. Во втором окне запустите REPL от имени нового пользователя: su demouser -c "node --v8-pool-size=0"
6. В REPL выполните инструкцию `void (new worker_threads.Worker('/home/demouser/pause2.js')).on('message', msg => console.log(msg));`
7. Вы должны будете увидеть, как к имеющимся потокам добавится ещё один с загрузкой процессора 99% (в то время как остальные потоки останутся незагруженными!)
8. Через 10 секунд вновь созданный поток исчезнет

Можно изменить код файла с воркером, обернув в обработчик события от родительского процесса:

```
parentPort.on('message', () => {
  ((y = 10000, x = Date.now()) => { while (Date.now() - x < y) ;})();
  parentPort.postMessage('Done!');
});
```

В этом случае чтобы увидеть загрузку потока, нужно сохранить ссылку на воркер и затем отправить ему сообщение:

```
const worker = (new worker_threads.Worker('/home/demouser/pause2.js')).on('message', msg =>
console.log(msg));
worker.postMessage('go');
```

В подготовленном автором пособия сценарии, который используется в Упражнении 15, сравнивается поведение при выполнении блокирующего кода прямо в приложении (маршрут /blocking) и при выполнении в воркере (маршрут /nonblocking). Обработчики маршрутов сообщают клиенту свои идентификаторы процессов, чтобы можно было ясно видеть, кто обрабатывает текущий запрос.

**Упражнение 15**

Шаги 1 - 4 – как в предыдущем упражнении

5. Загрузите https://kodaktor.ru/j/cluster_workers.js и запустите этот файл на выполнение во втором окне терминала, подключенного к серверу
6. Откройте ещё 4 окна терминалов (для них не потребуется подключение к серверу, т.к. они предназначены для отправки запросов)
7. Если в системе 2 ядра, то в первом окне будет 3 основных процесса (чтобы это увидеть лучше временно скрыть отображение потоков) – один процесс является мастер-процессом кластера (MAIN), он не отвечает на запросы, два других являются результатами создания дубликата основного кода приложения, отвечающего на запросы) для каждого из ядер
8. Если из одного и того же окна терминала подавать одинаковые запросы на корневой маршрут, то скорее всего вы увидите чередование обработчика, это будет то одно ядро, то другое – карусельный принцип
9. Если в одном из окон подать запрос к блокирующему маршруту, то это окно тоже заблокируется, в первом окне вы увидите загрузку одного из основных процессов, но из свободного терминала в этот момент можно будет подать запрос к корневому или неблокирующему маршруту, и он будет успешно обработан.
10. Чтобы наблюдать картину более детально, можно вернуть режим просмотра потоков, в двух «запросных» терминалах подать

блокирующий и неблокирующий запросы, а в ещё двух – обычные к корневому маршруту.

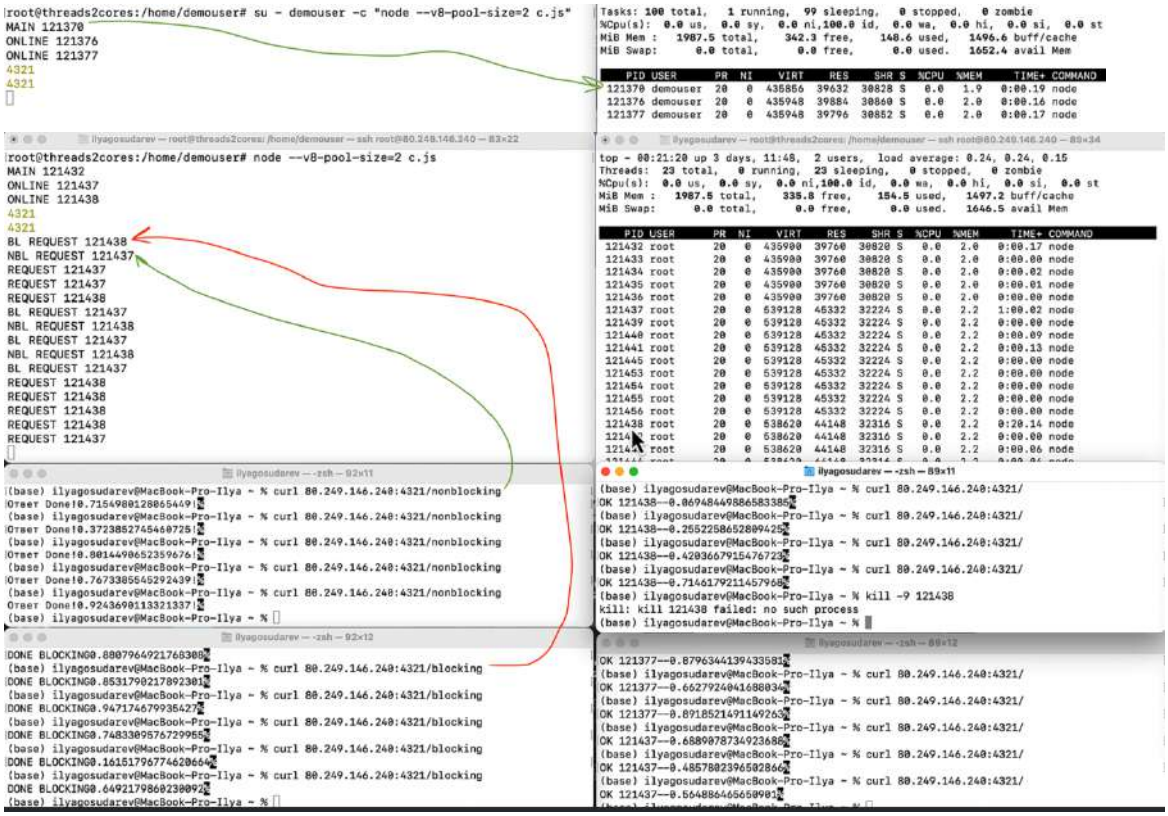

Рисунок 89. Отправка блокирующих и неблокирующих запросов

☞ Из эксперимента можно сделать главный вывод: эффективным решением будет запускать приложение на кластере при наличии нескольких ядер и выносить блокирующий код в воркеры. Это позволит на максимальную мощность использовать свойственные Node.js возможности асинхронного исполнения запросов. Подробное освещение темы воркеров см. в [5].

На этапе написания пособия (январь 2023) свежайшая стабильная версия Express имела номер 4.18.2, при этом доступна 5.0.0-beta.1 (npm i express@next)

https://expressjs.com/en/5x/api.html#res

В 5-й версии, которая относительно недавно вернулась в активное состояние разработки после нескольких лет затишья, пока заявлены такие изменения, как:

- Добавление модификаторов ?, *, и + в паттерн сопоставления маршрута;
- Упрощение использования async-функций в качестве обработчиков маршрутов и middleware
- Асинхронное исполнение res.render

**Примечание**

В течение некоторого периода «преемником» Express многие считали библиотеку Koa, популярность которой пришлась на время популярности генераторов в ES2015, но затем она отошла на второй план… Часто можно видеть, что вместо Express рекомендуют использовать fastify или другую более свежую библиотеку, но это никак не снижает значимости Express в качестве референсного решения.

☞ Поскольку это пособие посвящено введению в разработку на Node на материале серверных веб-приложений (а не глубокому профессиональному изучению), мы не будем здесь обсуждать реализацию на Express MVC-подхода (предлагаем вам исследовать состояние рынка, рассмотреть возможность использования Nest.js, основанного на TypeScript), а также не касаемся сессий, использования баз данных (но заметим, что в связке с Express и Nest часто встречается MongoDB, которая основана на JSON и включает JS REPL). Кроме того, здесь не затрагивается вопрос тестирования.

Что касается экспериментов с применением нагрузочного тестирования (бенчмаркинга) в качестве основного метода получения данных о работе Express и других библиотек-фреймворков, то такие исследования уже стали частью научного анализа. Определённый опыт накоплен нами в рамках магистратуры «Веб-технологии», см. например совместную работу D. Demashov and I. B. Gosudarev, "Efficiency evaluation of Node.js web-server frameworks", CEUR Workshop Proceedings, vol. 2590, pp. 1, 2020.

Подробное освещение всех аспектов работы с Express дано в замечательной книге Итана Брауна [6].

В 2000 году была опубликована докторская диссертация Роя Филдинга род названием Architectural Styles and the Design of Network-based Software Architectures – Архитектурные стили и проектирование архитектуры программного обеспечения на основе сетей (https://kodaktor.ru/fielding.pdf).

Существенной частью этой диссертации стало то, что называется Representational State Transfer (REST) – Передача Репрезентативного Состояния (или Репрезентативная передача состояния (приложения)). Необходимо отметить, что Рой Филдинг был одним из создателей протокола HTTP и в процессе работы над HTTP формировались те принципы, выкристаллизовывались те лучшие практики, которые в итоге и стали частью Веба, веб-технологий. Это архитектурный стиль, который не представляет собою формального стандарта. Скорее это обобщение, которое было сделано в результате работы над Вебом как распределенной системой. REST выглядит как набор ограничений или требований, накладываемых на взаимодействие между компонентами системы. Эти принципы воплощались в жизни с 1994 года при создании соответствующих библиотек и спецификаций.

В этот период их называли HTTP Object Model по аналогии с Document Object Model (DOM). Некоторое время разработчики веб-приложений игнорировали часть этой модели; в основном использовались, в частности, методы GET и POST-протокола HTTP (в HTML DVD в определении элемента form мы находим, что значениями атрибута method являются GET либо POST). В то же время, помимо GET и POST, протокол HTTP предполагает также и другие методы. И развитие веб-технологий, усложнение структуры и функциональности веб-приложений привело к появлению идеи о том, что и другие методы тоже следует использовать. Это очень сильно связано с развитием веб-сервисов, которые выходят, конечно, же за пределы элементов форм и пользовательского интерфейса браузера. Таким образом интерес к REST возродился, сама эта аббревиатура стала популярной, как, например, AJAX, хотя составляющие технологии использовались задолго до появления моды на слово.

И хотя REST (RESTful) является именно стилем, на сегодняшний день уже появились спецификации – здесь уместно будет упомянуть OpenAPI — эта формализованная спецификация и экосистема инструментов, представляющая интерфейс между фронтэнд-системам, кодом библиотек низкого уровня и коммерческими решениями в виде API, используется для описания создания и использования и визуализации веб-сервисов REST.

Изначально разработка спецификации под названием Swagger Specification проводилась с 2010 года компанией Smart Bear. Теперь эта спецификация используется повсеместно для того, чтобы создавать, в том числе, автоматически генерировать программные интерфейсы.

### Создание декларативного сервиса на основе OpenAPI

Создание шлюзов на основе спецификации OpenAPI в Яндекс Облаке использует декларативный подход, при котором мы не описываем, как это должно работать – мы описываем, что должно получаться.

Если обратиться к ресурсу с этой спецификацией через браузер, то ответ будет выдан в виде HTML, в соответствии с тем, что находится в заголовке Accept запроса, отправляемого браузером. В то же время, если в curl добавить заголовок Accept с указанием Content-Type: application/json, ответ будет именно в формате JSON.

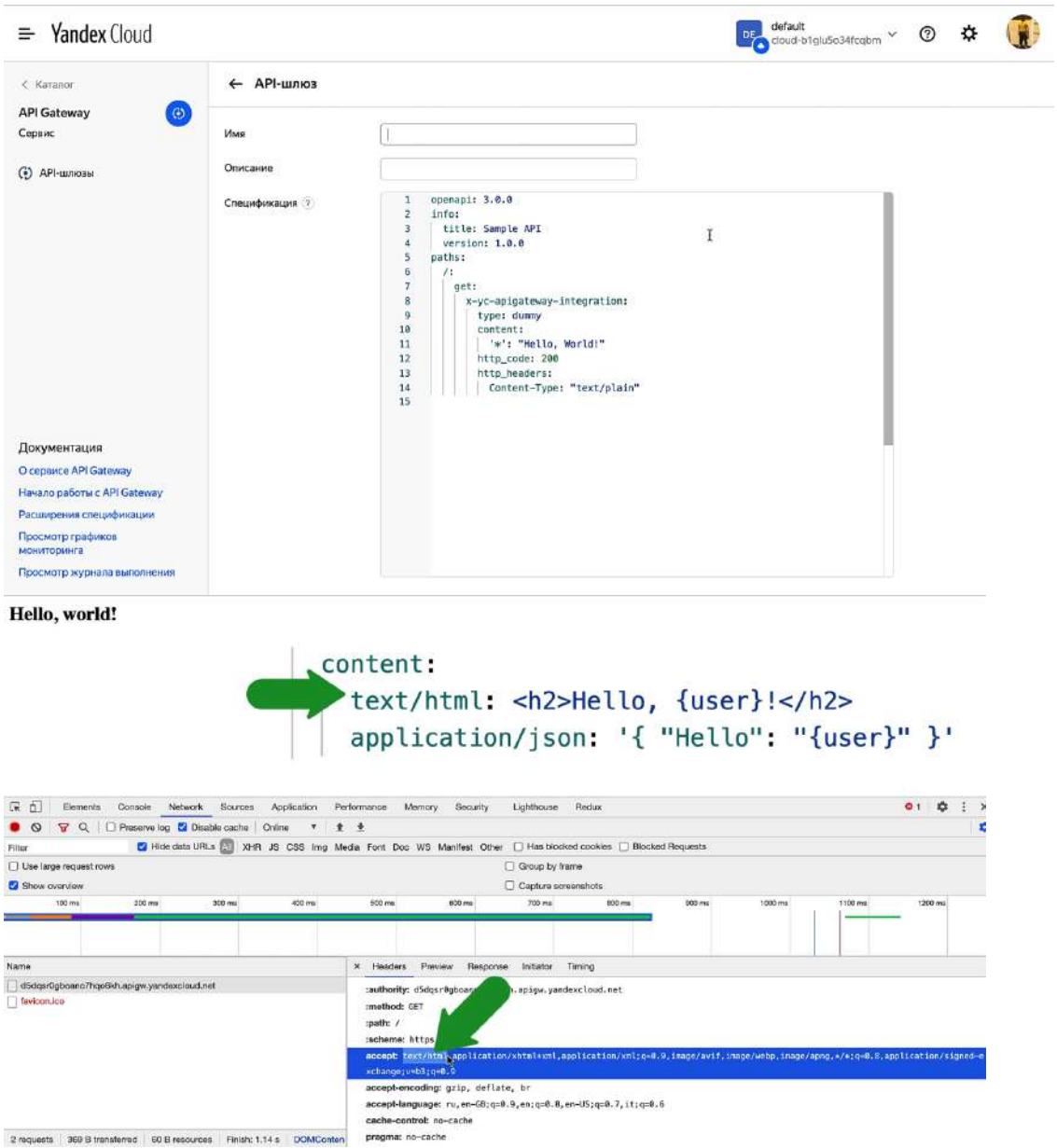


Рисунок 90. Content Negotiation на языке OpenAPI

Используя описательные возможности OpenAPI, мы можем реализовать выдачу содержимого в различных форматах (Глава 5) или, в терминах REST, различные репрезентации или представления одного и того же ресурса.

