

Руководство по Node.js



Перевод опубликован в [блоге](#) компании RUVDS. [Оригинал](#) статей.

Читать онлайн (много полезных комментариев):

- Часть 1: [Общие сведения и начало работы](#)
- Часть 2: [JavaScript, V8, некоторые приёмы разработки](#)
- Часть 3: [Хостинг, REPL, работа с консолью, модули](#)
- Часть 4: [npm, файлы package.json и package-lock.json](#)
- Часть 5: [npm и прх](#)
- Часть 6: [цикл событий, стек вызовов, таймеры](#)
- Часть 7: [асинхронное программирование](#)
- Часть 8: [протоколы HTTP и WebSocket](#)
- Часть 9: [работа с файловой системой](#)
- Часть 10: [стандартные модули, потоки, базы данных, NODE_ENV](#)

Оглавление

Часть 1: общие сведения и начало работы

Обзор Node.js

Скорость

Простота

JavaScript

Движок V8

Асинхронность

Библиотеки

Установка Node.js

Первое Node.js-приложение

Фреймворки и вспомогательные инструменты для Node.js

Краткая история Node.js

2009

2010

2011

2012

2013

2014

2015

2016

2017

2018

Часть 2: JavaScript, V8, некоторые приёмы разработки

Какими JS-знаниями нужно обладать для Node.js-разработки?

Различия между платформой Node.js и браузером

V8 и другие JavaScript-движки

Разработка JS-движков и стремление к производительности

Интерпретация и компиляция

Выход из Node.js-приложения

Чтение переменных окружения из Node.js

Часть 3: хостинг, REPL, работа с консолью, модули

Хостинг для Node.js-приложений

Самый простой вариант хостинга: локальный туннель

Среды для развёртывания Node.js-проектов, не требующие настройки

Glitch

Codepen

Бессерверные среды

PAAS-решения

Zeit Now

Nanobox

Heroku

Microsoft Azure

Платформа Google Cloud

VPS-хостинг

[Обычный сервер](#)

[Использование Node.js в режиме REPL](#)

[Автозавершение команд с помощью клавиши Tab](#)

[Исследование объектов JavaScript](#)

[Исследование глобальных объектов](#)

[Специальная переменная](#)

[Команды, начинающиеся с точки](#)

[Работа с аргументами командной строки в Node.js-скриптах](#)

[Вывод данных в консоль с использованием модуля console](#)

[Очистка консоли](#)

[Подсчёт элементов](#)

[Вывод в консоль результатов трассировки стека](#)

[Измерение времени, затраченного на выполнение некоего действия](#)

[Работа с stdout и stderr](#)

[Использование цвета при выводе данных в консоль](#)

[Создание индикатора выполнения операции](#)

[Приём пользовательского ввода из командной строки](#)

[Система модулей Node.js, использование команды exports](#)

[Часть 4: npm, файлы package.json и package-lock.json](#)

[Основы npm](#)

[Загрузка пакетов](#)

[Установка всех зависимостей проекта](#)

[Установка отдельного пакета](#)

[Обновление пакетов](#)

[Загрузка пакетов определённых версий](#)

[Запуск скриптов](#)

[Куда npm устанавливает пакеты?](#)

[Использование и выполнение пакетов, установленных с помощью npm](#)

[Файл package.json](#)

[Структура файла](#)

[Свойства, используемые в package.json](#)

[Свойство name](#)

[Свойство author](#)

[Свойство contributors](#)

[Свойство bugs](#)

[Свойство homepage](#)

[Свойство version](#)

[Свойство license](#)

[Свойство keywords](#)

[Свойство description](#)

[Свойство repository](#)

[Свойство main](#)

[Свойство private](#)

[Свойство scripts](#)

[Свойство dependencies](#)

[Свойство devDependencies](#)

[Свойство engines](#)

[Свойство browserlist](#)

[Хранение в package.json настроек для различных программных инструментов](#)

[О версиях пакетов и семантическом версионировании](#)

[Файл package-lock.json](#)

[Пример файла package-lock.json](#)

[Часть 5: npm и npx](#)

[Выяснение версий установленных npm-пакетов](#)

[Установка старых версий npm-пакетов](#)

[Обновление зависимостей проекта до их самых свежих версий](#)

[Локальная или глобальная деинсталляция пакетов](#)

[О выборе между глобальной и локальной установкой пакетов](#)

[О зависимостях проектов](#)

[Утилита npx](#)

[Использование npx для упрощения запуска локальных команд](#)

[Выполнение утилит без необходимости их установки](#)

[Запуск JavaScript-кода с использованием различных версий Node.js](#)

[Запуск произвольных фрагментов кода, доступных по некоему адресу](#)

[Часть 6: цикл событий, стек вызовов, таймеры](#)

[Цикл событий](#)

[Блокировка цикла событий](#)

[Стек вызовов](#)

[Цикл событий и стек вызовов](#)

[Постановка функции в очередь на выполнение](#)

[Очередь событий](#)

[Очередь заданий ES6](#)

[process.nextTick\(\)](#)

[setImmediate\(\)](#)

[Таймеры](#)

[Функция setTimeout\(\)](#)

[Нулевая задержка](#)

[Функция setInterval\(\)](#)

[Рекурсивная установка setTimeout\(\)](#)

[Часть 7: асинхронное программирование](#)

[Асинхронность в языках программирования](#)

[Асинхронность в JavaScript](#)

[Коллбэки](#)

[Обработка ошибок в коллбэках](#)

[Проблема коллбэков](#)

[Промисы и async/await](#)

[Промисы](#)

[Знакомство с промисами](#)

[Как работают промисы](#)

[Создание промисов](#)

[Работа с промисами](#)

[Объединение промисов в цепочки](#)

[Обработка ошибок](#)

[Каскадная обработка ошибок](#)

[Promise.all\(\)](#)

[Promise.race\(\)](#)

[Об ошибке Uncaught TypeError, которая встречается при работе с промисами](#)

[Конструкция async/await](#)

[Как работает конструкция async/await](#)

[О промисах и асинхронных функциях](#)

[Сильные стороны async/await](#)

[Использование последовательностей из асинхронных функций](#)

[Упрощённая отладка](#)

[Генерирование событий в Node.js](#)

[Часть 8: протоколы HTTP и WebSocket](#)

[Что происходит при выполнении HTTP-запросов?](#)

[Протокол HTTP](#)

[Фаза DNS-поиска](#)

[Функция gethostbyname](#)

[Установление TCP-соединения](#)

[Отправка запроса](#)

[Строка запроса](#)

[Заголовок запроса](#)

[Тело запроса](#)

[Ответ](#)

[Разбор HTML-кода](#)

[О создании простого сервера средствами Node.js](#)

[Выполнение HTTP-запросов средствами Node.js](#)

[Выполнение GET-запросов](#)

[Выполнение POST-запроса](#)

[Выполнение PUT-запросов и DELETE-запросов](#)

[Выполнение HTTP-запросов в Node.js с использованием библиотеки Axios](#)

[Установка](#)

[API Axios](#)

[Запросы GET](#)

[Использование параметров в GET-запросах](#)

[Запросы POST](#)

[Использование протокола WebSocket в Node.js](#)

[Отличия от HTTP](#)

[Защищённая версия протокола WebSocket](#)

[Создание WebSocket-соединения](#)

[Отправка данных на сервер](#)

[Получение данных с сервера](#)

[Реализация WebSocket-сервера в среде Node.js](#)

[Часть 9: работа с файловой системой](#)

[Работа с файловыми дескрипторами в Node.js](#)

[Данные о файлах](#)

[Пути к файлам в Node.js и модуль path](#)

[Получение информации о пути к файлу](#)

[Работа с путями к файлам](#)

[Чтение файлов в Node.js](#)

[Запись файлов в Node.js](#)

[Присоединение данных к файлу](#)

[Об использовании потоков](#)

[Работа с директориями в Node.js](#)

[Проверка существования папки](#)

[Создание новой папки](#)

[Чтение содержимого папки](#)

[Переименование папки](#)

[Удаление папки](#)

[Модуль fs](#)

[Модуль path](#)

[path.basename\(\)](#)

[path.dirname\(\)](#)

[path.extname\(\)](#)

[path.isAbsolute\(\)](#)

[path.join\(\)](#)

[path.normalize\(\)](#)

[path.parse\(\)](#)

[path.relative\(\)](#)

[path.resolve\(\)](#)

[Часть 10: стандартные модули, потоки, базы данных, NODE_ENV](#)

[Модуль Node.js os](#)

[os.arch\(\)](#)

[os.cpus\(\)](#)

[os.endianness\(\)](#)

[os.freemem\(\)](#)

[os.homedir\(\)](#)

[os.hostname\(\)](#)

[os.loadavg\(\)](#)

[os.networkInterfaces\(\)](#)

[os.platform\(\)](#)

[os.release\(\)](#)

[os.tmpdir\(\)](#)

[os.totalmem\(\)](#)

[os.type\(\)](#)

[os.uptime\(\)](#)

[Модуль Node.js events](#)

[emitter.addListener\(\)](#)

[emitter.emit\(\)](#)

[emitter.eventNames\(\)](#)

[emitter.getMaxListeners\(\)](#)

[emitter.listenerCount\(\)](#)

[emitter.listeners\(\)](#)

[emitter.off\(\)](#)

[emitter.on\(\)](#)

- [emitter.once\(\)](#)
- [emitter.prependListener\(\)](#)
- [emitter.prependOnceListener\(\)](#)
- [emitter.removeAllListeners\(\)](#)
- [emitter.removeListener\(\)](#)
- [emitter.setMaxListeners\(\)](#)

[Модуль Node.js http](#)

[Свойства](#)

- [http.METHODS](#)
- [http.STATUS_CODES](#)
- [http.globalAgent](#)

[Методы](#)

- [http.createServer\(\)](#)
- [http.request\(\)](#)
- [http.get\(\)](#)

[Классы](#)

- [http.Agent](#)
- [http.ClientRequest](#)
- [http.Server](#)
- [http.ServerResponse](#)
- [http.IncomingMessage](#)

[Работа с потоками в Node.js](#)

- [О сильных сторонах использования потоков](#)

- [Пример работы с потоками](#)

- [Метод pipe\(\)](#)

- [API Node.js, в которых используются потоки](#)

- [Разные типы потоков](#)

- [Создание потока для чтения](#)

- [Создание потока для записи](#)

- [Получение данных из потока для чтения](#)

- [Отправка данных в поток для записи](#)

- [Сообщение потоку для записи о том, что запись данных завершена](#)

[Основы работы с MySQL в Node.js](#)

- [Установка пакета](#)

- [Инициализация подключения к базе данных](#)

- [Параметры соединения](#)

- [Выполнение запроса SELECT](#)

- [Выполнение запроса INSERT](#)

- [Закрытие соединения с базой данных](#)

- [О разнице между средой разработки и продакшн-средой](#)

Часть 1: общие сведения и начало работы

Обзор Node.js

Node.js — это open-source кроссплатформенная среда выполнения для JavaScript, которая работает на серверах. С момента выпуска этой платформы в 2009 году она стала чрезвычайно популярной и в наши дни играет весьма важную роль в области веб-разработки. Если считать показателем

популярности число звёзд, которые собрал некий проект на GitHub, то [Node.js](#), у которого более 50000 звёзд, это очень и очень популярный проект.

Платформа Node.js построена на базе JavaScript движка V8 от Google, который используется в браузере Google Chrome. Данная платформа, в основном, используется для создания веб-серверов, однако сфера её применения этим не ограничивается.

Рассмотрим основные особенности Node.js.

Скорость

Одной из основных привлекательных особенностей Node.js является скорость. JavaScript-код, выполняемый в среде Node.js, может быть в два раза быстрее, чем код, написанный на компилируемых языках, вроде C или Java, и на порядки быстрее интерпретируемых языков наподобие Python или Ruby. Причиной подобного является неблокирующая архитектура платформы, а конкретные результаты зависят от используемых тестов производительности, но, в целом, Node.js — это очень быстрая платформа.

Простота

Платформа Node.js проста в освоении и использовании. На самом деле, она прямо-таки очень проста, особенно это заметно в сравнении с некоторыми другими серверными платформами.

JavaScript

В среде Node.js выполняется код, написанный на JavaScript. Это означает, что миллионы фронтенд-разработчиков, которые уже пользуются JavaScript в браузере, могут писать и серверный, и клиентский код на одном и том же языке программирования без необходимости изучать совершенно новый инструмент для перехода к серверной разработке.

В браузере и на сервере используются одинаковые концепции языка. Кроме того, в Node.js можно оперативно переходить на использование новых стандартов ECMAScript по мере их реализации на платформе. Для этого не нужно ждать до тех пор, пока пользователи обновят браузеры, так как Node.js — это серверная среда, которую полностью контролирует разработчик. В результате новые возможности языка оказываются доступными при установке поддерживающей их версии Node.js.

Движок V8

В основе Node.js, помимо других решений, лежит open-сценарный JavaScript-движок V8 от Google, применяемый в браузере Google Chrome и в других браузерах. Это означает, что Node.js пользуется наработками тысяч инженеров, которые сделали среду выполнения JavaScript Chrome невероятно быстрой и продолжают работать в направлении совершенствования V8.

Асинхронность

В традиционных языках программирования (C, Java, Python, PHP) все инструкции, по умолчанию, являются блокирующими, если только разработчик явным образом не позаботится об асинхронном выполнении кода. В результате если, например, в такой среде, произвести сетевой запрос для загрузки некоего JSON-кода, выполнение потока, из которого сделан запрос, будет приостановлено до тех пор, пока не завершится получение и обработка ответа.

JavaScript значительно упрощает написание асинхронного и неблокирующего кода с использованием единственного потока, функций обратного вызова (коллбэков) и подхода к разработке, основанной на событиях. Каждый раз, когда нам нужно выполнить тяжёлую операцию, мы передаём соответствующему механизму коллбэк, который будет вызван сразу после завершения этой операции. В результате, для того чтобы программа продолжила работу, ждать результатов выполнения подобных операций не нужно.

Подобный механизм возник в браузерах. Мы не можем позволить себе ждать, скажем, окончания выполнения AJAX-запроса, не имея при этом возможности реагировать на действия пользователя,

например, на щелчки по кнопкам. Для того чтобы пользователям было удобно работать с веб-страницами, всё, и загрузка данных из сети, и обработка нажатия на кнопки, должно происходить одновременно, в режиме реального времени.

Если вы создавали когда-нибудь обработчик события нажатия на кнопку, то вы уже пользовались методиками асинхронного программирования.