Эти форматы представляют ресурс – они являются его *представлениями*.

Рисунок 91. Запрос на представление ресурса с помощью Accept

В самом общем виде, **ресурс** — это что угодно, что может иметь имя. Это и изображение, и предоставляемый сервис и даже, возможно, человек с точки зрения именования. Под признаки ресурса подпадает любое понятие, которое может стать объектом гиперссылки. А с точки зрения реализации взаимодействия в сетевой структуре — это отображение, которое возвращает различные представления или репрезентации, то есть представление – это результат оформления ресурса в виде конкретного набора данных.

**Состояние** приложения – это конкретный набор его характеристик, значений параметров. Речь идет о совокупном виртуальном приложении, которое образуется в ходе взаимодействия компонентов системы. Оно определяется данными, выполненными запросами, обработкой этих данных агентами пользователей. Приложение развивается во времени от одного состояния к следующему. Это иллюстрируется браузером, его пользовательским интерфейсом. Пользователь выбирает ссылки, щёлкает по ним, в результате приложения переходит в новое состояние. После щелчка по ссылке новая страница представляет следующее состояние приложения, которое было передано как ответ на запрос. Отсюда и термин Representational State Transfer (REST), то есть передача состояния приложения на основе детерминирующего описания требуемого ресурса. В некоторой степени этот подход противопоставляется RPC (Глава 2), при котором на сервер передаётся команда или код для исполнения.

Важное значение приобретает вид **URL**, так как именно URL становится свёрнутым описанием требований, однозначно определяющих ресурс. Очень важное значение имеет использование методов HTTP. Хотелось бы здесь выделить такие характеристики, как **безопасность** и **идемпотентность**. Метод считается безопасным, если его действие не приводит к модификации представления ресурсов. Это как бы метод, не мутирующий состояние

объекта. Если предполагается, что воздействие методов модифицирует, мутирует ресурсы, то они в этом смысле небезопасны, например, очевидно, что DELETE означающий удаление, не является безопасным

| | SAFE | IDEMPOTENT |
|---|---|---|
| DELETE | ✗ | ✓ |
| GET | ✓ | ✓ |

Рисунок 92. Безопасность и идемпотентность

GET – метод, который предназначен для получения ресурсов, в этой трактовке является безопасным.

**Идемпотентность** означает, что действие может быть совершено несколько раз и при этом после первого раза результат уже не меняется, то есть действие выполняется один раз, получается результат и далее во второй и в третий и так далее раз результат остается тем же самым. При извлечении ресурса с указанием конкретного идентификатора, очевидно, мы ожидаем, что сколько раз ни затребуем этот ресурс, будет возвращаться именно он. С другой стороны, если рассмотреть удаление ресурса: мы удалили ресурс с определенным уникальным идентификатором. Очевидно, что после удаления ничего нового уже не случится, невозможно удалить повторно то, что уже было удалено и при этом что-то изменить, следовательно, результат будет оставаться тем же самым – отсутствие ресурса. Если метод безопасен, то он также идемпотентен.

### Критерии RESTful

Чтобы сказать является ли нечто REST или не является, нужно ответить на вопрос о соответствии реализованного API набору критериев.

- Клиент-серверное взаимодействие
- Stateless
- Кэшируемость
- Слои системы
- Интерфейс:
  - идентификаторы ресурсов,
  - представления
  - полные ответы от сервера
  - HATEOAS (Hypermedia as the engine of application state)
- [Code on demand]

Рисунок 93. Критерии RESTful

Первый критерий – это то, что взаимодействие происходит между компонентами, которые ясно идентифицируются как клиент и сервер по исполнению их функций.

Критерий Stateless означает, что нет состояния, на сервере не должна храниться информация о клиенте, то есть сервер не должен хранить клиентские сеансы сессии. Если необходимо, это должно быть сделано на клиенте. Любые запросы, поступающие на сервер, должны содержать всю информацию, необходимую для обработки этого запроса. Это не означает, что север не может хранить вообще какое бы то ни было состояние, поскольку очевидно, серверы сохраняют состояние ресурсов в базах данных, но они не должны хранить в памяти временное состояние приложения. Запросы могут выполняться любым из серверов в кластере, и эти сервера могут быть перезапущены без потери информации. Это ограничение обеспечивает масштабируемость приложения. Хотя, конечно, оно имеет и свои недостатки, так как клиент должен многократно посылать аутентификационную информацию с каждым запросом.

Кэшируемость: если при повторных одинаковых запросах получается один и тот же ответ, то этот ответ должен быть кэширован клиентом и соответственно, посредниками. Архитектура REST требует, чтобы ответное сообщение содержало указание на то, должен ли ответ быть кэширован, если да, то как долго.

Система потенциально может состоять из многих слоев. Например, за reverse proxy, за NGINX находятся множественные сервисы, каждый из которых решает какую-то свою задачу.

Должен быть зафиксирован интерфейс. Унификация интерфейса способствует уменьшению связанности и позволяют клиенту и серверу быть заменяемыми и независимыми друг от друга.

В REST каждому ресурсу присваивается идентификатор. HTTP-идентификаторы — это и есть URL-ы.

Предполагается возможность существования различных представлений ресурса: мы уже рассмотрели это на примере согласования формата контента.

Ответ от сервера должен содержать всю полноту информации так, чтобы полностью удовлетворить запрос клиента. HATEOAS (HyperMedia as the engine of application state) – гипермедиа как движок состояния приложения – означает что сервер в своём ответе включает в список тех действий, которые клиент может осуществить после обработки ответа. Проще говоря, вместе с каждым ресурсом, с каждой-веб страницей должен прилагаться список гиперссылок, по которым можно пройти дальше. Очевидно, что это требование имеет смысл только для о взаимодействии между таким агентом пользователя как браузер и сервером, который предоставляет веб-страницы сайта. Если же говорить о программных интерфейсах, обеспечивающих взаимодействие между программными сущностями, то это требование теряет смысл и поэтому строго говоря в большинстве случаев мы не можем иметь

дело с абсолютно точной в соответствии со всеми требованиями реализацией REST. Чаще корректнее говорить о REST-подобном API.

Есть еще необязательное требование – Code-on-demand, которое, однако, в некотором смысле привело к появлению языка JavaScript. Речь идет о том, что сервер может при определённых условиях возвращать не готовый ресурс, а код, который клиент должен исполнить и таким образом воссоздать ресурс на своей стороне. Однако, именно это делает возможным существование динамически перестраиваемых интерфейсов, которые используют AJAX, на этом зиждется работа таких фреймворков как React, работающих с компонентами, поскольку они получают часто информацию именно в формате JSON с сервера и, используя встроенную шаблонизацию, в итоге превращают её в то, что становится частью пользовательского интерфейса, веб-страницы.

### Операции с ресурсами на основе HTTP-глаголов

Вот примеры того, как могут выглядеть операции с ресурсами. Обычное обращение к адресу методом GET в браузере приводит к выдаче веб-страницы, так как браузер отсылает заголовок Accept: https://fork.kodaktor.ru/restusers

Если в curl указать принимаемый тип контента явным образом, мы получим HTML-теги

```
curl -kL fork.kodaktor.ru/restusers -HAccept:text/html
```

В противном случае результат будет выдан как application/json,

```
curl -kL fork.kodaktor.ru/restusers
```

Методом POST, указав логин и пароль, мы создадим нового пользователя.

```
curl -k -H 'Content-Type:' 'https://nd.kodaktor.ru/users' -d '{"login":"user31012023", "password":"unknown"}'
```

Отправляя запрос методом GET и указывая конкретный идентификатор, мы запросим информацию о конкретном пользователе:

```
curl -k -H 'Content-Type:' 'https://nd.kodaktor.ru/users/3'
{
  "login": "Zhigulskiy",
  "password": "1252",
  "id": 3
}
```

Мы также можем с помощью метода PATCH заменить этому пользователю пароль.

```
curl -k -H 'Content-Type:' 'https://nd.kodaktor.ru/users/3' -XPATCH -d '{"password":"new"}'
```

То же самое с PUT.

Ну и, конечно, мы можем полностью удалить запись о пользователе.
curl -k -H 'Content-Type:' 'https://nd.kodaktor.ru/users/10' -XDELETE

## Реализация Express-приложения в формате микросервиса

**Упражнение 16**

Рассмотрим пример создания приложения Express в виде сервиса под управлением инфраструктуры Яндекс Облако

Перейдем в консоль управления Яндекс Облаком.

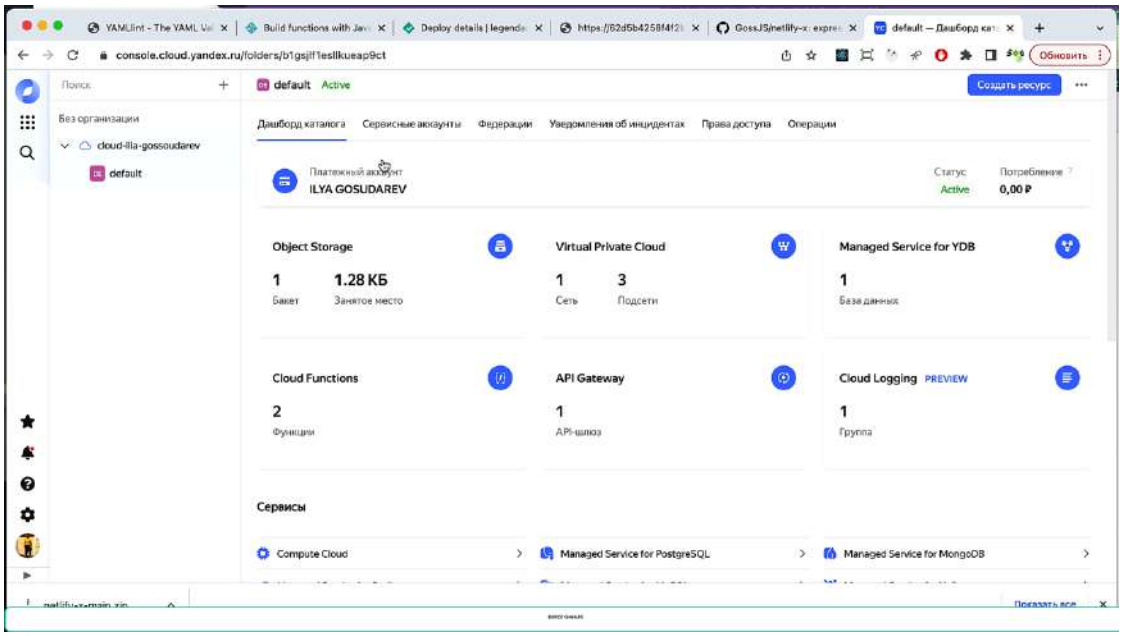


Рисунок 94. Консоль Yandex.Cloud

Для того, чтобы все это можно было делать, естественно, необходимо, сперва создать собственный аккаунт.

Также нам понадобится *сервисный* аккаунт – от его имени система работает с ресурсами.

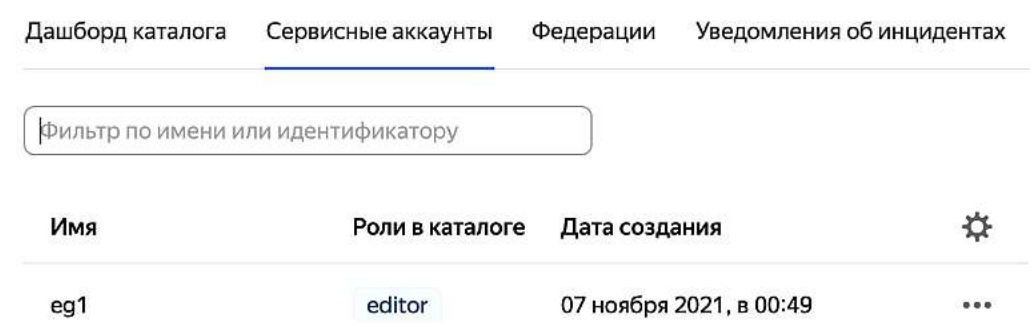


Рисунок 95. Сервисный аккаунт

Переходим в раздел функций и нажимаем на кнопку создания функции:

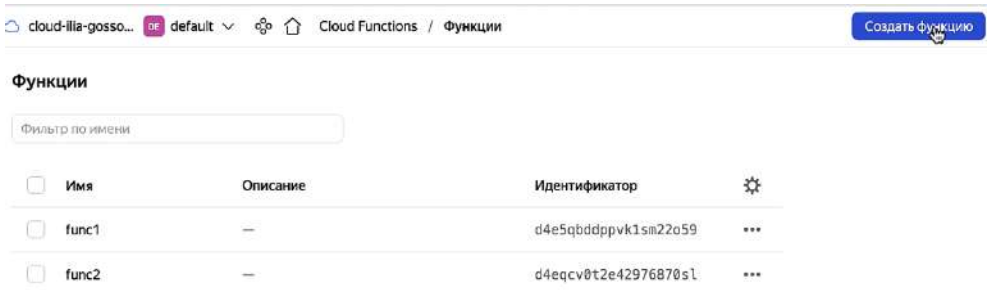


Рисунок 96. Облачные функции

Необходимо задать имя функции

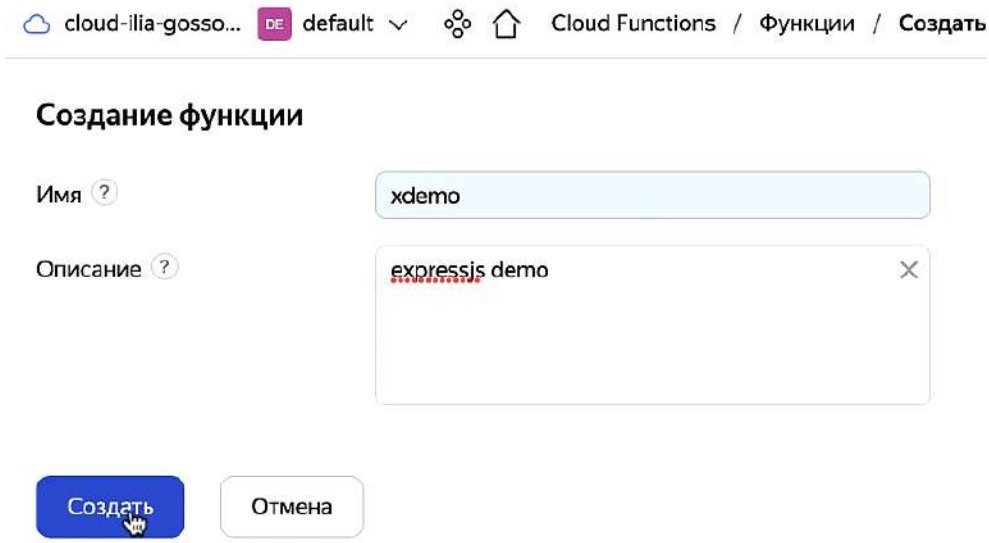


Рисунок 97. Создание облачной функции

И в следующем окне далее выбрать платформу.