Асинхронные механизмы позволяют единственному Node.js-серверу одновременно обрабатывать тысячи подключений, не нагружая при этом программиста задачами по управлению потоками и по организации параллельного выполнения кода. Подобные вещи часто являются источниками ошибок.

Node.js предоставляет разработчику неблокирующие базовые механизмы ввода вывода, и, в целом, библиотеки, использующиеся в среде Node.js, написаны с использованием неблокирующих парадигм. Это делает блокирующее поведение кода скорее исключением, чем нормой.

Когда Node.js нужно выполнить операцию ввода-вывода, вроде загрузки данных из сети, доступа к базе данных или к файловой системе, вместо того, чтобы заблокировать ожиданием результатов такой операции главный поток, Node.js иницирует её выполнение и продолжает заниматься другими делами до тех пор, пока результаты выполнения этой операции не будут получены.

Библиотеки

Благодаря простоте и удобству работы с менеджером пакетов для Node.js, который называется [npm](#), экосистема Node.js прямо-таки процветает. Сейчас в [реестре npm](#) имеется более полумиллиона open-source пакетов, которые может свободно использовать любой Node.js-разработчик.

Рассмотрев некоторые основные особенности платформы Node.js, опробуем её в действии. Начнём с установки.

Установка Node.js

Node.js можно устанавливать различными способами, которые мы сейчас рассмотрим.

Так, официальные установочные пакеты для всех основных платформ можно найти [здесь](#).

Существует ещё один весьма удобный способ установки Node.js, который заключается в использовании менеджера пакетов, имеющегося в операционной системе. Например, менеджер пакетов macOS, который является фактическим стандартом в этой области, называется [Homebrew](#). Если он в вашей системе есть, вы можете установить Node.js, выполнив эту команду в командной строке:

```
brew install node
```

Список менеджеров пакетов для других операционных систем, в том числе — для Linux и Windows, можно найти [здесь](#).

Популярным менеджером версий Node.js является [nvm](#). Это средство позволяет удобно переключаться между различными версиями Node.js, с его помощью можно, например, установить и попробовать новую версию Node.js, после чего, при необходимости, вернуться на старую. Nvm пригодится и в ситуации, когда нужно испытать какой-нибудь код на старой версии Node.js.

Я посоветовал бы начинающим пользоваться официальными установщиками Node.js. Пользователям macOS я порекомендовал бы устанавливать Node.js с помощью Homebrew.

Теперь, после того, как вы установили Node.js, пришло время написать «Hello World».

Первое Node.js-приложение

Самым распространённым примером первого приложения для Node.js можно назвать простой веб-сервер. Вот его код:

```
const http = require('http')

const hostname = '127.0.0.1'

const port = 3000

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

  res.statusCode = 200

  res.setHeader('Content-Type', 'text/plain')

  res.end('Hello World\n')

})

server.listen(port, hostname, () => {

  console.log(`Server running at http://${hostname}:${port}/`)

})
```

Для того чтобы запустить этот код, сохраните его в файле `server.js` и выполните в терминале такую команду:

```
node server.js
```

Для проверки сервера откройте какой-нибудь браузер и введите в адресной строке `http://127.0.0.1:3000`, то есть — тот адрес сервера, который будет выведен в консоли после его успешного запуска. Если всё работает как надо — на странице будет выведено «Hello World».

Разберём этот пример.

Для начала, обратите внимание на то, что код содержит команду подключения модуля [http](#).

Платформа Node.js является обладателем замечательного [стандартного набора модулей](#), в который входят отлично проработанные механизмы для работы с сетью.

Метод `createServer()` объекта `http` создаёт новый HTTP-сервер и возвращает его.

Сервер настроен на прослушивание определённого порта на заданном хосте. Когда сервер будет готов, вызывается соответствующий коллбэк, сообщающий нам о том, что сервер работает.

Когда сервер получает запрос, вызывается событие `request`, предоставляющее два объекта. Первый — это запрос (`req`, объект [http.IncomingMessage](#)), второй — ответ (`res`, объект [http.ServerResponse](#)). Они представляют собой важнейшие механизмы обработки HTTP-запросов.

Первый предоставляет в наше распоряжение сведения о запросе. В нашем простом примере этими данными мы не пользуемся, но, при необходимости, с помощью объекта `req` можно получить доступ к заголовкам запроса и к переданным в нём данным.

Второй нужен для формирования и отправки ответа на запрос.

В данном случае ответ на запрос мы формируем следующим образом. Сначала устанавливаем свойство `statusCode` в значение `200`, что указывает на успешное выполнение операции:

```
res.statusCode = 200
```

Далее, мы устанавливаем заголовок `Content-Type`:

```
res.setHeader('Content-Type', 'text/plain')
```

После этого мы завершаем подготовку ответа, добавляя его содержимое в качестве аргумента метода `end()`:

```
res.end('Hello World\n')
```

Мы уже говорили о том, что вокруг платформы Node.js сформировалась мощная экосистема. Обсудим теперь некоторые популярные фреймворки и вспомогательные инструменты для Node.js.

Фреймворки и вспомогательные инструменты для Node.js

Node.js — это низкоуровневая платформа. Для того чтобы упростить разработку для неё и облегчить жизнь программистам, было создано огромное количество библиотек. Некоторые из них со временем стали весьма популярными. Вот небольшой список библиотек, которые я считаю отлично сделанными и достойными изучения:

- [Express](#). Эта библиотека предоставляет разработчику предельно простой, но мощный инструмент для создания веб-серверов. Ключом к успеху Express стал минималистический подход и ориентация на базовые серверные механизмы без попытки навязать некое видение «единственно правильной» серверной архитектуры.
- [Meteor](#). Это — мощный фулстек-фреймворк, реализующий изоморфный подход к разработке приложений на JavaScript и к использованию кода и на клиенте, и на сервере. Когда-то Meteor представлял собой самостоятельный инструмент, включающий в себя всё, что только может понадобиться разработчику. Теперь он, кроме того, интегрирован с фронтенд-библиотеками, такими, как [React](#), [Vue](#) и [Angular](#). Meteor, помимо разработки обычных веб-приложений, можно использовать и в мобильной разработке.
- [Koa](#). Этот веб-фреймворк создан той же командой, которая занимается работой над Express. При его разработке, в основу которой легли годы опыта работы над Express, внимание уделялось простоте решения и его компактности. Этот проект появился как решение задачи внесения в Express серьёзных изменений, несовместимых с другими механизмами фреймворка, которые могли бы расколоть сообщество.
- [Next.js](#). Этот фреймворк предназначен для организации серверного рендеринга [React](#)-приложений.
- [Micro](#). Это — весьма компактная библиотека для создания асинхронных HTTP-микросервисов.
- [Socket.io](#). Это библиотека для разработки сетевых приложений реального времени.

На самом деле, в экосистеме Node.js можно найти вспомогательную библиотеку для решения практически любой задачи. Как вы понимаете, на строительство подобной экосистемы нужно немало времени. Платформа Node.js появилась в 2009 году. За время её существования случилось много всего такого, о чём стоит знать программисту, который хочет изучить эту платформу.

Краткая история Node.js

В этом году Node.js исполнилось уже 9 лет. Это, конечно, не так уж и много, если сравнить этот возраст с возрастом JavaScript, которому уже 23 года, или с 25-летним возрастом веба, существующем в таком виде, в котором мы его знаем, если считать от появления браузера Mosaic.

9 лет — это маленький срок для технологии, но сейчас возникает такое ощущение, что платформа Node.js существовала всегда.

Я начал работу с Node.js с ранних версий платформы, когда ей было ещё только 2 года. Даже тогда, несмотря на то, что информации о Node.js было не так уж и много, уже можно было почувствовать, что Node.js — это очень серьёзно.

Теперь поговорим о технологиях, лежащих в основе Node.js и кратко рассмотрим основные события, связанные с этой платформой.

Итак, JavaScript — это язык программирования, который был создан в Netscape как скриптовый язык, предназначенный для управления веб-страницами в браузере [Netscape Navigator](#).

Частью бизнеса Netscape была продажа веб-серверов, которые включали в себя среду, называемую Netscape LiveWire. Она позволяла создавать динамические веб-страницы, используя серверный JavaScript. Как видите, идея использования JS для серверной разработки гораздо старше чем Node.js. Этой идее почти столько же лет, сколько и самому JavaScript, но во времена, о которых идёт речь, популярности серверный JS не снискал.

Одним из ключевых факторов, благодаря которому платформа Node.js стала столь распространённой и популярной, является время её появления. Так, за несколько лет до этого JavaScript начали считать серьёзным языком. Случилось это благодаря приложениям Web 2.0, вроде Google Maps или Gmail, которые продемонстрировали миру возможности современных веб-технологий.

Благодаря конкурентной войне браузеров, которая продолжается и по сей день, серьёзно возросла производительность JavaScript-движков. Команды разработчиков, стоящих за основными браузерами, каждый день работают над повышением производительности их решений, что благотворно влияет на JavaScript в целом. Один из таких движков — это уже упомянутый V8, используемый в браузере Chrome и применяемый в Node.js. Он является одним из результатов стремления разработчиков браузеров к высокой производительности JS-кода.

Конечно же, популярность Node.js основана не только на удачном стечении обстоятельств и на том, что эта платформа появилась в правильное время. Она представила миру инновационный подход к серверной разработке на JavaScript. Рассмотрим основные вехи истории Node.js.

2009

- Появление Node.js
- Создание первого варианта [npm](#).

2010

- Появление [Express](#).
- Появление [Socket.io](#).

2011

- Выход npm 1.0.
- Большие компании, такие, как [LinkedIn](#) и [Uber](#), начали пользоваться Node.js.

2012

- Быстрый рост популярности Node.js.

2013

- Появление [Ghost](#), первой крупной платформы для публикаций, использующей Node.js.
- Выпуск [Koa](#).

2014

- В этом году произошли драматические события. Появился проект [IO.js](#), являющийся форком Node.js, целью создания которого, кроме прочего, было внедрение поддержки ES6 и ускорение развития платформы.

2015

- Основание организации [Node.js Foundation](#).
- Слияние IO.js и Node.js.
- В npm появляется возможность работать с приватными модулями.
- Выход [Node.js 4](#) (надо отметить, что версий 1, 2 и 3 у этой платформы не было).

2016

- Инцидент с пакетом [left-pad](#).
- Появление [Yarn](#).
- Выход Node.js 6.

2017

- В npm начинают больше внимания уделять безопасности.
- Выход Node.js 8
- Появление поддержки [HTTP/2](#).
- [V8](#) официально признают в качестве JS-движка, предназначенного не только для Chrome, но и для Node.
- Еженедельно осуществляется 3 миллиарда загрузок из npm.

2018

- Выход Node.js 10.
- Поддержка [ES-модулей](#).
- Экспериментальная поддержка [mjs](#).

Часть 2: JavaScript, V8, некоторые приёмы разработки

Какими JS-знаниями нужно обладать для Node.js-разработки?

Предположим, вы только начали заниматься программированием. Насколько глубоко вам нужно изучить JavaScript для успешного освоения Node.js? Начинающему трудно достичь такого уровня, когда он приобретёт достаточную уверенность в своих профессиональных навыках. К тому же, изучая программирование, вы можете почувствовать, что не понимаете точно, где заканчивается браузерный JavaScript и начинается разработка для Node.js.

Если вы находитесь в самом начале пути JavaScript-программиста, я посоветовал бы вам, прежде чем писать для Node.js, хорошо освоить следующие концепции языка:

- Лексические конструкции.
- Выражения.
- Типы.
- Переменные.
- Функции.
- Ключевое слово `this`.
- Стрелочные функции
- Циклы
- Области видимости.
- Массивы.
- Шаблонные строки.
- Применение точки с запятой.

- Работа в строгом режиме.

На самом деле, этот список можно продолжать, но если вы всё это освоите, это значит, что вы заложите хорошую базу для продуктивной клиентской и серверной разработки на JavaScript.

Следующие концепции языка, кроме того, весьма важны для понимания идей асинхронного программирования, которые являются одной из базовых частей Node.js. В частности, речь идёт о следующем:

- Асинхронное программирование и функции обратного вызова.
- Таймеры.
- Промисы.
- Конструкция `async/await`.
- Замыкания.
- Цикл событий.

Существует множество материалов по JavaScript, которые позволяют начинающим освоить язык. Например, [вот](#) учебный курс автора данного руководства, [вот](#) весьма полезный раздел MDN, [вот](#) учебник сайта javascript.ru. Изучить базовые механизмы JavaScript можно на freecodecamp.com.

Выше мы говорили о том, что начинающих может беспокоить вопрос о том, где проходит граница между серверной и клиентской разработкой на JavaScript. Поговорим об этом.

Различия между платформой Node.js и браузером

Чем JS-разработка для Node.js отличается от браузерного программирования? Сходство между этими средами заключается в том, что и там и там используется один и тот же язык. Но разработка приложений, рассчитанных на выполнение в браузере, очень сильно отличается от разработки серверных приложений. Несмотря на использование одного и того же языка, существуют некоторые ключевые различия, которые и превращают два эти вида разработки в совершенно разные занятия.

Надо отметить, что если тот, кто раньше занимался фронтендом, начинает изучать Node.js, у него имеется весьма серьёзная возможность довольно быстро освоить всё что нужно благодаря тому, что писать он будет на уже знакомом ему языке. Если к необходимости освоить новую среду добавляется ещё и необходимость изучить новый язык, задача значительно усложняется.

Итак главное различие между клиентом и сервером заключается в окружении, для которого приходится программировать, в экосистемах этих сред.

В браузере основной объём работы приходится на выполнение различных операций с веб-документами посредством DOM, а также — на использование других API веб-платформы, таких, скажем, как механизмы для работы с куки-файлами. Всего этого в Node.js, конечно, нет. Тут нет ни объекта `document`, ни объекта `window`, равно как и других объектов, предоставляемых браузером.

В браузере, в свою очередь, нет тех программных механизмов, которые имеются в среде Node.js и существуют в виде модулей, которые можно подключать к приложению. Например, это API для доступа к файловой системе.

Ещё одно различие между клиентской и серверной разработкой на JS заключается в том, что при работе в среде Node.js разработчик полностью контролирует окружение. Если только вы не занимаетесь разработкой опенсорсного приложения, которое может быть запущено где угодно, вы точно знаете, например, на какой версии Node.js будет работать ваш проект. Это очень удобно в сравнении с клиентским окружением, где вашему коду приходится работать в имеющемся у пользователя браузере. Кроме того, это означает, что вы можете, не опасаясь проблем, пользоваться новейшими возможностями языка.

Так как JavaScript крайне быстро развивается, браузеры просто не успевают достаточно оперативно реализовать все его новшества. К тому же, далеко не все пользователи работают на самых свежих версиях браузеров. В результате разработчики, которые хотят использовать в своих программах что-то новое, вынуждены это учитывать, заботиться о совместимости их приложений с используемыми браузерами, что может вылиться в необходимость отказа от современных возможностей JavaScript. Можно, конечно, для преобразования кода в формат, совместимый со стандартом ECMAScript 5, который поддерживают все браузеры, воспользоваться транспилятором Babel, но при работе с Node.js вам это не понадобится.

Ещё одно различие между Node.js и браузерами заключается в том, что в Node.js используется система модулей [CommonJS](#), в то время как в браузерах можно наблюдать начало реализации стандарта ES Modules. На практике это означает, что в настоящее время в Node.js, для подключения внешнего кода, используется конструкция `require()`, а в браузерном коде — `import`.

V8 и другие JavaScript-движки

V8 — это название JavaScript-движка, используемого в браузере Google Chrome. Именно он отвечает за выполнение JavaScript-кода, который попадает в браузер при работе в интернете. V8 предоставляет среду выполнения для JavaScript. DOM и другие API веб-платформы предоставляются браузером.

JS-движок независим от браузера, в котором он работает. Именно этот факт сделал возможным появление и развитие платформы Node.js. V8 был выбран в качестве движка для Node.js в 2009 году. В результате прямо-таки взрывного роста популярности Node.js V8 оказался движком, который в наши дни отвечает за выполнение огромного количества серверного JS-кода.

Экосистема Node.js огромна. Благодаря этому V8 также используется, посредством проектов наподобие [Electron](#), при разработке настольных приложений.

Надо отметить, что, помимо V8, существуют и другие движки:

- В браузере Firefox применяется движок [SpiderMonkey](#).
- В Safari применяется [JavaScriptCore](#) (он ещё называется Nitro).
- В Edge используется движок [Chakra](#).

Список JS-движков этим не ограничивается.

Эти движки реализуют спецификацию ECMA-262, называемую ещё ECMAScript. Именно эта спецификация стандартизирует JavaScript. Свежую версию стандарта можно найти [здесь](#).

Разработка JS-движков и стремление к производительности

Движок V8 написан на C++, его постоянно улучшают. Он может выполняться на многих системах, в частности, на Mac, Windows и Linux. Здесь мы не будем говорить о деталях реализации V8. Сведения о них можно найти в других публикациях, в том числе — на [официальном сайте V8](#). Они со временем меняются, иногда — очень серьёзно.

V8 постоянно развивается, то же самое можно сказать и о других движках. Это приводит, в частности, к росту производительности веб-браузеров и платформы Node.js. Производители движков для браузеров постоянно соревнуются, борясь за скорость выполнения кода, продолжается это уже многие годы. Всё это идёт на пользу пользователям и программистам.

Интерпретация и компиляция

JavaScript считается интерпретируемым языком, но современные движки занимаются далеко не только интерпретацией JS-кода. Они его компилируют. Это веяние можно наблюдать с 2009-го года, когда компилятор JavaScript был добавлен в Firefox 3.5, после чего и другие производители движков и браузеров переняли эту идею.

V8 выполняет компиляцию JavaScript для повышения производительности кода. Со времени появления Google Maps в 2004-м году JavaScript эволюционировал, превратился из языка, на котором, для реализации интерактивных возможностей веб-приложений, обычно писали по несколько десятков строк, в язык, на котором пишут браузерные приложения, состоящие из тысяч или даже сотен тысяч строк кода. Такие приложения могут выполняться в браузере часами, что серьёзно отличается от старых сценариев использования JS, код на котором, например, мог применяться лишь для проверки правильности данных, вводимых в формы. В современных условиях компиляция кода имеет огромный смысл, так как, хотя выполнение этого шага может отложить момент запуска кода, после компиляции код оказывается гораздо более производительным, чем тот, который обрабатывался бы исключительно интерпретатором и запускался бы быстрее, но работал бы медленнее.

Теперь, обсудив некоторые положения, касающиеся JS-движков, интерпретации и компиляции кода, перейдём к практике. А именно, поговорим о том, как завершать работу Node.js-приложений.

Выход из Node.js-приложения

Существует несколько способов завершения работы Node.js-приложений.

Так, при выполнении программы в консоли, завершить её работу можно, воспользовавшись сочетанием клавиш `ctrl+c`. Но нас больше интересуют программные способы завершения работы приложений. И начнём мы, пожалуй, с самой грубой команды выхода из программы, которую, как вы сейчас поймёте, лучше не использовать.

Модуль ядра `process` предоставляет удобный метод, который позволяет осуществить программный выход из Node.js-приложения. Выглядит это так:

```
process.exit()
```

Когда Node.js встречает в коде такую команду, это приводит к тому, что его процесс мгновенно завершается. Это означает, что абсолютно всё, чем занималась программа, будет довольно грубо и безусловно прервано. Речь идёт о невызванных коллбэках, о выполняемых в момент выхода сетевых запросах, о действиях с файлами, об операциях записи в `stdout` или `stderr`.

Если вас такое положение дел устраивает — можете этим методом пользоваться. При его вызове можно передать ему целое число, которое будет воспринято операционной системой как код выхода из программы.

```
process.exit(1)
```

По умолчанию этот код имеет значение 0, что означает успешное завершение работы. Другие коды выхода имеют другие значения, которые могут оказаться полезными для использования их в собственной системе для того, чтобы наладить взаимодействие одних программ с другими.

Подробности о кодах завершения работы программ можно почитать [здесь](#).

Код выхода, кроме того, можно назначить свойству `process.exitCode`. Выглядит это так:

```
process.exitCode = 1
```

После того, как программа завершит работу, Node.js вернёт системе этот код.

Надо отметить, что работа программы самостоятельно завершится естественным образом после того, как она выполнит все заданные в ней действия. Однако в случае с Node.js часто встречаются программы, которые, в идеальных условиях, рассчитаны на работу неопределённой длительности. Речь идёт, например, об HTTP-серверах, подобных такому:


```
const express = require('express')

const app = express()

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

  res.send('Hi!')

})

app.listen(3000, () => console.log('Server ready'))
```

Такая программа, если ничего не случится, в теории, может работать вечно. При этом, если вызвать `process.exit()`, выполняемые ей в момент вызова этой команды операции будут прерваны. А это плохо.

Для завершения работы подобных программ нужно воспользоваться сигналом `SIGTERM` и выполнить необходимые действия с помощью соответствующего обработчика.

Обратите внимание на то, что для использования объекта `process` не нужно ничего подключать с помощью `require`, так как этот объект доступен Node.js-приложениям по умолчанию.

Рассмотрим следующий пример:

```
const express = require('express')

const app = express()

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

  res.send('Hi!')

})

app.listen(3000, () => console.log('Server ready'))

process.on('SIGTERM', () => {

  app.close(() => {

    console.log('Process terminated')

  })

})
```

Что такое «сигналы»? Сигналы — это средства взаимодействия процессов в стандарте POSIX (Portable Operating System Interface). Они представляют собой уведомления, отправляемые процессу для того, чтобы сообщить ему о неких событиях.

Например, сигнал `SIGKILL` сообщает процессу о том, что ему нужно немедленно завершить работу. Он, в идеале, работает так же, как `process.exit()`.

Сигнал `SIGTERM` сообщает процессу о том, что ему нужно осуществить процедуру нормального завершения работы. Подобные сигналы отправляются из менеджеров процессов, вроде `upstart` или `supervisord`, и из многих других.

Отправить такой сигнал можно и из самой программы, воспользовавшись следующей командой:

```
process.kill(process.pid, 'SIGTERM')
```

Для успешного выполнения подобной команды нужно знать `PID` процесса, который планируется завершить.

Чтение переменных окружения из Node.js

Модуль ядра `process` имеет свойство `env`, которое даёт доступ ко всем переменным окружения, которые были заданы на момент запуска процесса.

Вот пример работы с переменной окружения `NODE_ENV`, которая, по умолчанию, установлена в значение `development`:

```
process.env.NODE_ENV // "development"
```

Если, до запуска скрипта, установить её в значение `production`, это сообщит Node.js о том, что программа выполняется в продакшн-окружении.

Подобным образом можно работать и с другими переменными среды, например, с теми, которые установлены вами самостоятельно.

Часть 3: хостинг, REPL, работа с консолью, модули

Хостинг для Node.js-приложений

Выбор хостинга для Node.js-приложений зависит от ваших потребностей. Вот небольшой список вариантов хостинга, который вы можете изучить, приняв решение развернуть своё приложение и сделать его общедоступным. Сначала рассмотрим простые варианты, возможности которых ограничены, а потом — более сложные, но и обладающие более серьёзными возможностями.

Самый простой вариант хостинга: локальный туннель

Даже если вашему компьютеру назначен динамический IP-адрес или вы находитесь за NAT, вы можете развернуть на нём своё приложение и обслуживать запросы пользователей к нему, используя локальный туннель.

Этот вариант подходит для быстрой организации тестирования, для того, чтобы устроить демонстрацию продукта, или для того, чтобы дать доступ к приложению очень маленькой группе людей.

Для организации локальных туннелей есть очень хороший сервис, [ngrok](#), доступный для множества платформ.

Используя `ngrok`, достаточно выполнить команду вида `ngrok PORT` и указанный вами порт будет доступен из интернета. У вас при этом, если вы пользуетесь бесплатной версией сервиса, будет адрес в домене `ngrok.io`. Если же вы решите оформить платную подписку, вы сможете использовать собственные доменные имена, и, кроме того, сможете повысить безопасность решения (пользуясь `ngrok`, вы открываете доступ к своему компьютеру всему интернету).

Ещё один инструмент, который можно использовать для организации локальных туннелей, называется [localtunnel](#).

Среды для развёртывания Node.js-проектов, не требующие настройки

Glitch

[Glitch](#) — это интерактивная среда и платформа для быстрой разработки приложений, которая позволяет разворачивать их в поддоменах `glitch.com`. Собственные домены пользователей эта платформа пока не поддерживает, при работе с ней существуют некоторые [ограничения](#), но она

отлично подходит для работы над прототипами приложений. Дизайн Glitch выглядит довольно забавно (пожалуй, это можно записать в плюсы данной платформы), но это не некая «игрушечная», ограниченная донельзя среда. Здесь к вашим услугам возможность работы с Node.js, CDN, защищённое хранилище для конфиденциальной информации, возможности обмена данными с GitHub и многое другое.

Проектом Glitch занимается та же компания, которая стоит за FogBugz и Trello (она же является одним из создателей StackOverflow). Я часто использую эту платформу для демонстрации приложений.

Codepen

[Codepen](#) — это замечательная платформа, вокруг которой сформировалось интересное сообщество. Здесь можно создавать проекты, включающие в себя множество файлов, и разворачивать их с использованием собственного домена.

Бессерверные среды

Бессерверные платформы позволяют публиковать приложения и при этом совершенно не думать о серверах, об их настройке или об управлении ими. Парадигма бессерверных вычислений заключается в том, что приложения публикуют в виде функций, которые реагируют на обращения к сетевой конечной точке. Подобный подход к развёртыванию приложений ещё называют FAAS (Functions As A Service, функция как услуга).

Вот пара популярных решений в этой области:

- Фреймворк [Serverless](#).
- Библиотека [Standard](#).

Оба эти проекта предоставляют разработчику некий уровень абстракции, позволяющий публиковать приложения на различных FAAS-платформах, например, на Amazon AWS Lambda, на Microsoft Azure и на Google Cloud.

PAAS-решения

PAAS (Platform As A Service, платформа как услуга) — это платформы, которые берут на себя заботу обо многих вещах, о которых, в обычных условиях, должен заботиться разработчик, развёртывающий приложение.

Zeit Now

[Zeit](#) — это интересный вариант для развёртывания приложений. Развёртывание, при использовании этой платформы, сводится к вводу в терминале команды `now`. Существует бесплатная версия Zeit, при работе с ней действуют некоторые ограничения. Есть и платная, более мощная версия этой платформы. Пользуясь Zeit, вы можете попросту не думать о том, что для работы вашего приложения нужен сервер. Вы просто разворачиваете приложение, а всё остальное находится в ведении этой платформы.

Nanobox

Создатели платформы [Nanobox](#), в возможности которой входит развёртывание Node.js-приложений, называют её PAAS V2.

Heroku

[Heroku](#) — это ещё одна замечательная платформа для размещения Node.js-приложений. [Вот](#) хорошая статья о том, как с ней работать.

Microsoft Azure

[Azure](#) — это облачная платформа от Microsoft. В её документации есть [раздел](#), посвящённый Node.js-приложениям.

Платформа Google Cloud

[Google Cloud](#) представляет собой замечательную среду для развёртывания Node.js-приложений. [Вот](#) соответствующий раздел её документации.

VPS-хостинг

Существует множество платформ, предоставляющих услуги [VPS-хостинга](#). Общей чертой таких платформ является тот факт, что разработчик получает в своё распоряжение виртуальный сервер, самостоятельно устанавливает на него операционную систему (Linux или Windows), самостоятельно развёртывает приложения.

Среди платформ, предоставляющих VPS-услуги, которых существует великое множество, можно отметить следующие, которыми я пользовался и которые мог бы порекомендовать другим:

- [Digital Ocean](#)
- [Linode](#)
- [Amazon Web Services](#) (в частности, хотелось бы отметить сервис [AWS Elastic Beanstalk](#), облегчающий развёртывание приложений и управление ресурсами AWS).

От себя добавим, что компания [RUVDS](#) тоже оказывает услуги VPS-хостинга. Мы лицензированы ФСТЭК, наши клиенты застрахованы AIG, у нас есть четыре дата-центра. Два расположены в Москве, на площадках [ММТС-9](#) и [MTW](#), один [в Швейцарских Альпах](#), и ещё один [в Лондоне](#).

Обычный сервер

Ещё одно решение в области хостинга представляет собой покупку (или аренду, например, с помощью службы [Vultr Bare Metal](#)) обычного сервера, установку на него Linux и другого ПО, подключение его к интернету и размещение на нём Node.js-приложений.

Хостинг — тема огромная, но, надеемся, материалы этого раздела позволят вам выбрать именно то, что вам нужно. Теперь переходим к рассказу о работе с Node.js в режиме REPL

Использование Node.js в режиме REPL

Аббревиатура REPL расшифровывается как Read-Evaluate-Print-Loop (цикл «чтение — вычисление — вывод»). Использование REPL — это отличный способ быстрого исследования возможностей Node.js.