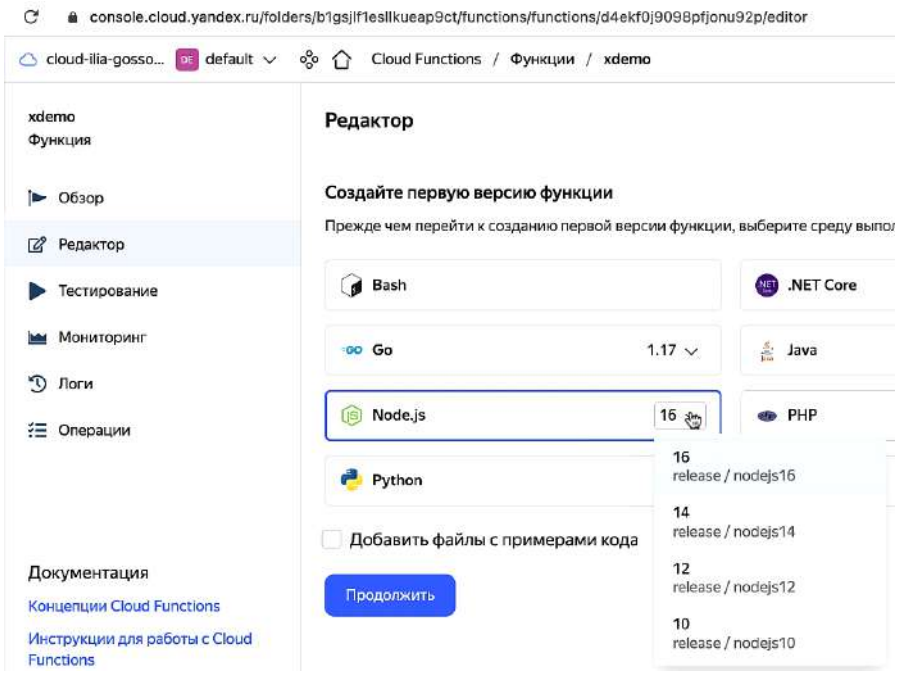


Рисунок 98. Выбор платформы для запуска функции

Мы видим перед собой теперь редактор. Если все уже заранее подготовлено – код и зависимости, то можно все одним пакетом в виде zip-архива загрузить прямо в редактор.

В нашем случае файлов будет совсем немного, поэтому мы создадим их с нуля в редакторе, и прежде всего package.json.

Рисунок 99. Создание файла зависимостей

В нём нам понадобится две зависимости Express и пакет serverless-http, назначение которого мы уточним далее.

Рисунок 100. Зависимости лямбда-функции

Для функции важно, что здесь указаны именно dependencies, а не devDependencies.

Второй файл index.js будет собственно нашим приложением, оформленном в виде λ-функции. Он содержит всё то что мы рассматривали ранее в Главе 5, за исключением использования зависимости serverless-http

```
const express = require('express');
const serverless = require('serverless-http');
const app = express();

const router = express.Router();
router.get('/', (req, res) => res.send(`
  <h1>Привет студентам от Express.js, Yandex.Cloud и GossJS!
${req.originalUrl}</h1>
  <h2>${new Date().toLocaleDateString('en-GB')} ${new
Date().toLocaleTimeString()}</h2>
`));
router.get('/login', (req, res) => res.json({ login: 'admin', route:
req.originalUrl }));
router.get('/api/mpy/:n1/:n2', (req, res) => res.json({ result: req.params.n1 *
req.params.n2 }));

app.use('/', router);

module.exports.handler = (event, context) => {
  const patchedEvent = {
    ...event,
    path: ( event.url.lastIndexOf('?') === event.url.length - 1 ) ?
event.url.substring(0, event.url.length-1) : event.url  ,
    originalPath: event.path,
  };
```

```
  return serverless(app)(patchedEvent, context) ;
};
```

Листинг 23. Базовое Express-приложение в виде λ-функции

После этого остается сделать один важный шаг. Такие системы, как облачные функции в Яндекс Облаке, исходят из того, что функциональность разбивается на сколь угодно много функций, каждая из которых работает как мини-приложение. Можно считать, что всё это ***μ***-сервисы и даже наносервисы. По отношению к основной системе все они являются модулями. Особенно когда используется require, достаточно естественно, что они подключаются **синхронно**. Такие λ-функции экспортируют в систему обработчик-handler, который нечто возвращает с помощью инструкции return.

Однако стиль библиотеки Express — это коллбэки, которые вызываются основой Express-предложения при наступлении события request и, конечно, эти коллбэки ничего не возвращают, а вызывают такие методы, как res.json или res.send.

Для того, чтобы увязать стиль синхронного возврата и стиль коллбэков используется обёртка serverless-http, на вход которой мы подаем app, являющееся по сути middleware, обрабатывающим запросы. Обёртка создает функцию, которой нужно передать объект события и объект контекста, которые в совокупности описывают всё, что функции нужно знать о данных, пришедших от клиента.

Для того, чтобы хорошо работал маршрутизатор Express, нам нужно несколько улучшить объект события, «пропатчить» его, тем самым обойдя тот момент, что шлюз для работы с данными передаёт знаки вопроса, так что в итоге мы будем возвращать в систему обёрнутое приложение app, которое будет работать с исправленным объектом события и с контекстом.

Чтобы наша функция могла заработать, нам нужно указать точку входа (это тот самый обработчик-хэндлер, который мы экспортируем в систему, т.е. index.handler.

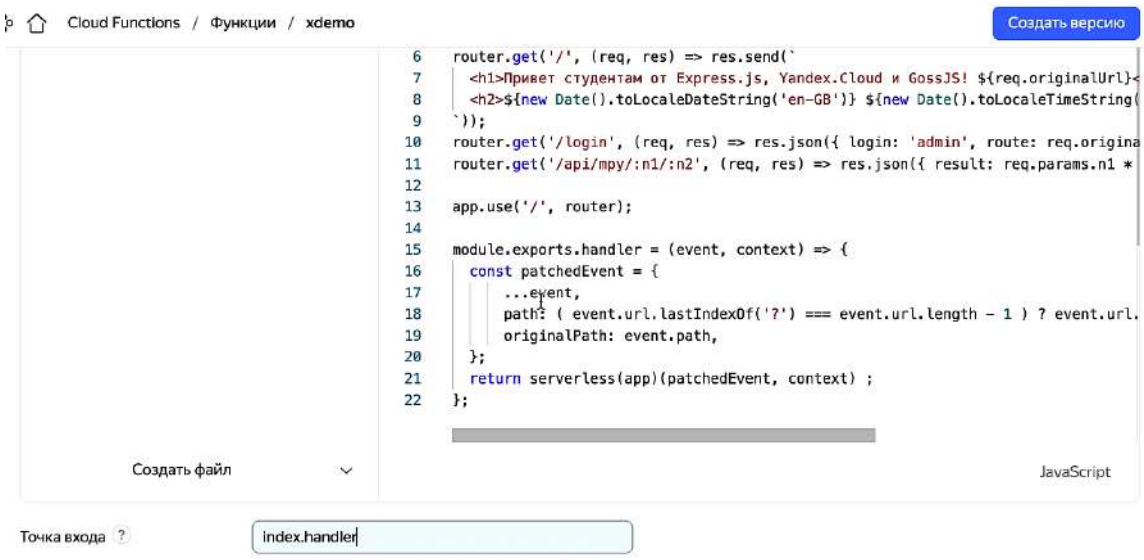


Рисунок 101. Основной код лямбда-функции

Параметры таймаута и памяти мы оставим по умолчанию – у нас небольшая быстро выполняющаяся функция. И так же нужно будет выбрать сервисный аккаунт, от имени которого будет выполняться эта функция.

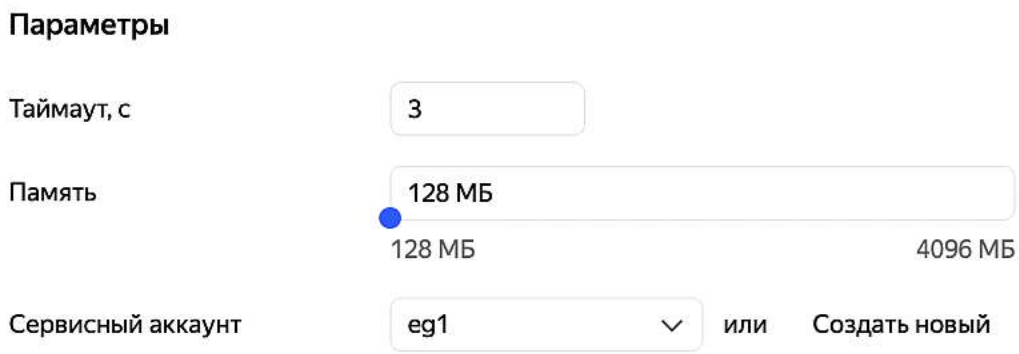


Рисунок 102. Параметры запуска лямбда-функции

Далее нужно создать версию и скопировать полученный идентификатор функции. Он понадобится для создания шлюза.

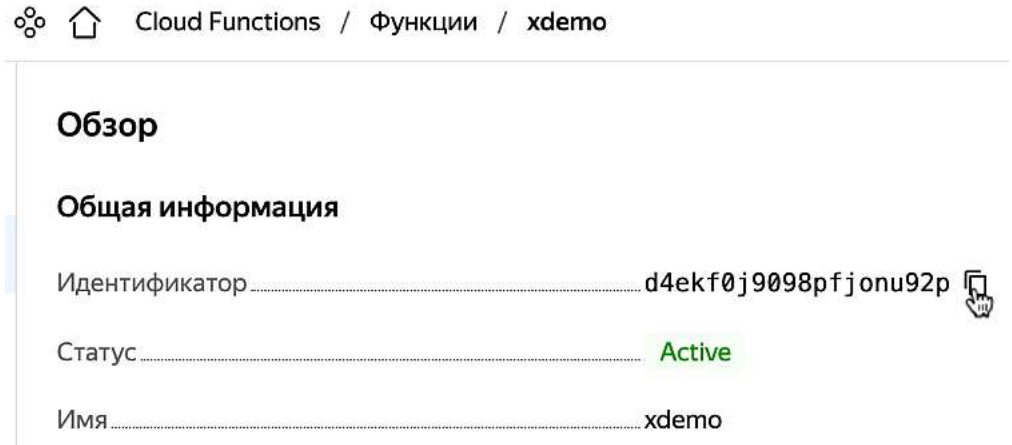


Рисунок 103. Присвоенный средой идентификатор функции

Шлюз нужен для того, чтобы стандартизовано и проверяемо, валидируемо передавать данные приложению. Они при этом проходят процесс сверки со схемой.

Создадим шлюз.

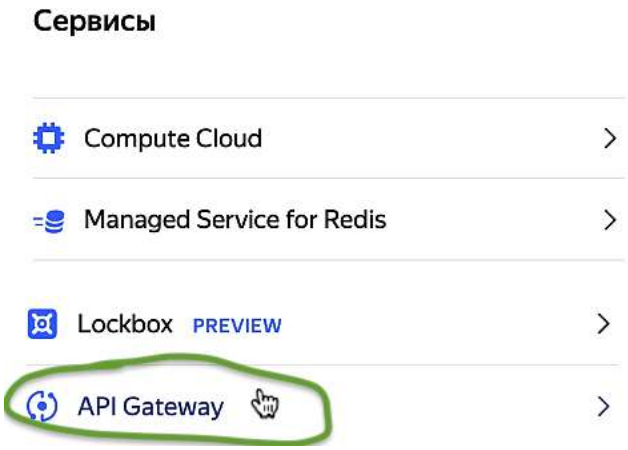


Рисунок 104. Создание шлюза для функции

В системе по умолчанию есть пример на языке OpenAPI – показано, как можно реагировать на корневой маршрут, достигаемый методом GET и выдавать простой текстовый контент.

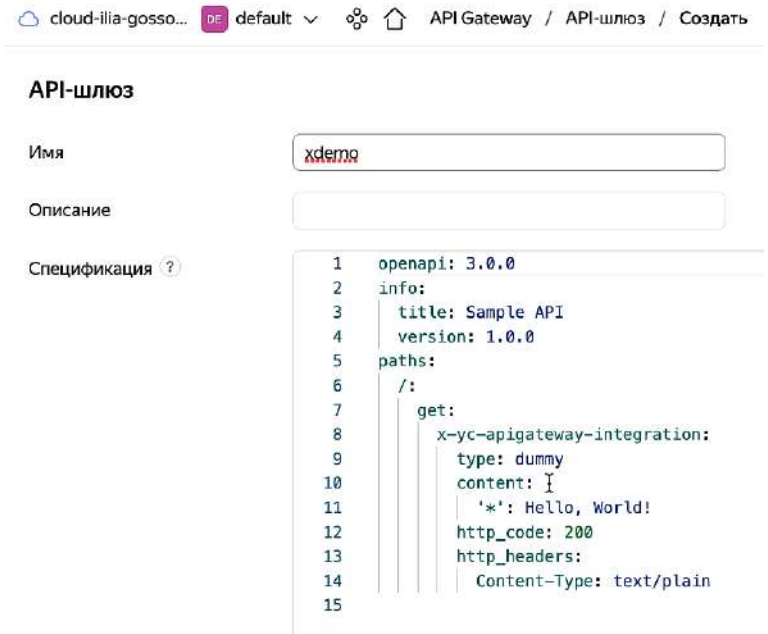


Рисунок 105. Простой пример описания шлюза на OpenAPI

Но для нашей функции потребуется более подробное описание, поскольку у нас есть три маршрута. Это корневой маршрут, по которому возвращается некоторый HTML, маршрут /login, по которому возвращается

Логин и, самое интересное, параметризованный маршрут, в котором сменная часть представлена регулярным выражением.

```
26    /api/{proxy+}:
27      get:
28        x-yc-apigateway-integration:
29          type: cloud_functions
30          function_id: d4e......dk
31          tag: $latest
```

Рисунок 106. Описание маршрута с параметрами

Необходимо вставить в нужные места ранее скопированный идентификатор функции, а также идентификатор аккаунта.

(Код см. по адресу https://kodaktor.ru/gateway_express)

```
openapi: 3.0.0
info:
  title: xdemo
  version: 1.0.0
servers:
- url: https://yyyyyyyy.apigw.yandexcloud.net
paths:
  /:
    get:
      x-yc-apigateway-integration:
        type: cloud_functions
        function_id: xyxyxyxyxyx
        tag: $latest
        service_account_id: xxxxxxxxxxxxx
      responses:
        '200':
          description: Ok
  /login:
    get:
      x-yc-apigateway-integration:
        type: cloud_functions
        function_id: xyxyxyxyxyx
        tag: $latest
        service_account_id: xxxxxxxxxxxxx
      responses:
        '200':
          description: Ok
  /api/{proxy+}:
    get:
```

```
    x-yc-apigateway-integration:
      type: cloud_functions
      function_id: xyxyxyxyxyx
      tag: $latest
      service_account_id: xxxxxxxxxxxxx
    parameters:
    - explode: true
      in: path
      name: proxy
      required: true
      schema:
        type: string
      style: simple
    responses:
      '200':
        description: Ok
```

Листинг 24. Конфигурация шлюза для Express-приложения

Когда процесс создания завершается и шлюз приобретает статус Active, мы можем видеть выделенный выше фрагмент с адресом для подключения

```
servers:
- url: https://yyyyyyyy.apigw.yandexcloud.net
```

Его мы сможем использовать для обращения к функции.