Как вы уже знаете, для запуска скриптов в Node.js используется команда `node`, выглядит это так:

```
node script.js
```

Если ввести такую же команду, но не указывать имя файла, Node.js будет запущен в режиме REPL:

```
node
```

Если вы попытаете сейчас ввести такую команду в своём терминале, то в результате увидите примерно следующее:

```
> node
```

```
>
```

Node.js теперь находится в режиме ожидания. Система ждёт, что мы введём в командной строке какой-нибудь JavaScript-код, который она будет выполнять.

Для начала попробуем что-нибудь очень простое:

```
> console.log('test')

test

undefined

>
```

Тут мы предложили Node.js выполнить команду, используемую для вывода данных в консоль. Первое значение, `test`, представляет собой то, что вывела команда `console.log('test')`. Второе значение, `undefined`, это то, что возвратила функция `console.log()`.

После завершения выполнения команды появляется приглашение REPL, это означает, что мы можем ввести здесь новую команду.

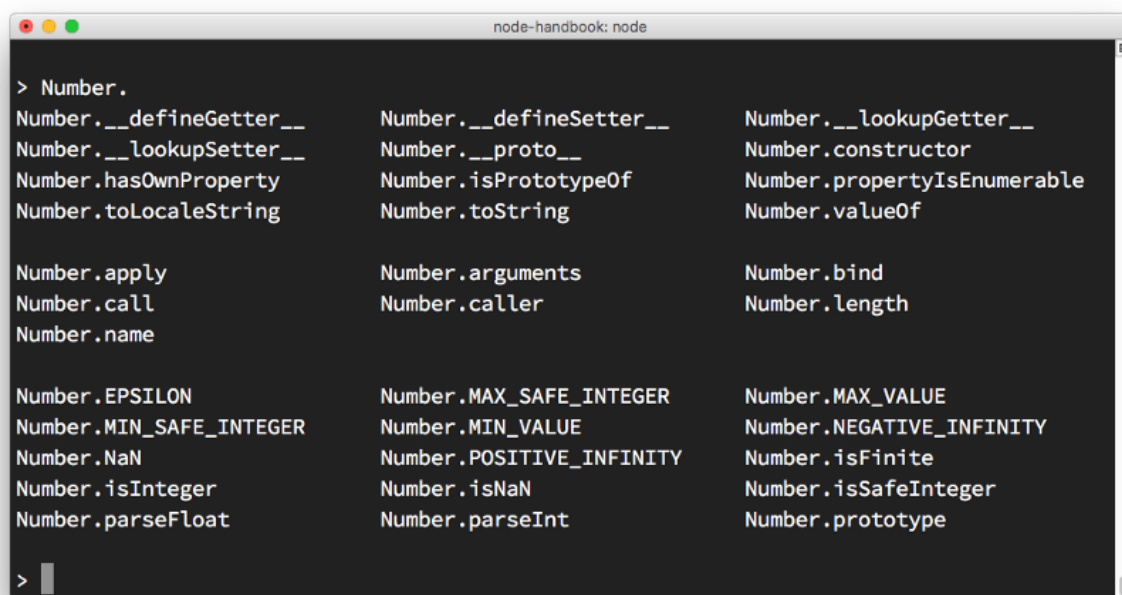
Автозавершение команд с помощью клавиши Tab

REPL — это интерактивная среда. Если в процессе написания кода нажать клавишу `Tab` на клавиатуре, REPL попытается автоматически завершить ввод, подобрав, например, подходящее имя уже объявленной вами переменной или имя некоего стандартного объекта.

Исследование объектов JavaScript

Введите в командную строку имя какого-нибудь стандартного объекта JavaScript, например — `Number`, добавьте после него точку и нажмите клавишу `Tab`.

REPL выведет список свойств и методов объекта, с которыми может взаимодействовать разработчик:



```
node-handbook: node

> Number.
Number.__defineGetter__      Number.__defineSetter__    Number.__lookupGetter__
Number.__lookupSetter__     Number.__proto__           Number.constructor
Number.hasOwnProperty       Number.isPrototypeOf       Number.propertyIsEnumerable
Number.toLocaleString       Number.toString            Number.valueOf

Number.apply                Number.arguments           Number.bind
Number.call                 Number.caller              Number.length
Number.name

Number.EPSILON              Number.MAX_SAFE_INTEGER    Number.MAX_VALUE
Number.MIN_SAFE_INTEGER     Number.MIN_VALUE           Number.NEGATIVE_INFINITY
Number.NaN                  Number.POSITIVE_INFINITY   Number.isFinite
Number.isInteger            Number.isNaN               Number.isSafeInteger
Number.parseFloat           Number.parseInt            Number.prototype

> |
```

И с с л е д о в а н и е о б ъ е к т а Number

Исследование глобальных объектов

Для того чтобы узнать, с какими глобальными объектам Node.js вы можете работать, введите в терминале команду `global.` и нажмите `Tab`.

```
node-handbook: node
> global.
global.__defineGetter__      global.__defineSetter__
global.__lookupGetter__     global.__lookupSetter__
global.__proto__            global.constructor
global.hasOwnProperty       global.isPrototypeOf
global.propertyIsEnumerable global.toLocaleString
global.toString             global.valueOf

global.Array                global.ArrayBuffer
global.Boolean              global.Buffer
global.DTRACE_HTTP_CLIENT_REQUEST global.DTRACE_HTTP_CLIENT_RESPONSE
global.DTRACE_HTTP_SERVER_REQUEST global.DTRACE_HTTP_SERVER_RESPONSE
global.DTRACE_NET_SERVER_CONNECTION global.DTRACE_NET_STREAM_END
global.DataView             global.Date
global.Error                global.EvalError
global.Float32Array         global.Float64Array
global.Function             global.GLOBAL
global.Infinity             global.Int16Array
global.Int32Array           global.Int8Array
global.Intl                 global.JSON
global.Map                  global.Math
global.NaN                  global.Number
global.Object               global.Promise
global.Proxy                global.RangeError
global.ReferenceError       global.Reflect
global.RegExp               global.Set
global.String               global.Symbol
global.SyntaxError          global.TypeError
global.URLError             global.Uint16Array
global.Uint32Array          global.Uint8Array
global.Uint8ClampedArray    global.WeakMap
global.WeakSet              global.WebAssembly
```

Исследование глобальных объектов

Специальная переменная _

Переменная _ (знак подчёркивания) хранит результат последней выполненной операции. Эту переменную можно использовать в составе команд, вводимых в консоль.

Команды, начинающиеся с точки

В режиме REPL можно пользоваться некоторыми специальными командами, которые начинаются с точки. Вот они:

- Команда `.help` выводит справочные сведения по командам, начинающимся с точки.
- Команда `.editor` переводит систему в режим редактора, что упрощает ввод многострочного JavaScript-кода. После того, как находясь в этом режиме, вы введёте всё, что хотели, для запуска кода воспользуйтесь командой `Ctrl+D`.
- Команда `.break` позволяет прервать ввод многострочного выражения. Её использование аналогично применению сочетания клавиш `Ctrl+C`.
- Команда `.clear` очищает контекст REPL, а так же прерывает ввод многострочного выражения.
- Команда `.load` загружает в текущий сеанс код из JavaScript-файла.
- Команда `.save` сохраняет в файл всё, что было введено во время REPL-сеанса.
- Команда `.exit` позволяет выйти из сеанса REPL, она действует так же, как два последовательных нажатия сочетания клавиш `Ctrl+C`.

Надо отметить, что REPL распознаёт ввод многострочных выражений и без использования команды `.editor`.

Например, мы начали вводить код итератора:

```
[1, 2, 3].forEach(num => {
```

Если, после ввода фигурной скобки, нажать на клавишу `Enter`, REPL перейдёт на новую строку, приглашение в которой будет выглядеть как три точки. Это указывает на то, что мы можем вводить код соответствующего блока. Выглядит это так:

```
... console.log(num)

... })
```

Нажатие на `Enter` после ввода последней скобки приведёт к выполнению выражения. Если ввести в этом режиме `.break`, ввод будет прерван и выражение выполнено не будет.

Режим REPL — полезная возможность Node.js, но область её применения ограничена небольшими экспериментами. Нас же интересует нечто большее, чем возможность выполнить пару команд. Поэтому переходим к работе с Node.js в обычном режиме. А именно, поговорим о том, как Node.js-скрипты могут принимать аргументы командной строки.

Работа с аргументами командной строки в Node.js-скриптах

При запуске Node.js-скриптов им можно передавать аргументы. Вот обычный вызов скрипта:

```
node app.js
```

Передаваемые скрипту аргументы могут представлять собой как самостоятельные значения, так и конструкции вида ключ-значение. В первом случае запуск скрипта выглядит так:

```
node app.js flavio
```

Во втором — так:

```
node app.js name=flavio
```

От того, какой именно способ передачи аргументов используется, зависит то, как с ними можно будет работать в коде скрипта.

Так, для того, чтобы получить доступ к аргументам командной строки, используется стандартный объект Node.js `process`. У него есть свойство `argv`, которое представляет собой массив, содержащий, кроме прочего, аргументы, переданные скрипту при запуске.

Первый элемент массива `argv` содержит полный путь к файлу, который выполняется при вводе команды `node` в командной строке.

Второй элемент — это путь к выполняемому файлу скрипта.

Все остальные элементы массива, начиная с третьего, содержат то, что было передано скрипту при его запуске.

Перебор аргументов, имеющихся в `argv` (сюда входят и путь к `node`, и путь к выполняемому файлу скрипта), можно организовать с использованием цикла `forEach`:

```
process.argv.forEach((val, index) => {
```



```
    console.log(`${index}: ${val}`)  
  })  
})
```

Если два первых аргумента вас не интересуют, на основе `argv` можно сформировать новый массив, в который войдёт всё из `argv` кроме первых двух элементов:

```
const args = process.argv.slice(2)
```

Предположим, при запуске скрипта, ему передали лишь один аргумент, в виде самостоятельного значения:

```
node app.js flavio
```

Обратиться к этому аргументу можно так:

```
const args = process.argv.slice(2)  
  
args[0]
```

Теперь попробуем воспользоваться конструкцией вида ключ-значение:

```
node app.js name=flavio
```

При таком подходе, после формирования массива `args`, в `args[0]` окажется строка `name=flavio`. Прежде чем пользоваться аргументом, эту строку надо разобрать. Самый удобный способ это сделать заключается в использовании библиотеки [minimist](#), которая предназначена для облегчения работы с аргументами командной строки:

```
const args = require('minimist')(process.argv.slice(2))  
  
args['name'] //flavio
```

Теперь рассмотрим вывод данных в консоль.

Вывод данных в консоль с использованием модуля `console`

Стандартный модуль Node.js [console](#) даёт разработчику массу возможностей по взаимодействию с командной строкой во время выполнения программы. В целом, это — то же самое, что объект `console`, используемый в браузерном JavaScript. Пожалуй, самый простой и самый широко используемый метод модуля `console` — это `console.log()`, который применяется для вывода передаваемых ему строковых данных в консоль. При этом, если передать ему объект, то он, перед выводом, будет преобразован к своему строковому представлению.

Методу `console.log()` можно передавать несколько значений:

```
const x = 'x'  
  
const y = 'y'  
  
console.log(x, y)
```

После выполнения этой последовательности инструкций в консоль попадёт и значение `x`, и значение `y`.

Для формирования сложных строк команда `console.log()` поддерживает использование подстановочных символов, которые, при выводе данных, заменяются на соответствующие им значения в порядке очередности.

Например, вот команда, которая выводит текст `My cat has 2 years`:


```
console.log('My %s has %d years', 'cat', 2)
```

Рассмотрим особенности подстановочных символов:

- `%s` форматирует значение в виде строки.
- `%d` или `%i` форматируют значение в виде целого числа.
- `%f` форматирует значение в виде числа с плавающей точкой.
- `%O` используется для вывода строковых представлений объектов.

Вот ещё один пример использования подстановочных символов:

```
console.log('%O', Number)
```

Очистка консоли

Для очистки консоли используется команда `console.clear()` (её поведение в разных терминалах может различаться).

Подсчёт элементов

Сейчас мы рассмотрим полезный метод `console.count()`. Взгляните на этот код:

```
const x = 1

const y = 2

const z = 3

console.count(
  'The value of x is ' + x + ' and has been checked .. how many times?'
)

console.count(
  'The value of x is ' + x + ' and has been checked .. how many times?'
)

console.count(
  'The value of y is ' + y + ' and has been checked .. how many times?'
)
```

Метод `count()` подсчитывает количество выводов строк и выводит результат рядом с ними.

Используя этот метод можно, в следующем примере, посчитать яблоки и апельсины:

```
const oranges = ['orange', 'orange']

const apples = ['just one apple']

oranges.forEach(fruit => {
  console.count(fruit)
})

apples.forEach(fruit => {
```

```
    console.count(fruit)
  })
}
```

Вывод в консоль результатов трассировки стека

Иногда бывает полезно вывести в консоль трассировку стека функции. Например, для того, чтобы ответить на вопрос о том, как мы попали в некое место программы. Сделать это можно с помощью такой команды:

```
console.trace()
```

Вот пример её использования:

```
const function2 = () => console.trace()

const function1 = () => function2()

function1()
```

Вот что произошло, когда я запустил этот код в режиме REPL:

```
Trace
    at function2 (repl:1:33)
    at function1 (repl:1:25)
    at repl:1:1
    at ContextifyScript.Script.runInThisContext (vm.js:44:33)
    at REPLServer.defaultEval (repl.js:239:29)
    at bound (domain.js:301:14)
    at REPLServer.runBound [as eval] (domain.js:314:12)
    at REPLServer.onLine (repl.js:440:10)
    at emitOne (events.js:120:20)
    at REPLServer.emit (events.js:210:7)
```

Измерение времени, затраченного на выполнение некоего действия

Измерить время, которое занимает, например, выполнение некоей функции, можно с использованием методов `console.time()` и `console.timeEnd()`. Выглядит это так:

```
const doSomething = () => console.log('test')

const measureDoingSomething = () => {
  console.time('doSomething()')

  //вызываем функцию и замеряем время, необходимое на её выполнение
  doSomething()

  console.timeEnd('doSomething()')
```

```
}
```

```
measureDoingSomething()
```

Работа с stdout и stderr

Как мы уже видели, команда `console.log()` отлично подходит для вывода сообщений в консоль. При её применении используется так называемый стандартный поток вывода, или `stdout`.

Команда `console.error()` выводит данные в стандартный поток ошибок, `stderr`. Данные, отправляемые в `stderr`, попадают в консоль, хотя то, что выводится в этот поток, можно, например, перенаправить в файл журнала ошибок.

Использование цвета при выводе данных в консоль

Для того чтобы раскрасить выводимые в консоль тексты, можно воспользоваться escape-последовательностями, идентифицирующими цвета:

```
console.log('\x1b[33m%s\x1b[0m', 'hi!')
```

Если выполнить эту команду, например, в режиме REPL, текст `hi` будет выведен жёлтым цветом.

Такой подход, однако, не особенно удобен. Для вывода в консоль цветных надписей удобно будет воспользоваться специализированной библиотекой, например — [chalk](#). Эта библиотека, помимо цветового форматирования текстов, поддерживает и другие способы их стилизации. Например, с её помощью можно оформить текст полужирным, курсивным или подчёркнутым шрифтом.

Для её установки из npm воспользуйтесь такой командой:

```
npm install chalk
```

Пользоваться ей можно так:

```
const chalk = require('chalk')

console.log(chalk.yellow('hi!'))
```

Пользоваться командой `chalk.yellow()` гораздо удобнее, чем escape-последовательностями, да и текст программы при таком подходе читать гораздо легче.

Для того чтобы узнать подробности о `chalk`, посмотрите [страницу](#) этой библиотеки на GitHub.

Создание индикатора выполнения операции

Индикатор выполнения операции (progress bar) может пригодиться в разных ситуациях. Для создания индикаторов выполнения, работающих в консоли, можно воспользоваться пакетом [progress](#).

Установить его можно так:

```
npm install progress
```

Ниже показан пример кода, в котором создаётся индикатор, который можно использовать для вывода сведений о некоей задаче, состоящей из 10 шагов. В нашем случае на выполнение каждого шага уходит 100 мс. После того, как индикатор заполнится, вызывается команда `clearInterval()` и выполнение программы завершается.

```
const ProgressBar = require('progress')

const bar = new ProgressBar(':bar', { total: 10 })

const timer = setInterval(() => {
```

```
bar.tick()

if (bar.complete) {
  clearInterval(timer)
}

}, 100)
```

Приём пользовательского ввода из командной строки

Как сделать приложения командной строки, написанные для платформы Node.js, интерактивными? Начиная с 7 версии Node.js содержит модуль [readline](#), который позволяет принимать данные из потоков, которые можно читать, например, из `process.stdin`. Этот поток, во время выполнения Node.js-программы, представляет собой то, что вводят в терминале. Данные вводятся по одной строке за раз.

Рассмотрим следующий фрагмент кода:

```
const readline = require('readline').createInterface({
  input: process.stdin,
  output: process.stdout
})

readline.question(`What's your name?`, (name) => {
  console.log(`Hi ${name}!`)
  readline.close()
})
```

Здесь мы спрашиваем у пользователя его имя, а после ввода текста и нажатия на клавишу `Enter` на клавиатуре, выводим приветствие.

Метод `question()` выводит то, что передано ему в качестве первого параметра (то есть — вопрос, задаваемый пользователю) и ожидает завершения ввода. После нажатия на `Enter` он вызывает коллбэк, переданный ему во втором параметре и обрабатывает то, что было введено. В этом же коллбэке мы закрываем интерфейс `readline`.

Модуль `readline` поддерживает и другие методы, подробности о них вы можете узнать в документации, ссылка на которую приведена выше.

Если вам, с использованием этого механизма, надо запросить у пользователя пароль, то лучше не выводить его, в ходе ввода, на экран, а показывать вместо введённых символов символ звёздочки — `*`.

Для того чтобы это сделать, можно воспользоваться пакетом [readline-sync](#), устройство которого похоже на то, как устроен модуль `readline`, и который поддерживает подобные возможности сразу после установки.

Есть и ещё один пакет, предоставляющий более полное и абстрактное решение подобной проблемы. Это пакет [inquirer](#). Установить его можно так:

```
npm install inquirer
```

С его использованием вышеприведенный пример можно переписать следующим образом:

```
const inquirer = require('inquirer')

var questions = [{

  type: 'input',

  name: 'name',

  message: "What's your name?",

}]

inquirer.prompt(questions).then(answers => {

  console.log(`Hi ${answers['name']}!`)

})
```

Пакет `inquirer` обладает обширными возможностями. Например, он может помочь задать пользователю вопрос с несколькими вариантами ответа или сформировать в консоли интерфейс с радиокнопками.

Программисту стоит знать о наличии альтернативных возможностей по выполнению неких действий в Node.js. В нашем случае это стандартный модуль `readline`, пакеты `readline-sync` и `inquirer`. Выбор конкретного решения зависит от целей проекта, от наличия времени на реализацию тех или иных возможностей и от сложности пользовательского интерфейса, который планируется сформировать средствами командной строки.

Система модулей Node.js, использование команды `exports`

Поговорим о том, как использовать API `module.exports` для того, чтобы открывать доступ к возможностям модулей другим файлам приложения. В Node.js имеется встроенная система модулей, каждый файл при этом считается самостоятельным [модулем](#). Общедоступный функционал модуля, с помощью команды `require`, могут использовать другие модули:

```
const library = require('./library')
```

Здесь показан импорт модуля `library.js`, файл которого расположен в той же папке, в которой находится файл, импортирующий его.

Модуль, прежде чем будет смысл его импортировать, должен что-то экспортировать, сделать общедоступным. Ко всему, что явным образом не экспортируется модулем, нет доступа извне. Собственно говоря, API `module.exports` позволяет организовать экспорт того, что будет доступно внешним по отношению к модулю механизмам.

Экспорт можно организовать двумя способами.

Первый заключается в записи объекта в `module.exports`, который является стандартным объектом, предоставляемым системой модулей. Это приводит к экспорту только соответствующего объекта:

```
const car = {

  brand: 'Ford',

  model: 'Fiesta'

}
```

```
module.exports = car
```

```
//..в другом файле
```

```
const car = require('./car')
```

Второй способ заключается в том, что экспортируемый объект записывают в свойство объекта `exports`. Такой подход позволяет экспортировать из модуля несколько объектов, и, в том числе — функций:

```
const car = {  
  brand: 'Ford',  
  model: 'Fiesta'  
}  
  
exports.car = car
```

То же самое можно переписать и короче:

```
exports.car = {  
  brand: 'Ford',  
  model: 'Fiesta'  
}
```

В другом файле воспользоваться тем, что экспортировал модуль, можно так:

```
const items = require('./items')  
  
items.car
```

Или так:

```
const car = require('./items').car
```

В чём разница между записью объекта в `module.exports` и заданием свойств объекта `exports`?

В первом экспортируется объект, который записан в `module.exports`. Во втором случае экспортируются свойства этого объекта.

Часть 4: npm, файлы `package.json` и `package-lock.json`

Основы npm

Npm (node package manager) — это менеджер пакетов Node.js. В первой части этого материала мы уже упоминали о том, что сейчас в npm имеется более полумиллиона пакетов, что делает его самым большим в мире репозиторием кода, написанного на одном языке. Это позволяет говорить о том, что в npm можно найти пакеты, предназначенные для решения практически любых задач.

Изначально npm создавался как система управления пакетами для Node.js, но в наши дни он используется и при разработке фронтенд-проектов на JavaScript. Для взаимодействия с реестром npm используется одноимённая команда, которая даёт разработчику огромное количество возможностей.

Загрузка пакетов

С помощью команды `npm` можно загружать пакеты из реестра. Ниже мы рассмотрим примеры её использования.

Установка всех зависимостей проекта

Если в проекте имеется файл `package.json`, то установить все зависимости этого проекта можно такой командой:

```
npm install
```

Эта команда загрузит всё, что нужно проекту, и поместит эти материалы в папку `node_modules`, создав её в том случае, если она не существует в директории проекта.

Установка отдельного пакета

Отдельный можно установить следующей командой:

```
npm install <package-name>
```

Часто можно видеть, как эту команду используют не в таком вот простом виде, а с некоторыми флагами. Рассмотрим их:

- Флаг `--save` позволяет установить пакет и добавить запись о нём в раздел `dependencies` файла `package.json`, который описывает зависимости проекта. Эти зависимости используются проектом для реализации его основного функционала, они устанавливаются в ходе его развёртывания на сервере (после выхода `npm 5` записи об устанавливаемых пакетах в разделе зависимостей делаются автоматически, и без использования этого флага).
- Флаг `--save-dev` позволяет установить пакет и добавить запись о нём в раздел, содержащий перечень зависимостей разработки (то есть — пакетов, которые нужны в ходе разработки проекта, вроде библиотек для тестирования, но не требуются для его работы) файла `package.json`, который называется `devDependencies`.

Обновление пакетов

Для обновления пакетов служит следующая команда:

```
npm update
```

Получив эту команду, `npm` проверит все пакеты на наличие их новых версий, и, если найдёт их новые версии, соответствующие ограничениям на версии пакетов, заданным в `package.json`, установит их.

Обновить можно и отдельный пакет:

```
npm update <package-name>
```

Загрузка пакетов определённых версий

В дополнение к стандартной загрузке пакетов, `npm` поддерживает и загрузку их определённых версий. В частности, можно заметить, что некоторые библиотеки совместимы лишь с некими крупными релизами других библиотек, то есть, если бы зависимости таких библиотек устанавливались бы без учёта версий, это могло бы нарушить их работу. Возможность установить определённую версию некоего пакета полезна и в ситуациях, когда, например, вам вполне подходит самый свежий релиз этого пакета, но оказывается, что в нём имеется ошибка. Ожидая выхода исправленной версии пакета, можно воспользоваться и его более старым но стабильным релизом.

Возможность задавать конкретные версии необходимых проекту библиотек полезна в командной разработке, когда все члены команды пользуются в точности одними и теми же библиотеками. Переход

на их новые версии так же осуществляется централизованно, путём внесения изменений в файл проекта `package.json`.

Во всех этих случаях возможность указания версий пакетов, необходимых проекту, чрезвычайно полезна. Npm следует стандарту семантического версионирования (semver).

Запуск скриптов

Файл `package.json` поддерживает возможность описания команд (скриптов), запускать которые можно с помощью такой конструкции:

```
npm <task-name>
```

Например, вот как выглядят перечень скриптов, имеющийся в соответствующем разделе файла:

```
{
  "scripts": {
    "start-dev": "node lib/server-development",
    "start": "node lib/server-production"
  }
}
```

Весьма распространено использование этой возможности для запуска Webpack:

```
{
  "scripts": {
    "watch": "webpack --watch --progress --colors --config webpack.conf.js",
    "dev": "webpack --progress --colors --config webpack.conf.js",
    "prod": "NODE_ENV=production webpack -p --config webpack.conf.js",
  }
}
```

Такой подход даёт возможность заменить ввод длинных команд, чреватый ошибками, следующими простыми конструкциями:

```
$ npm watch
```

```
$ npm dev
```

```
$ npm prod
```

Куда npm устанавливает пакеты?

При установке пакетов с использованием npm (или [yarn](#)) доступны два варианта установки: локальная и глобальная.

По умолчанию, когда для установки пакета используют команду наподобие `npm install lodash`, пакет оказывается в папке `node_modules`, расположенной в папке проекта. Кроме того, если была

выполнена вышеописанная команда, npm также добавит запись о библиотеке `lodash` в раздел `dependencies` файла `package.json`, который имеется в текущей директории.

Глобальная установка пакетов выполняется с использованием флага `-g`:

```
npm install -g lodash
```

Выполняя такую команду, npm не устанавливает пакет в локальную папку проекта. Вместо этого он копирует файлы пакета в некое глобальное расположение. Куда именно попадают эти файлы?

Для того чтобы это узнать, воспользуйтесь следующей командой:

```
npm root -g
```

В macOS или Linux файлы пакетов могут оказаться в директории `/usr/local/lib/node_modules`. В Windows это может быть нечто вроде `C:\Users\YOU\AppData\Roaming\npm\node_modules`.

Однако если вы используете для управления версиями Node.js `nvm`, путь к папке с глобальными пакетами может измениться.

Я, например, использую `nvm`, и вышеописанная команда сообщает мне о том, что глобальные пакеты устанавливаются по такому адресу:

```
/Users/flavio/.nvm/versions/node/v8.9.0/lib/node_modules.
```

Использование и выполнение пакетов, установленных с помощью npm

Как использовать модули, установленные с помощью npm, локально или глобально, попадающие в папки `node_modules`? Предположим, вы установили популярную библиотеку `lodash`, содержащую множество вспомогательных функций, используемых в JavaScript-разработке:

```
npm install lodash
```

Такая команда установит библиотеку в локальную папку проекта `node_modules`.

Для того чтобы использовать её в своём коде, достаточно импортировать её с применением команды `require`:

```
const _ = require('lodash')
```

Как быть, если пакет представляет собой исполняемый файл?

В таком случае исполняемый файл попадёт в папку `node_modules/.bin/` folder.

Посмотреть на то, как выглядит работа этого механизма можно, установив пакет [cowsay](#). Он представляет собой шуточную программу, написанную для командной строки. Если передать этому пакету какой-нибудь текст, в консоли, в стиле ASCII-арта, будет выведено изображение коровы, которая «произносит» соответствующий текст. «Озвучивать» текст могут и другие существа.

Итак, после установки пакета с использованием команды `npm install cowsay`, он, вместе со своими зависимостями, попадёт в `node_modules`. А в скрытую папку `.bin` будут записаны символические ссылки на бинарные файлы `cowsay`.

Как их выполнять?

Конечно, можно, для вызова программы, ввести в терминале нечто вроде

`./node_modules/.bin/cowsay`, это рабочий подход, но гораздо лучше воспользоваться [npx](#), средством для запуска исполняемых файлов npm-пакетов, включаемым в npm начиная с версии 5.2. А именно, в нашем случае понадобится такая команда:

```
npx cowsay
```

Путь к пакету прх найдёт автоматически.

Файл `package.json`

Файл `package.json` является важнейшим элементов множества проектов, основанных на экосистеме Node.js. Если вы программировали на JavaScript, была ли это серверная или клиентская разработка, то вы, наверняка, уже встречались с этим файлом. Зачем он нужен? Что вам следует о нём знать и какие возможности он вам даёт?

`Package.json` представляет собой нечто вроде файла-манифеста для проекта. Он даёт в распоряжение разработчика множество разноплановых возможностей. Например, он представляет собой центральный репозиторий настроек для инструментальных средств, используемых в проекте. Кроме того, он является тем местом, куда [npm](#) и [yarn](#) записывают сведения об именах и версиях установленных пакетов.