В результате обрабатывается корневой маршрут:

Рисунок 107. Результат запуска функции через шлюз

Маршрут Логина:

Рисунок 108. Выдача ответа в формате JSON

И маршрут API:

Рисунок 109. Срабатывание параметризованного маршрута

На этом упражнение завершено. В формате видеоинструкции эти шаги доступны по адресу: https://kodaktor.ru/ycl-x-2022.mp4

Достаточно часто в приложении Express мы просто по мере реализации наших целей добавляем маршруты. На каком-то этапе мы захотим документировать их, систематизировать информацию о подключённых маршрутах в виде спецификации OpenAPI. Для этого удобно использовать пакет express-oas-generator. Пример вы можете найти по адресу
https://replit.com/@igossoudarev/swagger1#index.js

Минимальный работающий вариант – это строки

```
import expressOasGenerator from 'express-oas-generator';

const app = x();
expressOasGenerator.init(app, {});
```

Рисунок 110. Отображение маршрутов с помощью swagger

При посещении маршрута, который по умолчанию выглядит как /api-docs, модуль генерирует таблицу маршрутов в стиле Swagger, которая содержит описания маршрутов и на основе которой можно пробовать исполнять запросы с разными фактическими параметрами с помощью curl или иных инструментов.

☞ С помощью таких инструментов как Loopback или Hapi вы можете исследовать обратную операцию – генерацию кода обработчиков маршрутов. Благодаря этой возможности разработка серверных веб-приложений на Node приблизилась к возможностям таких инструментов как Yii или Symfony из экосистемы PHP.

### Скаффолдинг

Для создания нового проекта LoopBack необходимо воспользоваться программой командной строки StrongLoop (https://www.npmjs.com/package/strongloop). После глобальной установки пакета strongloop инструментарий командной строки запускается командой slc.

```
npm install -g loopback-cli
```

При генерации используется SDK IBM API Connect, в который входит команда apic для скаффолдинга. Как правило скаффолдинг связывается с кодогенерацией при подключении к базе данных, но упрощённо можно рассматривать его в контексте генерации средств управления данными с помощью API.

LoopBack позволяет создать хранимую в памяти модель – структуру для хранения и изменения данных типа «Пользователь»

```
? Выберите источник данных для подключения к нему
❯ db (memory)
  (no datasource)
```

Рисунок 111. Выбор источника данных при скаффолдинге

и предоставить к ней доступ через API

```
? Выберите базовый класс модели
  (пользовательское)
  Model
❯ PersistedModel
  ACL
  AccessToken
  Application
  Change
```

Рисунок 112. Базовый класс модели в LoopBack

После успешной генерации LoopBack создаёт маршрут Explorer с перечислением сгенерированных API-маршрутов:

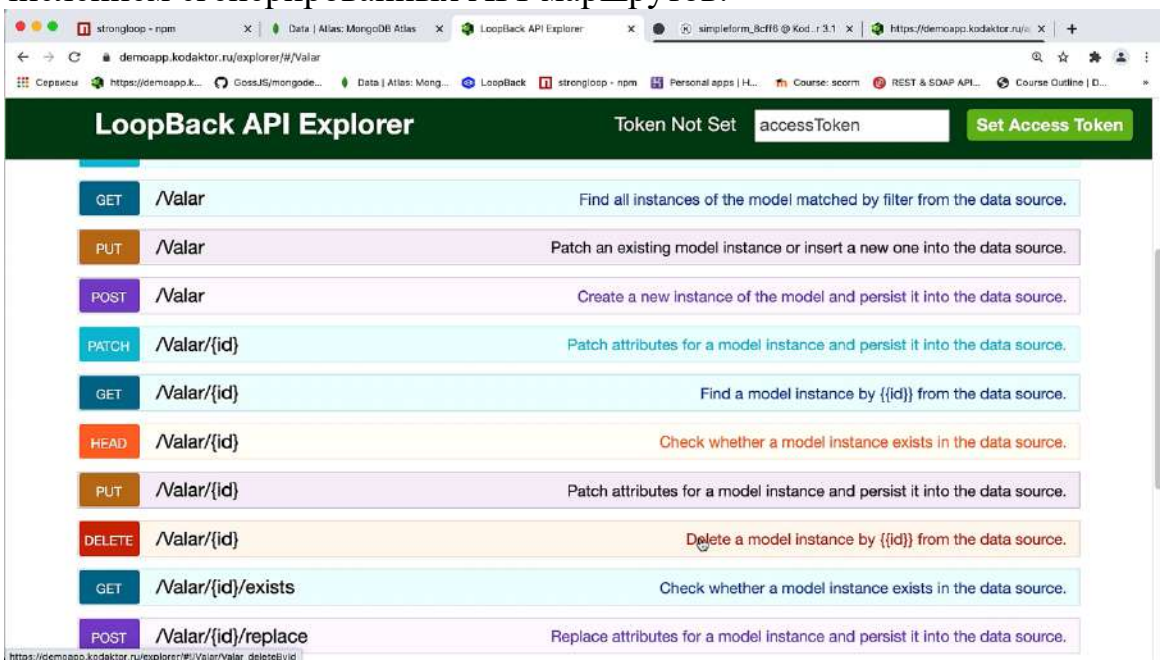


Рисунок 113. Сгенерированные маршруты

В формате видеоинструкции шаги по генерации REST API с помощью Loopback доступны по адресу: https://kodaktor.ru/loopback-short.mp4

***μ***-сервисы/наносервисы предоставляют API, чаще всего REST API для решения конкретных предметных задач. В содержание этого пособия не входит анализ паттерна микросервисов, однако отметим здесь следующее. Микросервисы независимы друг от друга, не обращаются друг к другу напрямую. Одно из главных требований к микросервисам – их доступность друг другу через некоторый стандартный способ (интерфейс), который позволяет скрывать за ним детали реализации. Грубо говоря, чтобы организовать приложение в виде системы микросервисов, нужно ввести некий реестр, в котором будет иметь место сопоставление ключ – значение, где значение – реальный адрес/порт, где работает микросервис. Этот реестр – как фронтконтроллер, который распределяет запросы по нужным исполнителям. Его называют API Gateway. В node-экосистеме с Express связан инструмент под названием Hydra, который позволяет достаточно легко превратить Express-приложение в микросервис, в целом отвечающий требованиям этого паттерна.

### Создание микросервиса на основе проекта Hydra

Рассмотрим простой пример создания сервиса для переворачивания изображения.

По адресу https://kodaktor.ru/flopper находится клиентская часть с выбором изображения, а по адресу https://fork.kodaktor.ru/flopper – бэкэнд.

Эту задачу мы кратко реализуем с помощью Hydra. Согласно официальному описанию, это легковесная библиотека для создания распределенных вычислительных приложений, таких как микросервисы. Преимуществом Hydra является интеграция с Express. Для этой интеграции служит проект HydraExpress.

Следует установить в свою систему на глобальном уровне такой инструмент как Yeoman, который позволяет по заранее подготовленным конфигурациям генерировать код проекта, в который заранее включены необходимые и полезные вещи. Для работы используется такая конфигурация генератора и кроме того, необходим инструмент командной строки.

```
npm i -g yo generator-fwsp-hydra hydra-cli
```

Перед вами пример простого проекта на Hydra, который создает и запускает службу на основе приложения, использующего Express

```
const version = require('./package.json').version;
const hydraExpress = require('hydra-express');
let config = require('fwsp-config');

config
.init('./config.json')
.then(() => {
   const express = hydraExpress.getExpress();
   const api = express.Router();
   api
   .get('/', (req, res) => res
     .status(200)
     .set({"Content-Type":"application/json; charset=utf-8"})
     .end('{"Hello":"world!"}')
   );
   config.version = version;
   return hydraExpress.init(config.getObject(), version, () => {
    hydraExpress.registerRoutes({
      '/v1': api
    });
   });
})
.then(serviceInfo => console.log('serviceInfo', serviceInfo))
.catch(err => console.log('err', err));
```

Листинг 25. Пример простого проекта на Hydra для Express-приложения

По адресу http://kodaktor.ru/x0 содержится пример минималистичного кода для запуска приложения на Express, а по адресу https://kodaktor.ru/service0.js – аналогичный пример приложения, полученного путем обёртывания кода, использующего Express, кодом, использующим hydra-Express.

Hydra выступает своего рода оркестратором, который берёт на себя управление запуском основы микросервиса, которая сделана в данном случае на Express.

Для того, чтобы создать проект с помощью системы генерации по заранее подготовленному шаблону, мы используем команду:

`yo fwsp-hydra`

… которая запускает yeoman, после этого нужно ответить на вопросы генератора.

После завершения этой процедуры нужно не забыть установить зависимости (`npm i` или `yarn`), поскольку генератор отвечает за скаффолдинг – создает лишь структуру проекта и описание зависимости.

Результатом выполнения этих действий оказывается проект.

В config.json указываются параметры, запускаемой службы.

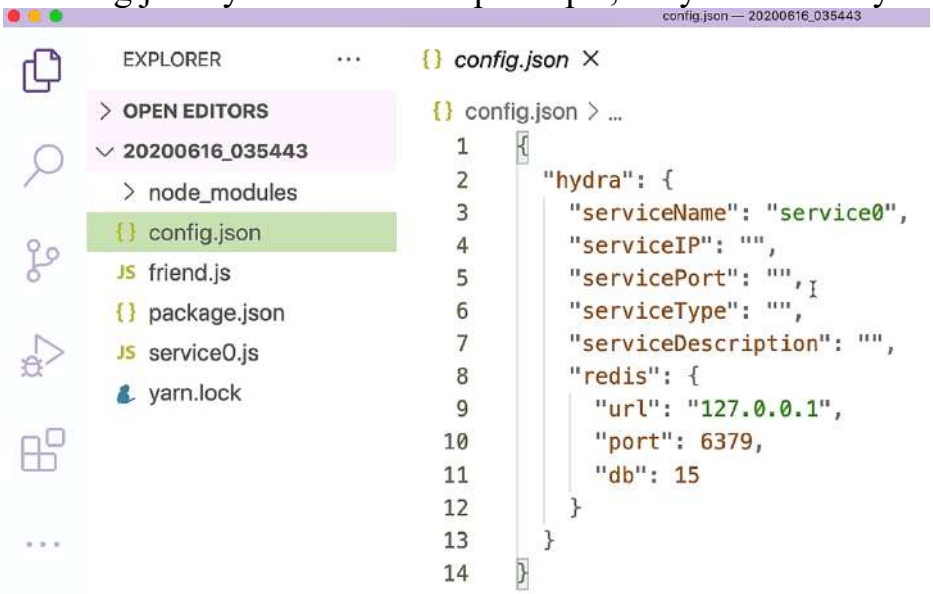


Рисунок 114. Конфигурация службы в Hydra

Нужно заметить, что для работы с микросервисами Hydra требуется система для хранения данных в оперативной памяти, и здесь в роли такой системы выступает Redis. Именно с помощью Redis осуществляется организация сохранения реестра запускаемых служб.

Инструкция

hydraExpress.**init**

инициализирует создание службы на основании данных из конфигурации, присваивает версию и регистрирует службу.

Запустив сценарий, мы можем видеть что Hydra автоматически присвоила порт — это информация из реестра.

```
INFO
{ event: 'info', message: 'Using environment: development' }
serviceInfo {
  serviceName: 'service0',
  serviceIP: '192.168.1.100',
  servicePort: 10190
}
```

Рисунок 115. Автоматическое присвоение порта в Hydra

То есть для каждой службы в реестре будет находиться соответствующая запись.

Для того, чтобы убедиться в том, что происходит отслеживание запущенных служб мы обращаемся к hydra-cli и просим показать запущенные узлы.

hydra-cli nodes

Рисунок 116. Список узлов

Точно так же можно посмотреть зарегистрированные маршруты.

Рисунок 117. Список маршрутов

Службы могут общаться друг с другом благодаря тому, что запускающая их среда hydra создаёт для этого соответствующие условия.

На рисунке 118 показана служба, посылающая сообщение первой службе, созданной выше:

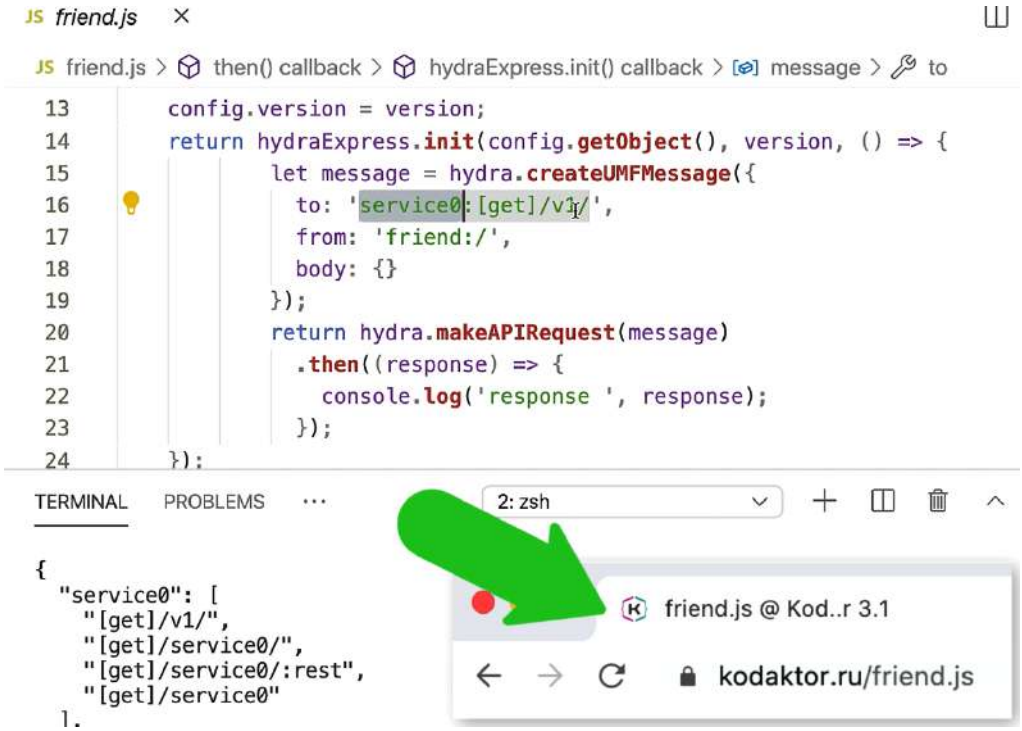


Рисунок 118. Взаимодействие служб

Запускать службы можно сколько угодно раз – создадутся экземпляры, за которыми Hydra будет следить, и если один из них, например, уничтожить, то Hydra уже не будет посылать ему запросы от соседних служб. Таким образом, реализуется обнаружение сервисов.