Структура файла

Вот пример простейшего файла `package.json`:

```
{  
  
}
```

Как видите, он пуст. Нет жёстких требований, касающихся того, что должно присутствовать в подобном файле для некоего приложения. Единственное требование к структуре файла заключается в том, что она должна следовать правилам формата JSON. В противном случае этот файл не сможет быть прочитан программами, которые попытаются получить доступ к его содержимому.

Если вы создаёте Node.js-пакет, который собираетесь распространять через npm, то всё радикальным образом меняется, и в вашем `package.json` должен быть набор свойств, которые помогут другим людям пользоваться пакетом. Подробнее мы поговорим об этом позже.

Вот ещё один пример `package.json`:

```
{  
  
  "name": "test-project"  
  
}
```

В нём задано свойство `name`, значением которого является имя приложения или пакета, материалы которого содержатся в той же папке, где находится этот файл.

Вот пример посложнее, который я взял из приложения-примера, написанного с использованием Vue.js:

```
{  
  
  "name": "test-project",  
  
  "version": "1.0.0",  
  
  "description": "A Vue.js project",  
  
  "main": "src/main.js",  
  
  "private": true,
```

```
"scripts": {  
  "dev": "webpack-dev-server --inline --progress --config  
build/webpack.dev.conf.js",  
  "start": "npm run dev",  
  "unit": "jest --config test/unit/jest.conf.js --coverage",  
  "test": "npm run unit",  
  "lint": "eslint --ext .js,.vue src test/unit",  
  "build": "node build/build.js"  
},  
"dependencies": {  
  "vue": "^2.5.2"  
},  
"devDependencies": {  
  "autoprefixer": "^7.1.2",  
  "babel-core": "^6.22.1",  
  "babel-eslint": "^8.2.1",  
  "babel-helper-vue-jsx-merge-props": "^2.0.3",  
  "babel-jest": "^21.0.2",  
  "babel-loader": "^7.1.1",  
  "babel-plugin-dynamic-import-node": "^1.2.0",  
  "babel-plugin-syntax-jsx": "^6.18.0",  
  "babel-plugin-transform-es2015-modules-commonjs": "^6.26.0",  
  "babel-plugin-transform-runtime": "^6.22.0",  
  "babel-plugin-transform-vue-jsx": "^3.5.0",  
  "babel-preset-env": "^1.3.2",  
  "babel-preset-stage-2": "^6.22.0",  
  "chalk": "^2.0.1",  
  "copy-webpack-plugin": "^4.0.1",  
  "css-loader": "^0.28.0",  
  "eslint": "^4.15.0",  
  "eslint-config-airbnb-base": "^11.3.0",
```

```
"eslint-friendly-formatter": "^3.0.0",  
"eslint-import-resolver-webpack": "^0.8.3",  
"eslint-loader": "^1.7.1",  
"eslint-plugin-import": "^2.7.0",  
"eslint-plugin-vue": "^4.0.0",  
"extract-text-webpack-plugin": "^3.0.0",  
"file-loader": "^1.1.4",  
"friendly-errors-webpack-plugin": "^1.6.1",  
"html-webpack-plugin": "^2.30.1",  
"jest": "^22.0.4",  
"jest-serializer-vue": "^0.3.0",  
"node-notifier": "^5.1.2",  
"optimize-css-assets-webpack-plugin": "^3.2.0",  
"ora": "^1.2.0",  
"portfinder": "^1.0.13",  
"postcss-import": "^11.0.0",  
"postcss-loader": "^2.0.8",  
"postcss-url": "^7.2.1",  
"rimraf": "^2.6.0",  
"semver": "^5.3.0",  
"shelljs": "^0.7.6",  
"uglifyjs-webpack-plugin": "^1.1.1",  
"url-loader": "^0.5.8",  
"vue-jest": "^1.0.2",  
"vue-loader": "^13.3.0",  
"vue-style-loader": "^3.0.1",  
"vue-template-compiler": "^2.5.2",  
"webpack": "^3.6.0",  
"webpack-bundle-analyzer": "^2.9.0",  
"webpack-dev-server": "^2.9.1",
```

```

    "webpack-merge": "^4.1.0"
  },
  "engines": {
    "node": ">= 6.0.0",
    "npm": ">= 3.0.0"
  },
  "browserslist": [
    "> 1%",
    "last 2 versions",
    "not ie <= 8"
  ]
}

```

Как видите, тут прямо-таки немеряно всего интересного. А именно, здесь можно выделить следующие свойства:

- `name` — задаёт имя приложения (пакета).
- `version` — содержит сведения о текущей версии приложения.
- `description` — краткое описание приложения.
- `main` — задаёт точку входа в приложение.
- `private` — если данное свойство установлено в `true`, это позволяет предотвратить случайную публикацию пакета в `npm`.
- `scripts` — задаёт набор Node.js-скриптов, которые можно запускать.
- `dependencies` — содержит список `npm`-пакетов, от которых зависит приложение.
- `devDependencies` — содержит список `npm`-пакетов, используемых при разработке проекта, но не при его реальной работе.
- `engines` — задаёт список версий Node.js, на которых работает приложение.
- `browserslist` — используется для хранения списка браузеров (и их версий), которые должно поддерживать приложение.

Все эти свойства используются либо `npm` либо другими инструментальными средствами, применяемыми в течение жизненного цикла приложения.

Свойства, используемые в `package.json`

Поговорим о свойствах, которые можно использовать в `package.json`. Здесь мы будем использовать термин «пакет», но всё, что сказано о пакетах, справедливо и для локальных приложений, которые не планируется использовать в роли пакетов.

Большинство свойств, которые мы опишем, используются лишь для нужд [репозитория](#) `npm`, некоторые используются программами, которые взаимодействуют с кодом, вроде того же `npm`.

Свойство `name`

Свойство `name` задаёт имя пакета:

```
"name": "test-project"
```

Имя должно быть короче 214 символов, не должно включать в себя пробелы, должно состоять только из прописных букв, дефисов (-) и символов подчёркивания (_).

Подобные ограничения существуют из-за того, что когда пакет публикуется в npm, его имя используется для формирования URL страницы пакета.

Если вы публиковали код пакета на GitHub, в общем доступе, то хорошим вариантом имени пакета является имя соответствующего GitHub-репозитория.

Свойство *author*

Свойство `author` содержит сведения об авторе пакета:

```
{  
  
  "author": "Flavio Copes <flavio@flaviocopes.com> (https://flaviocopes.com)"  
  
}
```

Оно может быть представлено и в таком формате:

```
{  
  
  "author": {  
  
    "name": "Flavio Copes",  
  
    "email": "flavio@flaviocopes.com",  
  
    "url": "https://flaviocopes.com"  
  
  }  
  
}
```

Свойство *contributors*

Свойство `contributors` содержит массив со сведениями о людях, внёсших вклад в проект:

```
{  
  
  "contributors": [  
  
    "Flavio Copes <flavio@flaviocopes.com> (https://flaviocopes.com)"  
  
  ]  
  
}
```

Это свойство может выглядеть и так:

```
{  
  
  "contributors": [  
  
    {  
  
      "name": "Flavio Copes",  
  
      "email": "flavio@flaviocopes.com",  
  
    }  
  
  ]  
  
}
```

```
    "url": "https://flaviocopes.com"
  }
]
}
```

Свойство *bugs*

В свойстве `bugs` содержится ссылка на баг-трекер проекта, весьма вероятно то, что такая ссылка будет вести на страницу системы отслеживания ошибок GitHub:

```
{
  "bugs": "https://github.com/flaviocopes/package/issues"
}
```

Свойство *homepage*

Свойство `homepage` позволяет задать домашнюю страницу пакета:

```
{
  "homepage": "https://flaviocopes.com/package"
}
```

Свойство *version*

Свойство `version` содержит сведения о текущей версии пакета:

```
"version": "1.0.0"
```

При формировании значения этого свойства нужно следовать правилам [семантического версионирования](#). Это означает, в частности, что номер версии всегда представлен тремя цифрами: `x.x.x`.

Первое число — это мажорная версия пакета, второе — минорная версия, третье — патч-версия.

Изменение этих чисел несёт в себе определённый смысл. Так, релиз пакета, в котором лишь исправляются ошибки, приводит к увеличению значения патч-версии. Если выходит релиз пакета, изменения, внесённые в который, отличаются обратной совместимостью с предыдущим релизом — то меняется минорная версия. В мажорных версиях пакетов могут присутствовать изменения, которые делают эти пакеты несовместимыми с пакетами предыдущих мажорных версий.

Свойство *license*

Свойство `license` содержит сведения о лицензии пакета:

```
"license": "MIT"
```

Свойство *keywords*

Свойство `keywords` содержит массив ключевых слов, имеющих отношение к функционалу пакета:

```
"keywords": [
  "email",
  "machine learning",
```

```
"ai"
```

```
]
```

Правильный подбор ключевых слов помогает людям находить то, что им нужно, при поиске пакетов для решения неких задач, позволяет группировать пакеты и быстро оценивать их возможный функционал при просмотре сайта npm.

Свойство *description*

Свойство `description` содержит краткое описание пакета:

```
"description": "A package to work with strings"
```

Это свойство особенно важно в том случае, если вы планируете публиковать пакет в npm, так как оно позволяет пользователям сайта npm понять предназначение пакета.

Свойство *repository*

Свойство `repository` указывает на то, где находится репозиторий пакета:

```
"repository": "github:flaviocopes/testing",
```

Обратите внимание, что у значения этого свойства имеется префикс `github`. Npm поддерживает префиксы и для некоторых других популярных сервисов подобного рода:

```
"repository": "gitlab:flaviocopes/testing",
```

```
"repository": "bitbucket:flaviocopes/testing",
```

Используемую при разработке пакета систему контроля версий можно задать и в явном виде:

```
"repository": {  
  "type": "git",  
  "url": "https://github.com/flaviocopes/testing.git"  
}
```

Один и тот же пакет может использовать разные системы контроля версий:

```
"repository": {  
  "type": "svn",  
  "url": "..."  
}
```

Свойство *main*

Свойство `main` задаёт точку входа в пакет:

```
"main": "src/main.js"
```

Когда пакет импортируют в приложение, именно здесь будет осуществляться поиск того, что экспортирует соответствующий модуль.

Свойство *private*

Свойство `private`, установленное в `true`, позволяет предотвратить случайную публикацию пакета в `npm`:

```
"private": true
```

Свойство *scripts*

Свойство `scripts` задаёт список скриптов или утилит, которые можно запускать средствами `npm`:

```
"scripts": {  
  
  "dev": "webpack-dev-server --inline --progress --config  
build/webpack.dev.conf.js",  
  
  "start": "npm run dev",  
  
  "unit": "jest --config test/unit/jest.conf.js --coverage",  
  
  "test": "npm run unit",  
  
  "lint": "eslint --ext .js,.vue src test/unit",  
  
  "build": "node build/build.js"  
  
}
```

Эти скрипты являются приложениями командной строки. Запускать их можно с помощью `npm` или `yarn`, выполняя, соответственно, команды вида `npm run XXXX` или `yarn XXXX`, где `XXXX` — имя скрипта. Например, выглядеть это может так:

```
npm run dev
```

Скрипты можно называть так, как вам хочется, делать они могут практически всё, чего может пожелать разработчик.

Свойство *dependencies*

Свойство `dependencies` содержит список `npm`-пакетов, установленных в виде зависимостей пакета:

```
"dependencies": {  
  
  "vue": "^2.5.2"  
  
}
```

При установке пакета с использованием `npm` или `yarn` используются команды такого вида:

```
npm install <PACKAGENAME>
```

```
yarn add <PACKAGENAME>
```

Эти пакеты автоматически добавляются в список зависимостей разрабатываемого пакета.

Свойство *devDependencies*

Свойство `devDependencies` содержит список `npm`-пакетов, установленных как зависимости разработки:

```
"devDependencies": {
```

```
"autoprefixer": "^7.1.2",  
  
"babel-core": "^6.22.1"  
  
}
```

Этот список отличается от того, который хранится в свойстве `dependencies`, так как имеющиеся в нём пакеты устанавливаются лишь в системе разработчика пакета, при практическом использовании пакета они не применяются.

Пакеты попадают в этот список при их установке с помощью `npm` или `yarn`, выполняемой следующим образом:

```
npm install --dev <PACKAGENAME>
```

```
yarn add --dev <PACKAGENAME>
```

Свойство `engines`

Свойство `engines` указывает, какие версии Node.js и других программных продуктов используются для обеспечения работы пакета:

```
"engines": {  
  
  "node": ">= 6.0.0",  
  
  "npm": ">= 3.0.0",  
  
  "yarn": "^0.13.0"  
  
}
```

Свойство `browserlist`

Свойство `browserlist` позволяет сообщить о том, какие браузеры (и их версии) собирается поддерживать разработчик пакета:

```
"browserslist": [  
  
  "> 1%",  
  
  "last 2 versions",  
  
  "not ie <= 8"  
  
]
```

Этим свойством пользуются Babel, Autoprefixer и другие инструменты. Анализ этого списка позволяет им добавлять в пакет только те полифиллы и вспомогательные механизмы, которые нужны для перечисленных браузеров.

Показанное здесь в качестве примера значение свойства `browserlist` означает, что вы хотите поддерживать как минимум 2 мажорные версии всех браузеров с как минимум 1% использования (эти данные берутся с ресурса CanIUse.com), за исключением IE 8 и более старых версий этого браузера (подробнее об этом можно узнать на странице пакета [browserlists](#)).

Хранение в `package.json` настроек для различных программных инструментов

В `package.json` можно хранить настройки для различных вспомогательных инструментов вроде Babel или ESLint.

Каждому из таких инструментов соответствует особое свойство, наподобие `eslintConfig` или `babel`. Подробности об использовании подобных свойств можно найти в документации соответствующих проектов.

О версиях пакетов и семантическом версионировании

В вышеприведённых примерах вы могли видеть, что номера версий пакетов задаются не только в виде обычных чисел, разделённых точками, но и с использованием неких специальных символов. Например, в виде `~3.0.0` или `^0.13.0`. Здесь использованы так называемые спецификаторы версий, которые определяют диапазон версий пакетов, подходящих для использования в нашем пакете.