И наконец, реализуем под управлением Hydra сервис по преобразованию изображений. Склонируем репозиторий, находящийся по адресу https://github.com/GossJS/hydra

После клонирования нужно будет временно – на время работы с этим проектом – перейти к версии, node, которая соответствует этому проекту

`sudo n 10.0.0`

… и запустить инсталляцию зависимости `npm i` или `yarn`

Клиентский код для подключения к этому микросервису на localhost (предполагается, что он будет работать на порту 3000) доступен по адресу https://kodaktor.ru/unsafe_hydra.

Порт явно указан в конфигурационном файле config.json

Рисунок 119. Пример сервиса, созданного с помощью Hydra

Вышеприведённые шаги раскрыты в видеоинструкции, доступной по адресу https://kodaktor.ru/seneca-flopper-hydra.mp4

## Глава 7. HTTPS и HTTP/2

### Локальный сертификат

При разработке на localhost для более близкой имитации происходящего в Интернете можно сделать две вещи:

1. Выпустить SSL-сертификат для локального сервера
2. Создать доменное имя в файле hosts

Для локального выпуска сертификата нам будет нужно установить openssl и понадобятся следующие команды (Linux / MacOS):

1. Сгенерировать приватный ключ:
openssl genrsa -out rootCA.key 2048
2. Сгенерировать по ключу корневой CA сертификат
openssl req -x509 -new -nodes -sha256 -days 1024 -key rootCA.key -out rootCA.pem
3. Создать ключ для дальнейшего создания запроса и выпуска сертификата
openssl genrsa -out server.key 2048
4. Создать конфигурацию (файл openssl-csr.conf) для выпуска wildcard-сертификата на несколько «доменов»
nano openssl-csr.conf

```
[ req ]
default_bits = 4096
req_extensions = req_ext
distinguished_name = req_distinguished_name

[ req_ext ]
```

```
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
subjectAltName = DNS:localhost, DNS:app.localhost, DNS:*.app.localhost

[ req_distinguished_name ]
countryName                 = ru
stateOrProvinceName         = Saint-Petersburg
localityName                = Saint-Petersburg
organizationName            =
commonName                  = kodaktor
```

Листинг 26. Конфигурацию для выпуска wildcard-сертификата

5. Создать запрос (server.csr) для сертификата нескольких доменов

```
openssl req -new -key server.key -config openssl-csr.conf -reqexts req_ext -out server.csr
```

6. Т.к. в нашем случае в одном лице объединены и тот кто посылает запрос, и тот кто его одобряет (certificate authority), то нужно подтвердить запрос ранее созданным корневым сертификатом:

```
openssl x509 -days 500 -sha256 -req -set_serial 01 -extfile openssl-csr.conf -extensions req_ext -in server.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out server.crt
```

Мы получим сертификат, который сможет использовать серверное приложение: server.crt

А корневой сертификат нужно загрузить в операционную систему. Например, в MacOS для этого удобно использовать программу Keychain. Следует запустить Keychain, перейти в раздел сертификатов, выбрать в меню File импортирование и далее указать файл с сертификатом: rootCA.pem

По умолчанию система не доверяет сертификату, и поэтому нужно вручную установить режим доверия:

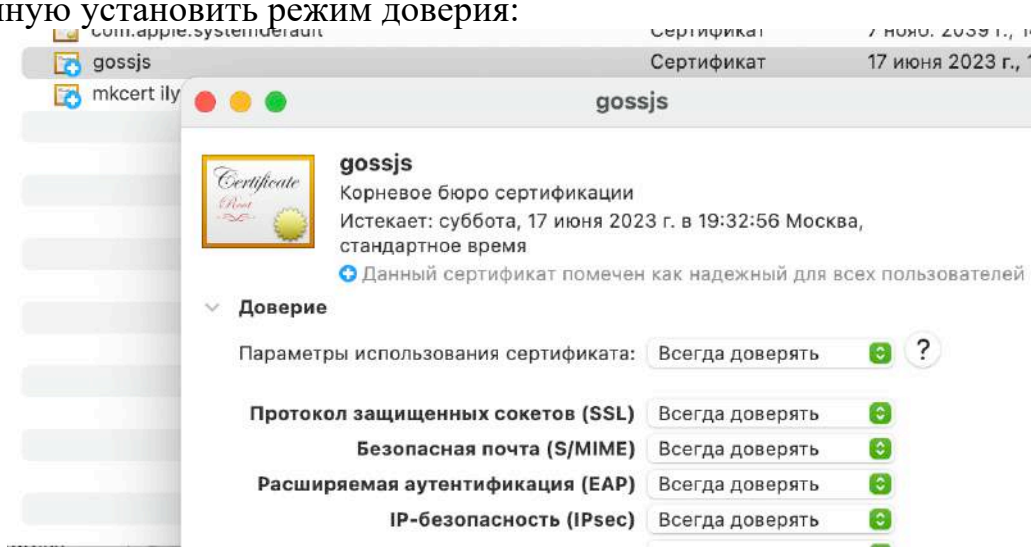


Рисунок 120. Установка доверия сертификату в MacOS Keychain

С другой стороны, браузер тоже по умолчанию не доверяет, поэтому нужно щелкнуть по знаку замка

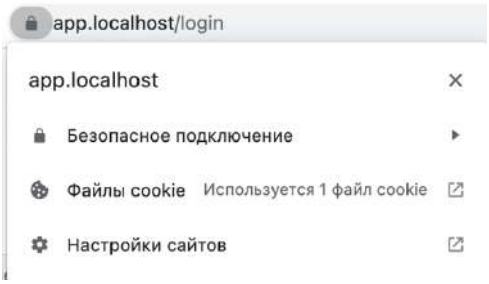


Рисунок 121. Просмотр сведений о сертификате в браузере Chrome

Далее нужно выбрать Безопасное подключение, Действительный сертификат и опять же установить режим доверия:

Рисунок 122. Установка доверия сертификату в браузере Chrome

После этих настроек нам будут доступны сайты такие как https://localhost/ и https://app.localhost/

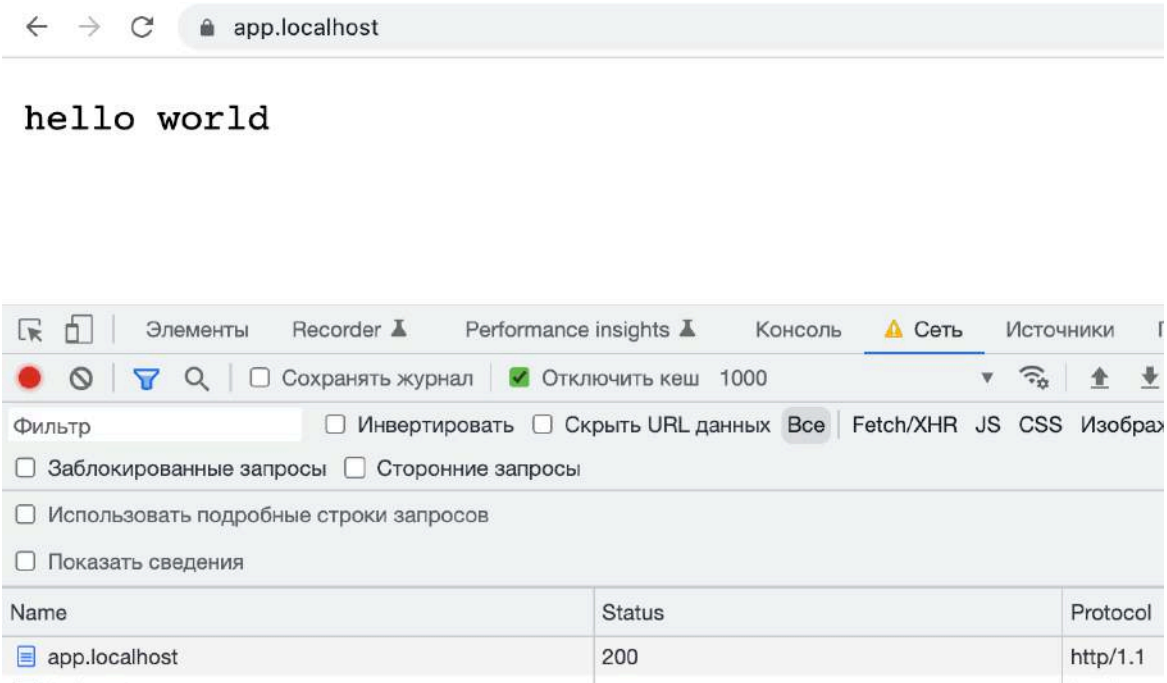


Рисунок 123. Успешная обработка домена второго уровня на localhost

### Модуль node:https

Чтобы создать node-сервер, который обслуживает https-запросы, нужно использовать соответствующий встроенный модуль, в котором функция Server требует указания параметров **key** и **cert**

```
import { Server } from 'node:https';
import { readFileSync } from 'node:fs';

Server({
   key: readFileSync('server.key'),cert: readFileSync('server.crt')
   }, (req, res) => {
   if (req.url == '/login') return res.end('admin');
   if (req.url.startsWith('/deg/') && req.url.split('/').length == 4) {
      const a = req.url.split('/')[2];
      const b = req.url.split('/')[3];
      return res.end(`${a ** b}`);
   }
   res.end('hello world');
})
.listen(443);
```

Листинг 27. Основа запуска HTTPS-сервера с данными сертификата

> **Примечание**
> Что же касается Express, то напрямую объект app не может задействовать сертификат, поэтому мы используем всё тот же Server и передаём ему app в качестве параметра – здесь уместно снова вспомнить о том, что app – это одно большое middleware, обработчик с сигнатурой (req, res) => {}, поэтому мы пишем Server(app).listen. С этой целью в коде базового приложения kodaktor.ru/x затребован модуль http (здесь это должен быть, естественно, https). Аналогично обстоят дела при совместном использовании socket.io и Express

### Контейнер companion

Получение сертификатов для production в единичных случаях можно осуществлять вручную, например с помощью команды certbot решения Let’s Encrypt.

Но хостинги, как правило, обеспечивают выдачу сертификатов как одну из услуг. Тем не менее, существует весьма удобный подход, при котором действия с certbot автоматизируются в соответствии с паттерном Sidecar (Прицеп) и мы можем навесить этот прицеп на наше приложение.

Пусть у нас есть виртуальный сервер с IP-адресом, доступным в Интернете, и домен. Пусть также на сервере установлен Docker.

Зайдём на сервер по ssh и в некоторой папке создаём сценарий с сервером на основе http. В качестве примера можно воспользоваться подготовленными материалами:

```
wget kodaktor.ru/x -O index.js
wget kodaktor.ru/express/x.zip
unzip x.zip
```

В файле index.js меняем require('express') на require('./express') т.к. папка с Express у нас рядом с файлом. Несмотря на то, что на сервере может не быть установлен node, мы при наличии Docker можем использовать контейнер как «открывашку»

```
docker run -d --rm --name gossnodedocker -p 80:4321 -v "$PWD":/usr/src/app -w /usr/src/app node:18 node index.js
```

Здесь мы запускаем контейнер по официальному node-образу и монтируем к его внутренней папке app содержимое нашей текущей папки на сервере. Поскольку серверный сценарий слушает порт 4321, его нужно отобразить на порт внешнего сервера. Контейнер создаётся в отсоединённом (detached) от консоли режиме, чтобы далее в терминале можно было выполнить команды. Контейнеру присваивается имя gossnodedocker и, кроме того, он будет уничтожен сразу после остановки (docker stop).

После этого можно перейти по IP-адресу и убедиться, что приложение работает.

Это одиночный контейнер, демонстрирующий сам факт запуска сценария таким способом. Чтобы создать систему с прицепом, нам нужен образ.

Создадим образ локально. Для этого создадим файл Dockerfile с содержимым:

```
FROM node:14.8.0-alpine3.12

WORKDIR /app
ADD index.js /app/index.js
ADD express /app/express

EXPOSE 4321

CMD node /app/index.js
```

Листинг 28. Dockerfile для образа с Express

Фактически это просто обобщение создания контейнера с помощью некоторого DSL.

Для создания образа по этому файлу запустим команду:

DOCKER_BUILDKIT=1 docker build -t igossoudarev/dockerdemo27082022:latest .

Если после этого выполнить docker images, то мы увидим этот образ в списке.

И теперь мы можем запустить приложение уже изначально из контейнера, т.е. из этого образа (существующего локально)

docker run --name gosdd -d --rm -p 80:4321 igossoudarev/dockerdemo27082022

У контейнера теперь есть имя, мы можем подключиться к нему и зайти в его оболочку

docker exec -it gosdd sh

И наконец финальный шаг: конфигурация системы docker-compose, которая связывает этот наш образ (исходный образ, src) с прицепом

wget kodaktor.ru/express/docker-compose.yml

```
version: '2'

services:
  src:
    container_name: src
    image: igossoudarev/dockerdemo27082022
    environment:
      VIRTUAL_HOST: dockerdemo.kodaktor.ru
      LETSENCRYPT_HOST: dockerdemo.kodaktor.ru
      LETSENCRYPT_EMAIL: admin@dockerdemo.kodaktor.ru

  nginx-proxy:
    image: jwilder/nginx-proxy
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - "/etc/nginx/vhost.d"
      - "/usr/share/nginx/html"
      - "/var/run/docker.sock:/tmp/docker.sock:ro"
      - "/etc/nginx/certs"

  letsencrypt-nginx-proxy-companion:
    image: jrcs/letsencrypt-nginx-proxy-companion
    volumes:
```

```
    - "/var/run/docker.sock:/var/run/docker.sock:ro"
  volumes_from:
    - "nginx-proxy"
```

Листинг 29. Конфигурация docker-compose для реализации «прицепа»

Контейнеры из образов nginx-proxy и letsencrypt-nginx-proxy-companion – это и есть прицеп, который обеспечивает создание reverse proxy-сервера на базе NGING, а также получение SSL-сертификата к нему. Переменные в разделе environment указывают прицепу, для какого домена получать сертификат.