Учитывая то, что при использовании семантического версионирования все номера версий пакетов состоят из последовательностей, представляющих собой три числа, о смысле которых мы говорили выше, опишем следующие правила использования спецификаторов версий:

- `~`: если вы задаёте версию в виде `~0.13.0` это означает, что вас интересуют лишь патч-релизы пакета. То есть, пакет `0.13.1` вам подойдёт, а `0.14.0` — нет.
- `^`: если номер версии задан в виде `^0.13.0`, это означает, что вам подходят новые патч-версии и минорные версии пакета. То есть, вас устроят версии пакета `0.13.1`, `0.14.0`, и так далее.
- `*`: воспользовавшись этим символом, вы сообщаете системе, что вас устроят любые свежие версии пакета, в том числе — его новые мажорные релизы.
- `>`: подходят любые версии пакета, которые больше заданной.
- `>=`: подходят любые версии пакета, которые равны или больше заданной.
- `<=`: вас устроят пакеты, версии которых равны заданной или меньше её.
- `<`: вас интересуют пакеты, версии которых меньше заданной.
- `=`: вам нужна только заданная версия пакета.
- `-`: используется для указания диапазона подходящих версий, например — `2.1.0 - 2.6.2`.
- `||`: позволяет комбинировать наборы условий, касающихся пакетов. Например это может выглядеть как `< 2.1 || > 2.6`.

Есть и ещё некоторые правила:

- отсутствие дополнительных символов: если используется номер версии пакета без дополнительных символов, это значит, что вашему пакету нужна только заданная версия пакета-зависимости и никакая другая.
- `latest`: указывает на то, что вам требуется самая свежая версия некоего пакета.

Большинство вышеописанных спецификаторов можно комбинировать, например, задавая диапазоны подходящих версий пакетов-зависимостей. Скажем, конструкция вида `1.0.0 || >=1.1.0 <1.2.0` указывает на то, что планируется использовать либо версию пакета `1.0.0`, либо версию, номер которой больше или равен `1.1.0`, но меньше `1.2.0`.

Файл `package-lock.json`

Файл `package-lock.json` используется с момента появления npm версии 5. Он создаётся автоматически при установке Node.js-пакетов. Что это за файл? Возможно, вы не знакомы с ним даже если знали о `package.json`, который существует гораздо дольше него.

Цель этого файла заключается в отслеживании точных версий установленных пакетов, что позволяет сделать разрабатываемый продукт стопроцентно воспроизводимым в его исходном виде даже в случае, если те, кто занимается поддержкой пакетов, их обновили.

Этот файл решает весьма специфическую проблему, которая не решается средствами `package.json`. В `package.json` можно указать, какие обновления некоего пакета вам подходят (патч-версии или минорные версии) с использованием вышеописанных спецификаторов версий.

В Git не коммитят папку `node_modules`, так как обычно она имеет огромные размеры. Когда вы пытаетесь воссоздать проект на другом компьютере, то использование команды `npm install` приведёт к тому, что, если, при использовании спецификатора `~` в применении к версии некоего пакета, вышел его патч-релиз, установлен будет не тот пакет, который использовался при разработке, а именно этот патч-релиз. То же самое касается и спецификатора `^`. Если же при указании версии пакета спецификаторы не использовались, то будет установлена именно его указанная версия и проблема, о которой идёт речь, окажется в такой ситуации неактуальной.

Итак, кто-то пытается инициализировать проект, пользуясь командой `npm install`. При выходе новых версий пакетов окажется, что этот проект отличается от исходного. Даже если, следуя правилам семантического версионирования, минорные релизы и патч-релизы не должны содержать в себе изменений, препятствующих обратной совместимости, все мы знаем, что ошибки способны проникать (и проникают) куда угодно.

Файл `package-lock.json` хранит в неизменном виде сведения о версии каждого установленного пакета и `npm` будет использовать именно эти версии пакетов при выполнении команды `npm install`.

Эта концепция не нова, менеджеры пакетов, применяемые в других языках программирования (вроде менеджера `Composer` в `Python`) используют похожую систему многие годы.

Файл `package-lock.json` нужно отправить в Git-репозиторий, что позволит другим людям скачать его в том случае, если проект является общедоступным, или тогда, когда его разработкой занимается команда программистов, или если вы используете Git для развёртывания проекта.

Версии зависимостей будут обновлены в `package-lock.json` после выполнения команды `npm update`.

Пример файла `package-lock.json`

В этом примере продемонстрирована структура файла `package-lock.json`, который входит в состав пакет `cowsay`, устанавливаемого в пустой папке командой `npm install cowsay`:

```
{
  "requires": true,
  "lockfileVersion": 1,
  "dependencies": {
    "ansi-regex": {
      "version": "3.0.0",
      "resolved": "https://registry.npmjs.org/ansi-regex/-/ansi-regex-3.0.0.tgz",
      "integrity": "sha1-7QMXwyIGT3lGbAKWa922Bas32Zg="
    },
    "cowsay": {
      "version": "1.3.1",
      "resolved": "https://registry.npmjs.org/cowsay/-/cowsay-1.3.1.tgz"
```

```
,
  "integrity": "sha512-3PVFe6FePVtPj1HTeLin9v8WyLl+VmM111H/5P+BTTDkMAjufp+0F9eLjzRnOHZVAYeIYFF5po5NjRrgef nRMQ==",
  "requires": {
    "get-stdin": "^5.0.1",
    "optimist": "~0.6.1",
    "string-width": "~2.1.1",
    "strip-eof": "^1.0.0"
  },
  "get-stdin": {
    "version": "5.0.1",
    "resolved": "https://registry.npmjs.org/get-stdin/-/get-stdin-5.0.1.tgz",
    "integrity": "sha1-Ei4WFZHiH/TFJTAwVpPyDmOT05g="
  },
  "is-fullwidth-code-point": {
    "version": "2.0.0",
    "resolved": "https://registry.npmjs.org/is-fullwidth-code-point/-/is-fullwidth-code-point-2.0.0.tgz",
    "integrity": "sha1-o7MKXE8ZkYMWeqq5O+764937ZU8="
  },
  "minimist": {
    "version": "0.0.10",
    "resolved": "https://registry.npmjs.org/minimist/-/minimist-0.0.10.tgz",
    "integrity": "sha1-3j+YVD2/lggr5IrRoMfNqDYwHc8="
  },
  "optimist": {
    "version": "0.6.1",
```

```
"resolved": "https://registry.npmjs.org/optimist/-/optimist-0.6.1.tgz",
"integrity": "sha1-2j6nRob6IaGaERwybpDrFaAZZoY=",
"requires": {
  "minimist": "~0.0.1",
  "wordwrap": "~0.0.2"
},
"string-width": {
  "version": "2.1.1",
  "resolved":
"https://registry.npmjs.org/string-width/-/string-width-2.1.1.tgz",
  "integrity":
"sha512-n0QqH59deCq9SRHlxq1Aw85Jnt4w6KvLKqWVik6oA9ZklXLNIOlqg4F2yrT1MVaTjAqvVwdf
eZ7w7aCvJD7ugkw==",
  "requires": {
    "is-fullwidth-code-point": "^2.0.0",
    "strip-ansi": "^4.0.0"
  },
  "strip-ansi": {
    "version": "4.0.0",
    "resolved":
"https://registry.npmjs.org/strip-ansi/-/strip-ansi-4.0.0.tgz",
    "integrity": "sha1-qEeQIusaw2iocTibY1JixQXuNo8=",
    "requires": {
      "ansi-regex": "^3.0.0"
    },
    "strip-eof": {
      "version": "1.0.0",
      "resolved": "https://registry.npmjs.org/strip-eof/-/strip-eof-1.0.0.tgz",
      "integrity": "sha1-u0P/VZim6wXYmIn80SnJgzE2Br8="
    }
  }
}
```

```

    },
    "wordwrap": {
      "version": "0.0.3",
      "resolved": "https://registry.npmjs.org/wordwrap/-/wordwrap-0.0.3.tgz",
      "integrity": "sha1-o9XabNXAvAAI03I0u68b7WMFkQc="
    }
  }
}

```

Разберём этот файл. Мы устанавливаем пакет `cowsay`, который зависит от следующих пакетов:

- `get-stdin`
- `optimist`
- `string-width`
- `strip-eof`

Эти пакеты, в свою очередь, зависят от других пакетов, сведения о которых мы можем почерпнуть из свойств `requires`, которые имеются у некоторых из них:

- `ansi-regex`
- `is-fullwidth-code-point`
- `minimist`
- `wordwrap`
- `strip-eof`

Они добавляются в файл в алфавитном порядке, у каждого есть поле `version`, есть поле `resolved`, указывающее на расположение пакета, и строковое свойство `integrity`, которое можно использовать для проверки целостности пакета.

Часть 5: npm и prx

Выяснение версий установленных npm-пакетов

Для того чтобы узнать версии всех установленных в папке проекта npm-пакетов, включая их зависимости, выполните следующую команду:

```
npm list
```

В результате, например, может получиться следующее:

```
> npm list
```

```
/Users/flavio/dev/node/cowsay
```

```
└─ cowsay@1.3.1
```

```
  └─ get-stdin@5.0.1
```

```
    └─ optimist@0.6.1
```

```
| └─ minimist@0.0.10
|   └─ wordwrap@0.0.3
└─ string-width@2.1.1
    | └─ is-fullwidth-code-point@2.0.0
    |   └─ strip-ansi@4.0.0
    |     └─ ansi-regex@3.0.0
    └─ strip-eof@1.0.0
```

То же самое можно узнать и просмотрев файл `package-lock.json` проекта, но древовидную структуру, которую выводит вышеописанная команда, удобнее просматривать.

Для того чтобы получить подобный список пакетов, установленных глобально, можно воспользоваться следующей командой:

```
npm list -g
```

Вывести только сведения о локальных пакетах верхнего уровня (то есть, тех, которые вы устанавливали самостоятельно и которые перечислены в `package.json`) можно так:

```
npm list --depth=0
```

В результате, если, например, устанавливали вы только пакет `cowsay`, выведено будет следующее:

```
> npm list --depth=0
/Users/flavio/dev/node/cowsay
└─ cowsay@1.3.1
```

Для того чтобы узнать версию конкретного пакета, воспользуйтесь следующей командой:

```
npm list cowsay
```

В результате её выполнения получится примерно следующее:

```
> npm list cowsay
/Users/flavio/dev/node/cowsay
└─ cowsay@1.3.1
```

Эта команда подходит и для выяснения версий зависимостей установленных вами пакетов. При этом в качестве имени пакета, передаваемого ей, выступает имя пакета-зависимости, а вывод команды будет выглядеть следующим образом:

```
> npm list minimist
/Users/flavio/dev/node/cowsay
└─ cowsay@1.3.1
  └─ optimist@0.6.1
```



```
└─ minimist@0.0.10
```

Запись о пакете-зависимости в этой структуре будет выделена.

Если вы хотите узнать о том, каков номер самой свежей версии некоего пакета, доступного в npm-репозитории, вам понадобится команда следующего вида:

```
npm view [package_name] version
```

В ответ она выдаёт номер версии пакета:

```
> npm view cowsay version
```

```
1.3.1
```

Установка старых версий npm-пакетов

Установка старой версии npm-пакета может понадобиться для решения проблем совместимости. Установить нужную версию пакета из npm можно, воспользовавшись следующей конструкцией:

```
npm install <package>@<version>
```

В случае с используемым нами в качестве примера пакетом `cowsay`, команда `npm install cowsay` установит его самую свежую версию (1.3.1 на момент написания этого материала). Если же надо установить его версию 1.2.0, воспользуемся такой командой:

```
npm install cowsay@1.2.0
```

Указывать версии можно и устанавливая глобальные пакеты:

```
npm install -g webpack@4.16.4
```

Если вам понадобится узнать о том, какие версии некоего пакета имеются в npm, сделать это можно с помощью такой конструкции:

```
npm view <package> versions
```

Вот пример результата её работы:

```
> npm view cowsay versions
```

```
[ '1.0.0',  
  '1.0.1',  
  '1.0.2',  
  '1.0.3',  
  '1.1.0',  
  '1.1.1',  
  '1.1.2',  
  '1.1.3',  
  '1.1.4',  
  '1.1.5',
```

```
'1.1.6',  
'1.1.7',  
'1.1.8',  
'1.1.9',  
'1.2.0',  
'1.2.1',  
'1.3.0',  
'1.3.1' ]
```

Обновление зависимостей проекта до их самых свежих версий

Когда вы устанавливаете пакет командой вида `npm install <packagename>`, из репозитория загружается самая свежая из доступных версий и помещается в папку `node_modules`. При этом соответствующие записи добавляются в файлы `package.json` и `package-lock.json`, находящиеся в папке проекта.

Кроме того, устанавливая некий пакет, `npm` находит и устанавливает его зависимости.

Предположим, мы устанавливаем уже знакомый вам пакет `cowsay`, выполняя команду `npm install cowsay`. Пакет будет установлен в папку `node_modules` проекта, а в файл `package.json` попадёт следующая запись:

```
{  
  "dependencies": {  
    "cowsay": "^1.3.1"  
  }  
}
```

В `package-lock.json` так же будут внесены сведения об этом пакете. Вот его фрагмент:

```
{  
  "requires": true,  
  "lockfileVersion": 1,  
  "dependencies": {  
    "cowsay": {  
      "version": "1.3.1",  
      "resolved": "https://registry.npmjs.org/cowsay/-/cowsay-1.3.1.tgz",  
      "integrity":  
"sha512-3PVFe6FePVtPj1HTeLin9v8WyLl+VmM1l1H/5P+BTtDkMAjufp+0F9eLjzRnOHzVAYeIYFF5po5NjRrgef nRMQ==",  
    }  
  }  
}
```

```

"requires": {
  "get-stdin": "^5.0.1",
  "optimist": "~0.6.1",
  "string-width": "~2.1.1",
  "strip-eof": "^1.0.0"
}
}
}
}
}

```

Из этих двух файлов можно узнать, что мы установили cowsay версии 1.3.1, и то, что правило обновления пакета указано в виде `^1.3.1`. В четвёртой части этой серии материалов мы уже говорили о правилах семантического версионирования. Напомним, что такая запись означает, что npm может обновлять пакет при выходе его минорных и патч-версий.

Если, например, выходит новая минорная версия пакета и мы выполняем команду `npm update`, то обновляется установленная версия пакета и при этом сведения об установленном пакете обновляются в файле `package-lock.json`, а файл `package.json` остаётся неизменным.