Команда
`docker-compose up`
запускает систему по этой конфигурации и обеспечивает работу приложения по HTTPS:

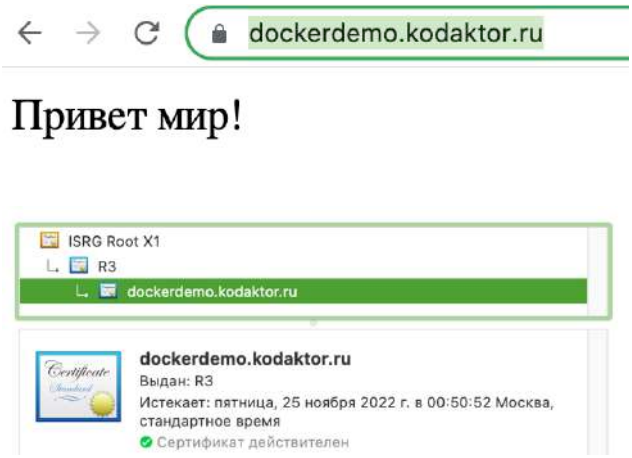


Рисунок 124. Успешная обработка запроса по HTTPS

### Модуль node:http2

Хотя рабочая группа второго поколения HTTP не пришла к единому мнению в отношении необходимости использования шифрования (например, TLS) для HTTP/2, в некоторых реализациях заявлено, что они будут поддерживать HTTP/2 только тогда, когда он используется через зашифрованное соединение, и в настоящее время ни один браузер не поддерживает незашифрованный HTTP/2. Ниже показан простой сервер HTTP/2, использующий Core API. Поскольку неизвестны браузеры, поддерживающие незашифрованный HTTP/2, необходимо использовать http2.createSecureServer() при взаимодействии с браузерами.

```
import { createSecureServer } from 'node:http2';
import { readFileSync } from 'node:fs';

createSecureServer({
  key: readFileSync('server.key'),
  cert: readFileSync('server.crt')
})
.on('error', (err) => console.error(err))
.on('stream', (stream, headers) => {
    stream.respond({ 'content-type': 'text/html; charset=utf-8', ':status': 200 });
    stream.end('<h1>Привет безопасный мир!</h1>');
})
.listen(443);
```

Листинг 30. Основа запуска HTTP/2-сервера с данными сертификата

Здесь используются те же ключ и сертификат, что и в первом пункте этой главы.

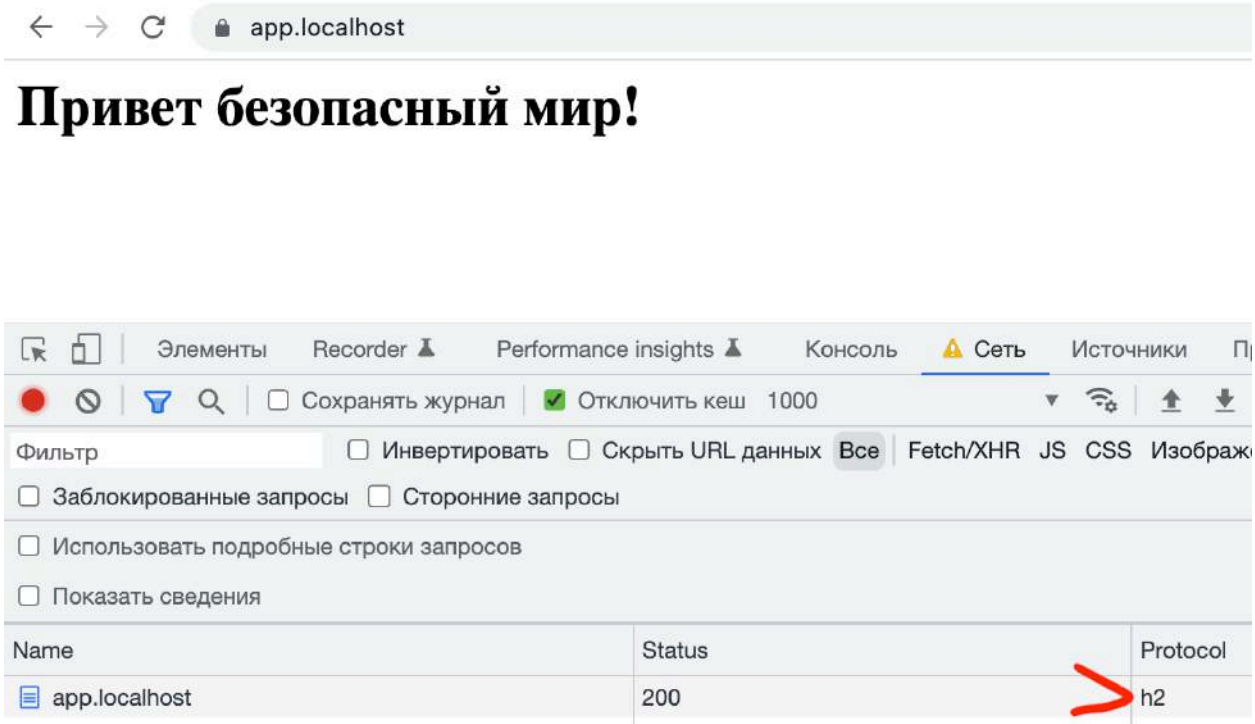


Рисунок 125. Обработка запроса по протоколу HTTP/2 (h2)

Более подробное описание реализации http/2 в свежих версиях Node вы можете найти по адресу https://nodejs.org/dist/latest-v18.x/docs/api/http2.html, а спецификацию – по адресу https://www.rfc-editor.org/rfc/rfc7540

Традиционно HTTP всегда мыслился как протокол запросов и ответов, которые следуют друг за другом. В целом поведение HTTP вряд ли может быть описано как полнодуплексное. Главное же, что в традиционной модели HTTP роли «клиент» и «сервер» однозначны: сервер отвечает клиенту; сервер не говорит когда его не просили. В WebSocket (https://websockets.spec.whatwg.org/#the-websocket-interface) сервер может в любой момент послать данные без запроса.

И HTTP 1.1, и WebSocket – протоколы верхнего (прикладного) уровня и используют в качестве транспорта TCP. WebSocket описывает, как организовать полнодуплексные каналы коммуникации поверх одного TCP-соединения. Порты при этом по умолчанию используются те же самые, что и для HTTP-соединения, т.е. 80 и 443.

## Клиентская реализация WebSocket

В любой момент клиентский сценарий может запустить процесс создания веб-сокетного соединения. Протокол описывает требуемые действия и форматы посылаемых данных, а реализация может быть разной. Клиентская реализация включена в HTML5 API и представлена интерфейсом WebSocket (https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API).

Когда выполняется создание экземпляра класса WebSocket:

```
const sock = new WebSocket('ws://....')
```

… то HTTP-серверу посылается запрос на переход к веб-сокетам:

```
GET / HTTP/1.1
Host: xxx:yyyy
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: Y2SQj….GmadQ==
```

На него сервер должен ответить, обработав событие upgrade:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
```

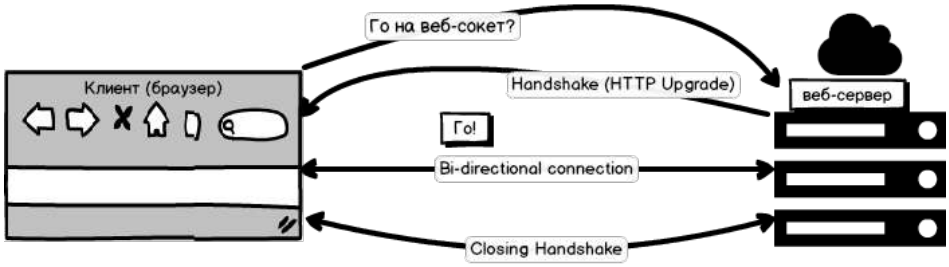


Рисунок 126. Первоначальное рукопожатие при переходе на веб-сокеты

После этого завершается создание соединения, и с именем sock связывается вновь созданный сокет, обладающий событиями open, close, message, на которые можно навесить обработчики, а также позволяющий послать данные с помощью метода send.

Теперь рассмотрим реализацию протокола WebSocket в похожей стилистике с помощью npm-модуля ws

**Упражнение 17**

Изучите изучение сервера и одного клиента, который инициирует общение через веб-сокет:

1. На сервере (в том числе на localhost) в папке инициализируйте проект и получите копию модуля ws (можно установить и через npm/yarn)

```
wget kodaktor.ru/ws.min.zip
```

2. Распакуйте архив так чтобы извлеклась папка ws

```
unzip ws.min.zip
```

3. Перейдите в REPL

Выполните затребование и создайте новый экземпляр сервера

```
const { Server } = require('./ws'); const ws = new Server({ port: 2222 });
```

4. Подпишитесь на сообщения

```
let wsock; void ws.on('connection', _ => { wsock = _; wsock.on('message', d => console.log(String(d)) ||  wsock.send(d)); });
```

5. Перейдите в браузер и откройте консоль. Создайте подключение к серверу

```
const wsock = new WebSocket('ws://node-server.online:2222');
```

6. Подпишитесь на сообщения

```
wsock.onmessage =  ev => console.log(ev.data);
```

7. Пошлите первое сообщение от клиента:

```
wsock.send('hello from chrome');
```

8. Убедитесь, что в REPL это сообщение возникло
Чтобы ответить на него, в REPL напишите:

```
wsock.send('hello to dear chrome!!!!');
```

Можно использовать готовый сценарий http://kodaktor.ru/g/wsclient?node-server.online – здесь после знака вопроса указан конкретный сервер для соединения, если используется localhost, ничего указывать не нужно.

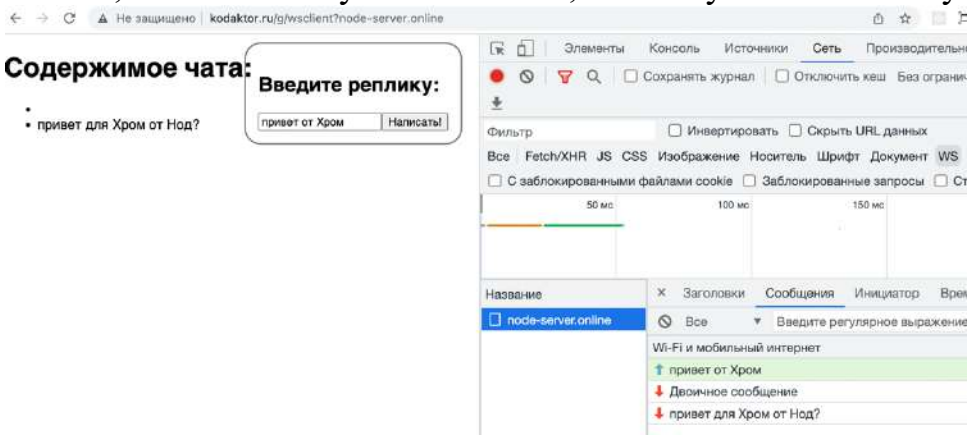


Рисунок 127.  Вкладка Сеть и цепочка реплик на веб-сокетах

И модуль net, и модуль ws (и модули http, https) реализуют клиентскую и серверную функциональность.

| | net | ws |
|---|---|---|
| require | отдаёт объект с такими методами как connect/createConnection и Server/createServer | отдаёт WebSocket с методами send и Server |
| клиентская функциональность | require('net') .connect(2222, '127.0.0.1') | const WebSocket = require('ws'); const ws = new WebSocket('ws://localhost:2222'); |

| | | |
|---|---|---|
| серверная функциональность | см. выше<br>объект net.Server | объект WebSocketServer |
| на вход конструктору | подаются опциональные настройки, связанные с работой потоков | подаётся объект с настройками слушания – просто объект с ключом порт (и другими настройками) или сервер типа экземпляра http.Server |
| первый аргумент коллбэка connection | чистый TCP-сокет<br>с событием data | объект<br>с событием message<br><br>можно написать второй аргумент, это будет req (IncommingMessage) при использовании http.Server |

WebSocketServer – это низкоуровневый инструментарий для менеджмента соединений. Мы получаем возможность отслеживать появление соединения, но что с ним делать дальше – наша забота. Например, если нужно сделать чат, простейшая функциональность для него уже готова. Но дальше нужно уже реализовывать более высокоуровневые возможности.

Чат может быть индивидуальным и групповым. Групповой чат – это некоторая комбинация индивидуальных чатов, позволяющая отправлять сообщения сразу всем или только некоторым. Упражнение выше создаёт один индивидуальный чат, а следующий пример создаёт по чату для каждого подключившегося. Т.е. будет столько объектов wsock, сколько подключилось клиентов. В такой реализации ответ отправляется только тому, который подключился – чтобы реализовать рассылку сообщения типа «Всем привет!» в ws нужно самостоятельно реализовать запоминание клиентов (т.е. иметь простейший мониторинг соединений) и сообщение отправлять в цикле.

☞ См. Упражнение 20 в Главе 9.

Таким образом, обработка данных на клиентской и серверной сторонах почти одна и та же, но всё же не совсем симметрична:

| клиент | сервер |
|---|---|
| создаём веб-сокет, подписываемся на его событие **message** | создаём веб-сокет-сервер, подписываемся на его событие connection<br><br>и уже коллбэк обработчика события connection получает веб-сокет, и мы |

| | подписываемся на его событие **message** |
|---|---|
| .onmessage | .on('message') |

Т.е. главное событие веб-сокета message, несущее передаваемые данные, клиенту доступно сразу, а серверу только в рамках конкретного соединения, что вполне логично и естественно: один клиент – одно соединение, а у сервера их может быть много.

«Родной» объект WebSocket представляет самую базовую функциональность:

```
const ws = new WebSocket('ws://localhost:2222');

ws.onopen = () => console.log(`Соединение установлено`);

ws.onclose = (event) => {
  console.log(event.wasClean ? 'Соединение закрыто чисто' : 'Обрыв соединения');
  console.log(`Код: ${event.code} причина: ${event.reason}`);
};

ws.onmessage = event => console.log(`Получены данные: ${event.data}`);
ws.onerror = error => console.log(`Ошибка: ${error.message}`);
```

Листинг 31. Базовые события WebSocket

Нужно отметить, что event виден телу этих функций не только как свой аргумент, но и глобально (MessageEvent) при наступлении соответствующих событий.

**Упражнение 18**

1. Выполняйте, если по адресу 3333.kodaktor.ru работает сервер kodaktor.ru/3333.vm

(если это не так, можно написать на goss@itmo.ru и попросить включить)

Если сервер включён, переход в браузере или CURL приведёт к ошибке 426

HTTP/1.1 426 Upgrade Required

2. Перейдите по адресу kodaktor.ru/ws и сделайте форк (см. kodaktor.ru/fork)

3. Вставьте клиентский код

4. Замените ?!= на g/ и уберите букву s из протокола (чтобы осталось http)

либо

выберите из правого меню Page no HTTPS (Просмотр без HTTPS) – из адреса (из протокола) будет убрана буква s из протокола (чтобы осталось http) – будет незащищённое соединение

5. Откройте инструменты веб-разработчика – вкладку Сеть
и убедитесь, что видите запрос по протоколу websocket

6. Проверьте, что чат работает с разными клиентами

7. Реализуйте свою версию, позволяющую отправлять картинки и сигнализирующую пользователям о том, что кто-то подсоединился или покинул их. Пусть сообщение отправляется по Enter (а не только при нажатии на кнопку). Подключите другой фреймворк вместо jQuery для обновления списка реплик на странице.

Ещё один вариант клиентского интерфейса – http://kodaktor.ru/g/_ws/_
Он предусматривает возможность выбора подключения по защищённому и незащищённому соединению – можно выбрать ws вместо wss и нажать кнопку «Соединиться»

## socket.io

Мы не можем «из коробки» переустановить соединение или обеспечить гарантию доставки (несмотря на то, что нижележащий протокол TCP гарантирует доставку). Эту более высокоуровневую функциональность реализуют такие библиотеки, как socket.io:

| Клиент | Сервер |
|---|---|
| <script src="/socket.io/socket.io.js"></script><br><br>Подключиться<br>const socket = **io**(); | import { Server as Sio } from 'socket.io';<br><br>Обернуться вокруг сервера sss<br>const io = new **Sio**(sss); |

| | |
|---|---|
| Отреагировать на сообщение<br>`socket.on('chat message', msg => {`<br>`  $('ul').append(`<li>${ msg }</li>`);`<br>`});`<br><br>Отправить сообщение<br>`socket.emit('chat message', text);` | Мониторить соединения и реагировать<br>`io.on('connection', socket => {`<br>`  console.log('a user connected');`<br>`  socket.on('chat message', msg => {`<br>`  io.emit('chat message', msg); // всем вообще`<br>`  });`<br>`});` |
| В этой базовой реализации сервер пересылает каждое сообщение каждого клиента всем включая его самого | |

Что касается гарантии или подтверждения доставки, то эта возможность реализуется указанием коллбэка в дополнение к сигналу и тексту сообщения:

```
socket.emit('message', text, res => console.log('delivered ' + res));
```

Этот коллбэк может быть вызван на клиенте, когда сообщение будет получено.

```
socket.on('message', (msg, cb) => console.log(msg) || cb('from client!'));
```

Это выглядит так, словно функция передаётся по сети и выполняется на другом узле, что напоминает о RPC (и пакете dnode, см. Главу 2).

Библиотека также обеспечивает повторную попытку соединения при разрыве, вводит дополнительные абстракции над основным соединением, такие как комнаты, а также позволяет не беспокоиться о проблеме аутентификации.

**Упражнение 19**

Рассмотрите пример https://g/simpleiochat или в реплите:

- https://replit.com/@igossoudarev/simpleiochat#index.js
- https://replit.com/@igossoudarev/simpleiochat#chat.html

☞ Обратите внимание на то, что app передаётся в качестве аргумента io-серверу (см. примечание к Листингу 27 в Главе 7).

Когда открывается клиентская страница

```
<script src="/socket.io/socket.io.js"></script>
<script>{

  const socket = io();
```

то происходит событие connection

| | | | |
|---|---|---|---|
| jquery.min.js | 200 | h3 | script |
| socket.io.js | 200 | http/1.1 | script |
| ?EIO=4&transport=polling&t=NjSHtGd | 200 | http/1.1 | xhr |
| ?EIO=4&transport=polling&t=NjSHtHD&sid=gACvzkEC7bh8xIh... | 200 | http/1.1 | xhr |
| ?EIO=4&transport=polling&t=NjSHtHE&sid=gACvzkEC7bh8xIh... | 200 | http/1.1 | xhr |
| ?EIO=4&transport=websocket&sid=gACvzkEC7bh8xIhMAAAI | 101 | websocket | websocket |

Рисунок 128. Переход к запросам на веб-сокетах в socket.io

После запуска реплита можно открывать чат на разных компьютерах, испытывать его на прочность и следить за появлением сообщений во вкладке Network/WS в инструментах DevTools: https://simpleiochat.igossoudarev.repl.co/chat

☞ Ознакомьтесь с документацией socket.io и реализуйте чат с несколькими комнатами.

Такие технологии как SSE (Server Side Events) и новые версии HTTP частично перекрывают функциональность веб-сокетов. Но тем не менее они остаются весьма широко используемой технологией. Подробное рассмотрение разных аспектов работы с веб-сокетами см. в учебном пособии [7].

## Глава 9. Использование Deno и Bun

В архиве сайта Node есть ссылка https://github.com/nodejs/node/commit/4cfc982c776475eb65fb1080e6b575a86505a347 на первый коммит в репозиторий (2009.06.11, Version 0.0.3). В 2011 г. создатель Райан Даль уже читал публичные лекции о Node, в 2015 г. имело место временное разделение основной ветки разработки и продукта io.js, но это событие, судя по всему, не имело критического значения.

Эволюция Node включала попытки ускорить и оптимизировать исполнение JavaScript, однако в систему изначально были заложены идеи, которые впоследствии показались создателю ущербными. Соответственно Райан основал новый проект, Deno, который должен был устранить эти недостатки, в числе которых пониженная безопасность, сверхцентрализация системы управления зависимостями, избыточность файла манифеста, огромный размер папки node_modules, отсутствие поддержки TypeScript и другие. Этот проект написан на языке Rust.

### Deno

Помимо устранения недостатков Node, в основу Deno также было положено стремление поддерживать веб-платформу, т.е. все Web API. Более того, «вести себя как браузер», чем и обусловлено использование зависимостей по URL (как в элементе script).

Это проявляется с самого начала – в том как происходит установка.
Чтобы установить Deno, следует выполнить команду
curl -fsSL https://deno.land/x/install/install.sh | sh
На сегодняшний день этому ресурсу вполне можно доверять.

```
######################################################################### 100,0%
Archive:  /Users/ilyagosudarev/.deno/bin/deno.zip
  inflating: /Users/ilyagosudarev/.deno/bin/deno
Deno was installed successfully to /Users/ilyagosudarev/.deno/bin/deno
Manually add the directory to your $HOME/.zshrc (or similar)
  export DENO_INSTALL="/Users/ilyagosudarev/.deno"
  export PATH="$DENO_INSTALL/bin:$PATH"
```

Как и в случае с Node, поддерживается REPL. Однако на момент написания пособия REPL не поддерживал проверку типов TypeScript прямо по ходу набора кода. Пока для такого рода экспериментирования с TypeScript подходит оболочка ts-node:

```
-▶  deno
Deno 1.25.0
exit using ctrl+d or close()
> const n: number = 'abc';
undefined
>
-▶  ts-node
> const n: number = 'abc';
[eval].ts:1:7 - error TS2322: Type 'string' is not assignable to type 'number'.

1 const n: number = 'abc';
        ~
```

Рисунок 129. Подсветка синтаксиса TypeScript в ts-node

До сих пор для установки другой версии Node приходилось использовать сторонний инструмент, такой как volta, nvm или n – в Deno предусмотрен свой инструмент:

```
deno upgrade
Looking up latest version
Found latest version 1.30.0
Checking  https://github.com/denoland/deno/releases/download/v1.25.0/deno-x86_64-apple-darwin.zip
31.6 MiB / 31.6 MiB (100.0%)
Deno is upgrading to version 1.30.0
Archive:
/var/folders/j9/9ywlyl3x6j34nhythk7sfw7h0000gn/T/.tmpq7nxNu/deno.zip
```

```
  inflating: deno
Upgraded successfully
deno --version
deno 1.30.0 (release, x86_64-apple-darwin)
v8 10.6.194.5
typescript 4.7.4
```

Для запуска сценария в отличие от Node используется команда второго уровня (run):

```
deno run index.ts
```

и опять же, следуя «браузерному» подходу, мы можем исполнять сценарии по URL:

```
deno run https://deno.land/std@0.76.0/examples/welcome.ts
```

Вместо process используется объект Deno.

Если выполнить файл с инструкцией

```
console.log(Deno.cwd());
```

то при попытке прочитать текущую папку Deno запросит разрешение – в этом проявляется повышенное внимание к безопасности. Чтобы дать такое разрешение, запуск происходит с типовым флагом:

```
deno run --allow-read index.ts
```

Можно ограничить разрешение: например указать только текущую папку

```
deno run --allow-read=. index.ts
```

… или как противоположный вариант в девелоперском режиме разрешить всё:

```
deno run -A index.ts
```

Управление зависимостями тоже связано с устранением недостатков Node. Мы не используем require в Deno, только import. И при этом можем импортировать как локальные файлы (с обязательным указанием расширения), так и удалённые.

«Удалённые» зависимости — это то, как Deno обрабатывает то, что сегодня в экосистеме Node известно как npm-пакеты. Различие в том, что Deno использует URL-адрес в качестве конструкции верхнего уровня для идентификации модуля, а npm загружает его и использует определенное место — node_modules. Deno поддерживает список одобренных сторонних модулей на своем сайте. В этом проявляются возникшие на этом этапе тенденции централизации.

У Deno также есть «Стандартная библиотека» – модули, которые в Node являются «встроенными».

Например, в Node есть часто используемый модуль node:path. Если вы хотите получить имя файла из пути, то можете использовать этот модуль и метод basename.

```
const path = require('node:path');

console.log(path.basename('/files/folders/folder/file.txt'));
```

В стандартной библиотеке Deno также есть модуль «путь». В Deno точно такой же функционал выглядит так...

```
import * as path from 'https://deno.land/std@0.73.0/path/mod.ts';

console.log(path.basename('/files/folders/folder/file.txt'));
```

При первом обращении удалённые зависимости загружаются в кэш. Мы можем запустить `deno info index.ts` и посмотреть дерево зависимостей, которые подтянул модуль ‘path’ из стандартной библиотеки.

```
file:///home/burkeholland/dev/burkeholland/deno-exercises/exercise/app.ts (128B)
└┬ https://deno.land/std@0.73.0/path/mod.ts (717B)
  ├── https://deno.land/std@0.73.0/path/_constants.ts *
  ├── https://deno.land/std@0.73.0/path/_interface.ts (732B)
  ├┬ https://deno.land/std@0.73.0/path/common.ts (1.14KB)
  │└── https://deno.land/std@0.73.0/path/separator.ts *
  ├┬ https://deno.land/std@0.73.0/path/glob.ts (12.06KB)
  │├── https://deno.land/std@0.73.0/path/_constants.ts *
  │├── https://deno.land/std@0.73.0/path/mod.ts *
  │└┬ https://deno.land/std@0.73.0/path/separator.ts (264B)
  │ └── https://deno.land/std@0.73.0/path/_constants.ts *
  ├┬ https://deno.land/std@0.73.0/path/posix.ts (12.67KB)
  │├── https://deno.land/std@0.73.0/path/_constants.ts *
  │└── https://deno.land/std@0.73.0/path/_util.ts *
  ├── https://deno.land/std@0.73.0/path/separator.ts *
  └┬ https://deno.land/std@0.73.0/path/win32.ts (27.36KB)
   ├── https://deno.land/std@0.73.0/_util/assert.ts (405B)
   ├── https://deno.land/std@0.73.0/path/_constants.ts *
   └┬ https://deno.land/std@0.73.0/path/_util.ts (3.3KB)
    └── https://deno.land/std@0.73.0/path/_constants.ts (2.35KB)
```

Рисунок 130. Дерево зависимостей проекта на Deno

Deno рекомендует выполнять весь импорт в корневом файле deps.ts и экспортировать его из этого единственного файла.

Теперь посмотрим, как создаётся базовый http-сервер в среде Deno

Сначала рассмотрим **низкоуровневый** вариант – это до некоторой степени аналог того что мы делали в Главе 2 с использованием модуля node:net

Чтобы принимать запросы, сначала нужно слушать соединение на порту (Deno.listen). При предоставлении порта Deno предполагает, что мы будем прослушивать TCP-сокет, а также привязываться к 0:0:0:0. Можно явно указать transport: 'tcp', а также указать IP-адрес или имя хоста в свойстве hostname.

Возвращаемое значение Deno.listen() представляет собой асинхронную итерацию (for await of или метод accept), которая создаёт соединения, а также предоставляет методы для обработки соединений.

HTTP-запросы и ответы в Deno, по сути, отражают Fetch API – из него берутся Request и Response, о которых мы говорили в Главе 3.

```
const server = Deno.listen({ port: 4321 });

async function handle(conn: Deno.Conn) {
  const httpConn = Deno.serveHttp(conn);
  for await (const requestEvent of httpConn) {
    await requestEvent.respondWith(new Response("Hello world", { status: 200 }));
  }
}

for await (const conn of server) handle(conn);
```

Листинг 32. Низкоуровневая реализация функциональности http-сервера

Таким образом пример выше запустит сервер, доступный и на localhost, и в локальной сети. Если же указать hostname

```
Deno.listen({ port: 4321, hostname: 'localhost'});
```

… то прослушивание будет привязано только к localhost.

Выполнить этот сценарий (index.ts) можно командой

```
deno run -A index.ts
```

Если не указывать параметр разрешения, Deno спросит о разрешении прямо в CLI – для случая когда указан hostname: localhost и для случая когда указан только порт.

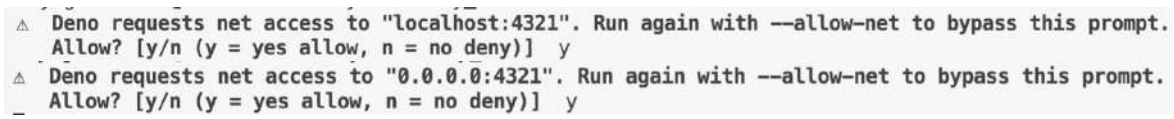


Рисунок 129. Запрос разрешения

(если мы указали hostname:localhost и запустили с параметром --allow-net=0.0.0.0:4321 то всё равно Deno спросит про доступ к localhost:4321)

Чтобы прослушивать 443, мы используем метод listenTls вместо listen и указываем сертификат и ключ в свойствах certFile и keyFile – и здесь нет разницы между http/1.1 и http/2.

Низкоуровневый вариант достаточно сложен и, кроме как в учебных или экспериментальных целях, работать с ним неудобно.

Аналогом использования модуля node:http является обращение к стандартной библиотеке на deno.land

> **Примечание**
> В Интернете можно найти десятки примеров, которые различаются между собой в связи с использованием разных версий Deno и стандартной библиотеки. Ниже в примере указывается текущая референсная версия – std – но можно указать точную версию, например std@0.153.0

> **Примечание**
> Как и в Node, в экосистеме Deno имеется встроенная возможность мониторинга изменений в файле сценария: параметр --watch

```
import { serve } from "https://deno.land/std/http/server.ts";

serve(() => new Response('Hello world!', { status: 200 }), { port: 4321 });
```

При таком разрешении сервер будет доступен и на localhost, и в локальной сети.

Аналогично первому примеру, чтобы ограничить доступность только узлом localhost, это нужно указать в параметрах вызова функции serve:

```
{ port: 4321, hostname: 'localhost'}
```

Как и в случае Node, работа с http тоже достаточно низкоуровневая, например, чтобы распознать параметр из Query String придётся извлекать его самостоятельно, для этого можно использовать URLSearchParams из Web API:

```
serve((req: Request): Response => {
   const searchParameters = new URLSearchParams(req.url.split('?')[1]);
   const name = searchParameters.get('name');
   return new Response(`Hello, ${name}!`, { status: 200 });
}, { port: 4321 });              /* http get "localhost:4321/hello?name=Ilya" */
```

И так далее: очевидно, лучше сразу перейти к более высокоуровневой библиотеке или фреймворку. Поскольку мы уже решили в Главе 5, что будем рассматривать Express как референсную технологию, то здесь возникает вопрос, что из экосистемы Deno годится для сравнения с Express. Примером такого решения является oak.

Синтаксис Oak похож на синтаксис Express, хотя взаимоотношения с middleware и способ выдачи ответа, императивный по форме, больше напоминают Koa.

Вот пример минимального приложения Oak, которое воспроизводит базовую функциональность Express и соответствует первым примерам из Главы 5.

```
import { Application, Router } from 'https://deno.land/x/oak/mod.ts';

const app = new Application();
const router = new Router();

router.get('/', context => {
  context.response.body = 'Привет мир!';
});
app.use(router.routes());

await app.listen({ port: 4321 });
```

Листинг 33. Реализация базовых возможностей Express на Oak

Проект Oak приобрёл определённую популярность, а другие проекты достаточно нестабильны, но, тем не менее для придания некоторый завершенности и полноты этому обсуждению мы упомянем здесь ещё один проект, который значительно больше похож на Express поскольку его целью и было перенести Express на Deno; это проект Opine. Структура базового приложения выглядит крайне знакомо:

```
import { opine as x } from 'https://deno.land/x/opine@1.9.1/mod.ts';

const app = x();
const PORT = 4321;
app
.get('/', (_req, res) => res.send('Привет мир!'))
.listen(PORT);
```

Листинг 34. Реализация базовых возможностей Express на Opine

**Примечание**

на TypeScript работа с Express может быть организована примерно так:

```
import express, { Application, Request, Response } from 'express';

const app: Application = express();

const PORT: number = 4321;

app.get('/', (req: Request, res: Response) => {
    debugger;
    res.send('Elias Goss typing..!');
});
app.listen(PORT);
```

Кроме того, описания типов доступны в пакете @types/express

В отличие от Node, Deno содержит в том числе поддержку глобального имени WebSocket.

**Упражнение 20**

Это продолжение упражнений по веб-сокетам, но в данном случае мы поддерживаем связь с каким угодно количеством подключившихся клиентов, и один из клиентов будет на Deno:

Вот код сервера:

```
const { Server } = require('./ws'); const ws = new Server({ port: 2222 });

const conns = new Map();
void ws.on('connection', sock => {
  sock
    .on('message', d => sock.send(d) || console.log(String(d)))
    .on('close', () => conns.delete(sock));
  conns.set(sock, new Date().toLocaleString());
});
```

Запустив Deno, создадим подключение в терминале:

```
ilyagosudarev@MacBook-Pro-Ilya ~ % deno
Deno 1.25.1
exit using ctrl+d or close()
> const wsock = new WebSocket('ws://node-server.online:2222')
undefined
> wsock.onmessage =  ev => console.log(ev.data);
[Function]
> wsock.send('hello from deno')
undefined
> Blob { size: 15, type: "" }
hey deno
```

И ещё одно подключение в браузере (в консоли или http://kodaktor.ru/g/wsclient)

После того как каждый из клиентов послал своё сообщение в чат, в ассоциативном массиве cons образовалось два подключения.

На сервере мы можем написать

```
conns.forEach(x => console.log(x))
```

… и увидеть список из двух дат. Первая дата соответствует ручному терминальному подключению из Deno, вторая – браузерному.

```
[...conns.keys()][0].send('hey terminal');
```

– так мы сможем отправить сообщение в сокет, представляющий соединение, открытое из терминала вручную.

```
[...conns.keys()][1].send('hey browser);
```

– это отправит сообщение в браузер. Так будет работать, если действительно сначала мы запустили подключение из командной строки, а потом открыли страницу в браузере.

Если же мы хотим создать широковещательную рассылку каждого приходящего сообщения всем подключённым клиентам, то:

```
for (let c of conns) c[0].send('!');
 или
for (let c of conns.keys()) c.send('!');
```

Не следует смешивать Deno с безголовым браузерным движком. Такие решения как Zombie, Puppeteer или Playwright представляют собой автономный автоматизируемый браузер и, безусловно, предоставляют поддержку парсинга HTML и операций с DOM из коробки. Чтобы осуществлять всё это и в Node и в Deno придётся использовать специальные библиотеки типа jsdom, deno_dom или cheerio.

Мы начинали в Главе 1 с примера обращения к интернет-ресурсу с целью прочитать из него текущее время. Это задача веб-парсинга. Вот решение на Deno:

```
const myURL = 'https://www.worldtimeserver.com/current_time_in_GB.aspx';

const html = await ((await fetch (myURL)).text());
const doc = new DOMParser().parseFromString(html, 'text/html');  // doc: any

console.log(doc.querySelector('#theTime').textContent);
```

Листинг 35. Изоморфная deno-реализация извлечения данных из HTML-страницы

Этот код будет работать и в браузере.

Сравните это решение с решением на Node с помощью Puppeteer

```
import puppeteer from 'puppeteer';

const myURL = 'https://www.worldtimeserver.com/current_time_in_GB.aspx';

const browser = await puppeteer.launch({ headless: true, args:['--no-sandbox'] });
const page = await browser.newPage();

await page.goto(myURL);

const selector = '#theTime';
await page.waitForSelector(selector);

const response = await page.$eval(selector, el => el.textContent);
browser.close();
console.log(response);
```

Листинг 36. Извлечение данных из HTML-страницы безголовым браузером

Относительно недавно (фактически на финальных этапах работы над этим пособиям) стало известно о новым фреймворке для Deno – https://fresh.deno.dev/ – функциональность которого находится где-то между Next.js и Nest.js – используется, разумеется, TypeScript, для описания UI – библиотека Preact, серверный рендеринг и маршрутизация на основе расположения файлов в папках. Возможно, появление такого инструментария ускорит процесс по замещению Node на Deno.

Целью этого обзора в Главе 9 было сформировать общее представление о Deno, поэтому мы не будем углубляться в детали и предоставляем читателю дальнейшие исследования. Единственное, что стоит ещё отметить – Deno

содержит команду compile, которая в будущем, вероятно, сможет любой проект превращать в единственный исполнимый файл (в зависимости от платформы), что должно сильно упростить деплой. Поэтому за этой возможностью есть смысл следить.

### Bun

Новая система, Bun, претендует на то, чтобы стать сверхбыстрым компилятором JavaScript в таких проектах как React или Next.js.

☞ Вы можете установить Bun с помощью команды curl -fsSL https://bun.sh/install | bash если доверяете этому источнику. Если не очень доверяете, то пока эксперименты можно проводить в докер-контейнере:

docker run --init --ulimit memlock=-1:-1 jarredsumner/bun:edge tail -f /dev/null

После запуска контейнера и получения его ID, например a8729c78e808, можно подключиться к нему и исследовать изнутри, в том числе выполняя команды из документации:

docker exec -it a8729c78e808 sh

Контейнер с bun можно использовать в качестве «открывашки» (мы обсуждали это в Главе 7). Например, можно создать в текущей папке файл index.js с кодом на JavaScript, а затем выполнить команду:

docker run --init --ulimit memlock=-1:-1 -v "$PWD":/usr/src/app -w /usr/src/app jarredsumner/bun:edge bun run index.js

В bun используется не V8 (в отличие от Node/Deno), а движок JavaScript Core, который является основой Safari и может быть запущен в MacOS командой

/System/Library/Frameworks/JavaScriptCore.framework/Versions/Current/Helpers/jsc

Bun был написан на языке Zig, который конкурируют с Rust. Заявлена поддержка WebSocket, Fetch, ReadableStream, а также собственный менеджер зависимостей и ещё ряд характеристик, за развитием которых будет интересно понаблюдать

## Заключение

Объём пособия и нежелание умножать неактуальные сущности удержали автора от включения в текст очень многих интересных вещей, но, к счастью, ежегодно публикуются книги, в которых на современных примерах рассматривается большинство важных аспектов веб-разработки.

Так, в книге [8] вы найдёте пример современного целостного взгляда на разработку с использованием JavaScript на разных уровнях стека.

В пособии не рассматривались некоторые темы, раскрытые или затронутые в онлайн-курсе (аутентификация и подключение базы данных, MVC, тестирование, CI/CD) в силу их самостоятельной сложности, требующей отдельного внимания. Но автор надеется вернуться к ним в следующих изданиях.

В пособии обильно использована полусленговая терминология разработчиков (коллбэк, бэкэнд, промисифицировать, задеплоить), которая в основном сводится к тому, чтобы записывать русскими буквами английские слова с небольшими добавлениями. Там, где это неудобно (middleware, production) или пока не удаётся подобрать адекватное соответствие (stateless), использованы оригинальные термины с пояснениями. Как представляется, встречая название фреймворка или инструмента, желательно поинтересоваться его переводом на русский язык (если таковой существует) – часто это проясняет специфику применения. Не исключено, что в будущем разработчики смогут полностью перейти на JavaScript не только для программирования, но и для общения =)

Автор благодарит студентов образовательной программы магистратуры «Веб-технологии», вопросы и внимание которых стимулировали разработку контента онлайн-курса и данного пособия.

Примите пожелания успеха в освоении веб-разработки и экосистемы, связанной с языком JavaScript! По возникающим вопросам пишите по адресу goss@itmo.ru

## Вопросы для самопроверки

**1.** Сервер на Express запущен так: `require('express')(3333).use((r1, r2, next, a = 5555) => next(42)).get('*', (req, res) => res.send('OK')).listen(8888);`

Какой ответ будет отослан браузеру либо иному клиенту по корневому маршруту?

1. 42
2. OK
3. 302
4. 500

**2.** Сервер на Express запущен так: `require('express')().get('*', (req, res) => res.write('OK')).listen(8888);`

Что нужно указать в качестве параметра после curl localhost:8888 чтобы получить ответ OK по прошествии как минимум 2000 миллисекунд после подачи команды?

1. --max-time 2
2. --max-time 2000
3. -t 2000
4. -t 2

**3**. Какой код ответа сервера обозначает успешный результат создания нового ресурса?

1. 201
2. 200
3. 400
4. 500

**4**. Какой объект в Fetch API предоставляет доступ к потоку?

1. Response.body
2. Request.stream
3. Response.stream
4. Response.thread

**5.** В ответ на запрос в приложении на Express запрограммирован такой ответ: `res.end('a') && res.writeHead(200, {b: 1})`

Какой код ошибки мы увидим при попытке подачи такого запроса? (он начинается с ERR_)

**6**. Какой термин ассоциируется с кодогенерацией для доступа к базам данных?

1. скаффолдинг
2. маршаллинг
3. сериализация
4. стрингификация

**7**. Какой шаблон (паттерн) шаблонизации используется при заполнении HTML-шаблона информацией с помощью маркеров, указанных в шаблоне?

1. TemplateView
2. TransformView
3. BoilerplateView
4. BlueprintView

**8.** Сервер на Express запущен так: `require('express')().get('*', (req, res) => res.write('OK')).listen(8888);`

Каким будет значение заголовка ответа Transfer-Encoding?

**9.** В ответ на запрос в приложении на Express запрограммирован такой ответ: `res.end('a') && res.writeHead({b: 1})`

Какого типа ошибка будет брошена при подаче такого запроса?

1. RangeError
2. TypeError
3. SyntaxError
4. BadError

**10.** Когда возвращаемый сервисом результат оформляется в виде текста либо JSON либо используется иной тип контента, то согласно REST это

1. представление
2. идентификатор
3. идемпотентность
4. endpoint

### ☞ Задание «Код http-сервера»

Перейдите по адресу
https://node-server.online/r?id=.#student|20

Все данные кроме ответа уже введены.

Пусть функция task(***x***) возвращает экземпляр http-сервера, созданный с помощью встроенного Node.js-модуля http. Этот сервер не должен быть запущен / слушать какой-либо порт. Используйте только res.end (не нужны заголовки или теги HTML). Он должен обрабатывать два маршрута: (1) /challenge – в ответ на такой запрос он должен возвращать ***x*** (2) /api/rv/abc где abc – произвольная строка длиной не менее 1 символа, состоящая только из строчных латинских букв. В ответ на такой запрос сервер должен возвращать перевёрнутую строку. Например: https://kodaktor.ru/api/rv/abc Для других маршрутов предусмотрен ответ No (Не нужно делать require) В последнем-нижнем поле напишите код этой функции task(x).

### ☞ Задание «Адрес простого https-сервера с параметризованным маршрутом»

Перейдите по адресу
https://node-server.online/r?id=javascripter#student|59

Все данные кроме ответа уже введены.

Напишите корневой адрес веб-приложения, которое по маршруту /login выдаёт ваш временный логин (в данном случае слово javascripter), а по маршруту /deg/{x1}/{x2} выдаёт числовой результат возведения числа x1 в степень x2.

**☞ Задание «Адрес приложения, вычисляющего размеры изображения»**

Перейдите по адресу
https://node-server.online/r?id=nodejsworker#student|84
Все данные кроме ответа уже введены.

Напишите корневой адрес работающего по https веб-приложения, которое по маршруту /login выдаёт ваш временный логин (в данном случае слово nodejsworker)

и по маршруту /size2json получает по имени поля image в формате multipart/form-data изображение в формате PNG и выдаёт JSON-строку вида {"width":123,"height":456}, содержащую, соответственно, ширину и высоту изображения в пикселях.

1. Флэнаган, Дэвид. JavaScript. Полное руководство, 7-е изд. : Пер. с англ. — СПб. : ООО “Диалектика”, 2021. — 720 с . : ил. — Парал. тит. англ. ISBN 978-5-907203-79-2
2. Государев, И. Б. Введение в веб-разработку на языке JavaScript : учебное пособие / И. Б. Государев. — Санкт-Петербург : Лань, 2022. — 144 с. — ISBN 978-5-8114-3539-5. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/206588 (дата обращения: 28.08.2022). — Режим доступа: для авториз. пользователей.
3. Пауэрс Ш. Изучаем Node. Переходим на сторону сервера. 2-е изд., доп. и перераб. — СПб.: Питер, 2017. — 304 с.: ил. — (Серия «Бестселлеры O’Reilly»). ISBN 978-5-496-02941-4
4. Каскиаро М., Маммино Л. Шаблоны проектирования Node.js / пер. с анг. А. Н. Киселева. – М.: ДМК Пресс, 2017. – 396 с.: ил
5. Хантер II Т., Инглиш Б. Многопоточный JavaScript / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2022. – 188 с.
6. Браун И.Веб-разработка с применением Node и Express. Полноценное использование стека JavaScript. 2-е издание. — СПб.: Питер, 2021. — 336 с. ISBN 978-5-4461-0590-8
7. Хабаров, С. П. Построение распределенных систем на базе WebSocket : учебное пособие для спо / С. П. Хабаров, М. Л. Шилкина. — Санкт-Петербург : Лань, 2021. — 216 с. — ISBN 978-5-8114-6967-3. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/153930 (дата обращения: 28.08.2022). — Режим доступа: для авториз. пользователей.
8. Скотт Адам Д. Разработка на JavaScript. Построение кроссплатформенных приложений с помощью GraphQL, React, React Native и Electron. — СПб.: Питер, 2021. — 320 с.: ил. — (Серия «Бестселлеры O’Reilly»). ISBN 978-5-4461-1462-7

Государев Илья Борисович

# Основы разработки веб-приложений на платформах Node.js и Deno

**Учебно-методическое пособие**

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н.Ф. Гусарова
Подписано к печати
Заказ №
Тираж
Отпечатано на ризографе

**Редакционно-издательский отдел**
**Университета ИТМО**
197101, Санкт-Петербург, Кронверкский пр., 49, литер А