Для того чтобы узнать, вышли ли новые версии используемых в проекте пакетов, можно воспользоваться следующей командой:

```
npm outdated
```

Вот результаты выполнения этой команды для проекта, зависимости которого давно не обновлялись:

```

~/www/flaviocopes.com/themes/ghostwriter master* 1d 31m 26s
> npm outdated
Package      Current  Wanted  Latest  Location
autoprefixer  8.6.5    8.6.5    9.1.0    ghostwriter
css-loader    0.28.4   0.28.11  1.0.0    ghostwriter
cssnano       3.10.0   3.10.0   4.0.5    ghostwriter
extract-text-webpack-plugin  2.1.2    2.1.2    3.0.2    ghostwriter
node-sass     4.5.3    4.9.2    4.9.2    ghostwriter
normalize.css  7.0.0    7.0.0    8.0.0    ghostwriter
optimize-css-assets-webpack-plugin  2.0.0    2.0.0    5.0.0    ghostwriter
postcss-cli   5.0.1    5.0.1    6.0.0    ghostwriter
postcss-discard-comments  2.0.4    2.0.4    4.0.0    ghostwriter
sass-loader   6.0.6    6.0.7    7.1.0    ghostwriter
style-loader   0.18.2   0.18.2   0.21.0   ghostwriter
webpack       3.0.0    3.12.0   4.16.4   ghostwriter

~/www/flaviocopes.com/themes/ghostwriter master*
> 

```

Анализ устаревших зависимостей проекта

Некоторые из доступных обновлений пакетов представляют собой их мажорные релизы, обновления до которых не произойдёт при выполнении команды `npm update`. Обновление до мажорных релизов этой командой не производится, так как они (по определению) могут содержать серьёзные изменения, не отличающиеся обратной совместимостью с предыдущими мажорными релизами, а npm стремится избавить разработчика от проблем, которые может вызвать использование подобных пакетов.

Для того чтобы обновиться до новых мажорных версий всех используемых пакетов, глобально установите пакет `npm-check-updates`:

```
npm install -g npm-check-updates
```

Затем запустите утилиту, предоставляемую им:

```
ncu -u
```

Эта команда обновит файл `package.json`, внося изменения в указания о подходящих версиях пакетов в разделы `dependencies` и `devDependencies`. Это позволит npm обновить пакеты, используемые в проекте, до новых мажорных версий после запуска команды `npm update`.

Если вы хотите установить самые свежие версии пакетов для только что только что загруженного проекта, в котором пока нет папки `node_modules`, то, вместо `npm update`, выполните команду `npm install`.

Локальная или глобальная деинсталляция пакетов

Для того чтобы деинсталлировать пакет, ранее установленный локально (с использованием команды `install <package-name>`), выполните команду следующего вида:

```
npm uninstall <package-name>
```

Если пакет установлен глобально, то для его удаления нужно будет воспользоваться флагом `-g` (`--global`). Например, подобная команда может выглядеть так:

```
npm uninstall -g webpack
```

При выполнении подобной команды текущая папка значения не имеет.

О выборе между глобальной и локальной установкой пакетов

Когда и почему пакеты лучше всего устанавливать глобально? Для того чтобы ответить на этот вопрос, вспомним о том, чем различаются локальная и глобальная установка пакетов:

- Локальные пакеты устанавливаются в директорию, в которой выполняют команду вида `npm install <package-name>`. Такие пакеты размещаются в папке `node_modules`, находящейся в этой директории.
- Глобальные пакеты устанавливаются в особую папку (в какую точно — зависит от конкретных настроек вашей системы) вне зависимости от того, где именно выполняют команду вида `npm install -g <package-name>`.

Подключение локальных и глобальных пакетов в коде осуществляется одинаково:

```
require('package-name')
```

Итак, какой же способ установки пакетов лучше всего использовать?

В общем случае, все пакеты следует устанавливать локально. Благодаря этому, даже если у вас имеются десятки Node.js-проектов, можно обеспечить, при необходимости, использование ими различных версий одних и тех же пакетов.

Обновление глобального пакета приводит к тому, что все проекты, в которых он применяется, будут использовать его новый релиз. Несложно понять, что это, в плане поддержки проектов, может привести к настоящему кошмару, так как новые релизы некоторых пакетов могут оказаться несовместимыми с их старыми версиями.

Если у каждого проекта имеется собственная локальная версия некоего пакета, даже при том, что подобное может показаться пустой тратой ресурсов, это — очень небольшая плата за возможность избежать негативных последствий, которые могут быть вызваны несовместимостью новых версий пакетов, обновляемых централизованно, с кодом проектов.

Пакеты следует устанавливать глобально в том случае, когда они представляют собой некие утилиты, вызываемые из командной строки, которые используются во множестве проектов.

Подобные пакеты можно устанавливать и локально, запуская предоставляемые ими утилиты командной строки с использованием `prx`, но некоторые пакеты, всё же, лучше устанавливать глобально. К таким пакетам, которые вам, вполне возможно, знакомы, можно отнести, например, следующие:

- `npm`
- `create-react-app`
- `vue-cli`
- `grunt-cli`
- `mocha`
- `react-native-cli`
- `gatsby-cli`
- `forever`
- `nodemon`

Не исключено, что в вашей системе уже имеются пакеты, установленные глобально. Для того чтобы об этом узнать, воспользуйтесь следующей командой:

```
npm list -g --depth 0
```

О зависимостях проектов

Когда пакет следует рассматривать как обычную зависимость проекта, необходимую для обеспечения его функционирования, а когда — как зависимость разработки?

При установке пакета с помощью команды вида `npm install <package-name>` он устанавливается как обычная зависимость. Запись о таком пакете делается в разделе `dependencies` файла `package.json` (до выхода `npm` 5 такая запись делалась только при использовании флага `--save`, теперь использовать его для этого необязательно).

Использование флага `--save-dev` позволяет установить пакет как зависимость разработки. Запись о нём при этом делается в разделе `devDependencies` файла `package.json`.

Зависимости разработки — это пакеты, которые нужны в процессе разработки проекта, в ходе его обычного функционирования они не требуются. К таким пакетам относятся, например инструменты тестирования, `Webpack`, `Babel`.

Когда проект разворачивают, используя команду `npm install` в его папке, в которой имеется папка, содержащая файл `package.json`, это приведёт к установке всех зависимостей, так как `npm` предполагает, что подобная установка выполняется для целей работы над проектом.

Поэтому, если пакет требуется развернуть в продакшне, то, при его развёртывании, нужно использовать команду `npm install --production`. Благодаря флагу `--production` зависимости разработки устанавливаться не будут.

Утилита `npx`

Сейчас мы поговорим об одной весьма мощной команде, [npx](#), которая появилась в `npm` 5.2. Одной из её возможностей является запуск исполняемых файлов, входящих в состав `npm`-пакетов. Мы уже рассматривали использование `npx` для запуска подобного файла из пакета `cowsay`. Теперь поговорим об этом подробнее.

Использование `npx` для упрощения запуска локальных команд

Node.js-разработчики опубликовали множество исполняемых файлов (утилит) в виде пакетов, которые предполагалось устанавливать глобально, что обеспечивало удобный доступ к их возможностям, так как запускать их из командной строки можно было, просто введя имя соответствующей команды. Однако работать в такой среде было весьма некомфортно в том случае, если требовалось устанавливать разные версии одних и тех же пакетов.

Применение команды вида `npx commandname` приводит к автоматическому поиску нужного файла в папке проекта `node_modules`. Это избавляет от необходимости знания точного пути к подобному файлу. Так же это делает ненужной глобальную установку пакета с обеспечением доступа к нему из любого места файловой системы благодаря использованию системной переменной `PATH`.

Выполнение утилит без необходимости их установки

В `npx` имеется ещё одна интереснейшая возможность, благодаря которой утилиты можно запускать без их предварительной установки. Полезно это, в основном, по следующим причинам:

- Не требуется установка утилит.
- Можно запускать разные версии одних и тех же утилит, указывая нужную версию с помощью конструкции `@version`.

Посмотрим на то, как пользоваться этим механизмом, на примере уже известной вам утилиты `cowsay`. Так, если пакет `cowsay` установлен глобально, выполнение в командной строке команды `cowsay "Hello"` приведёт к выводу в консоль «говорящей» коровы:

```
_____
< Hello >
-----
      \   ^__^
      \  (oo)\_______
          (__) \       )\/\
              ||----w |
              ||     ||
```

Если же пакет `cowsay` не будет установлен глобально, подобная команда выдаст ошибку.

Утилита `npx` позволяет выполнять подобные команды без их установки. Выглядит это, в рамках нашего примера, так:

```
npx cowsay "Hello"
```

Такая команда сработает, но, хотя «говорящая» корова, по большому счёту, особой пользы не приносит, тот же самый подход можно использовать и для выполнения куда более полезных команд. Вот несколько примеров:

- Существует инструмент командной строки, предназначенный для создания и запуска Vue-приложений. С использованием `npx` его можно вызвать так: `npx vue create my-vue-app`.
- Для создания React-приложений можно пользоваться утилитой `create-react-app`. Её вызов через `npx` выглядит так: `npx create-react-app my-react-app`.

После загрузки и использования соответствующего кода `npx` его удалит.

Запуск JavaScript-кода с использованием различных версий Node.js

Для того чтобы запускать некий код с использованием разных версий Node.js, можно, с использованием `npx`, обращаться к npm-пакету `node`, указывая его версию. Выглядит это так:

```
npx node@6 -v #v6.14.3
```

```
npx node@8 -v #v8.11.3
```

Это позволяет отказаться от использования инструментов наподобие `nvm` или других менеджеров версий Node.js.

Запуск произвольных фрагментов кода, доступных по некоему адресу

`Npx` позволяет запускать не только код, опубликованный в npm. В частности, если у вас есть ссылка на некий фрагмент кода (скажем, опубликованного на [GitHub gist](#)), запустить его можно так:

```
npx https://gist.github.com/zkat/4bc19503fe9e9309e2bfaa2c58074d32
```

Конечно, при выполнении подобного кода нельзя забывать о безопасности. `Npx` даёт в руки разработчика большие возможности, но они означают и большую ответственность.

Часть 6: цикл событий, стек вызовов, таймеры

Цикл событий

Если вы хотите разобраться с тем, как выполняется JavaScript-код, то цикл событий (Event Loop) — это одна из важнейших концепций, которую необходимо понять. Здесь мы поговорим о том, как JavaScript работает в однопоточном режиме, и о том, как осуществляется обработка асинхронных функций.

Я многие годы занимался разработкой на JavaScript, но не могу сказать, что полностью понимал то, как всё функционирует, так сказать, «под капотом». Программист вполне может не знать о тонкостях устройства внутренних подсистем среды, в которой он работает. Но обычно полезно иметь хотя бы общее представление о подобных вещах.

JavaScript-код, который вы пишете, выполняется в однопоточном режиме. В некий момент времени выполняется лишь одно действие. Это ограничение, на самом деле, является весьма полезным. Это значительно упрощает то, как работают программы, избавляя программистов от необходимости решать проблемы, характерные для многопоточных сред.

Фактически, JS-программисту нужно обращать внимание только на то, какие именно действия выполняет его код, и стараться при этом избежать ситуаций, вызывающих блокировку главного потока. Например — выполнения сетевых вызовов в синхронном режиме и бесконечных [циклов](#).

Обычно в браузерах, в каждой открытой вкладке, имеется собственный цикл событий. Это позволяет выполнять код каждой страницы в изолированной среде и избегать ситуаций, когда некая страница, в коде которой имеется бесконечный цикл или выполняются тяжёлые вычисления, способна «подвесить» весь браузер. Браузер поддерживает работу множества одновременно существующих циклов событий, используемых, например, для обработки вызовов к различным API. Кроме того, собственный цикл событий используется для обеспечения работы [веб-воркеров](#).

Самое важное, что надо постоянно помнить JavaScript-программисту, заключается в том, что его код использует собственный цикл событий, поэтому код надо писать с учётом того, чтобы этот цикл событий не заблокировать.

Блокировка цикла событий

Любой JavaScript-код, на выполнение которого нужно слишком много времени, то есть такой код, который слишком долго не возвращает управление циклу событий, блокирует выполнение любого другого кода страницы. Подобное приводит даже к блокировке обработки событий пользовательского интерфейса, что выражается в том, что пользователь не может взаимодействовать с элементами страницы и нормально с ней работать, например — прокручивать.

Практически все базовые механизмы обеспечения ввода-вывода в JavaScript являются неблокирующими. Это относится и к браузеру и к Node.js. Среди таких механизмов, например, можно отметить средства для выполнения сетевых запросов, используемые и в клиентской и в серверной средах, и средства для работы с файлами Node.js. Существуют и синхронные способы выполнения подобных операций, но их применяют лишь в особых случаях. Именно поэтому в JavaScript огромное значение имеют традиционные коллбэки и более новые механизмы — промисы и конструкция `async/await`.

Стек вызовов

Стек вызовов (Call Stack) в JavaScript устроен по принципу LIFO (Last In, First Out — последним вошёл, первым вышел). Цикл событий постоянно проверяет стек вызовов на предмет того, имеется ли в нём функция, которую нужно выполнить. Если при выполнении кода в нём встречается вызов некоей функции, сведения о ней добавляются в стек вызовов и производится выполнение этой функции.

Если даже раньше вы не интересовались понятием «стек вызовов», то вы, если встречались с сообщениями об ошибках, включающими в себя трассировку стека, уже представляете себе, как он выглядит. Вот, например, как подобное выглядит в браузере.


```
> const bar = () => {  
    throw new DOMException()  
}  
  
const baz = () => console.log('baz')  
  
const foo = () => {  
    console.log('foo')  
    bar()  
    baz()  
}  
  
foo()  
foo
```

✖ **Uncaught DOMException**
bar @ [VM570:2](#)
foo @ [VM570:9](#)
(anonymous) @ [VM570:13](#)

> |

Сообщение об ошибке в браузере

Браузер, при возникновении ошибки, сообщает о последовательности вызовов функций, сведения о которых хранятся в стеке вызовов, что позволяет обнаружить источник ошибки и понять, вызовы каких функций привели к сложившейся ситуации.

Теперь, когда мы в общих чертах поговорили о цикле событий и о стеке вызовов, рассмотрим пример, иллюстрирующий выполнение фрагмента кода, и то, как этот процесс выглядит с точки зрения цикла событий и стека вызовов.

Цикл событий и стек вызовов

Вот код, с которым мы будем экспериментировать:

```
const bar = () => console.log('bar')  
  
const baz = () => console.log('baz')  
  
const foo = () => {  
    console.log('foo')  
  
    bar()
```

```
baz ()
```

```
}
```

```
foo ()
```

Если этот код выполнить, в консоль попадёт следующее:

```
foo
```

```
bar
```

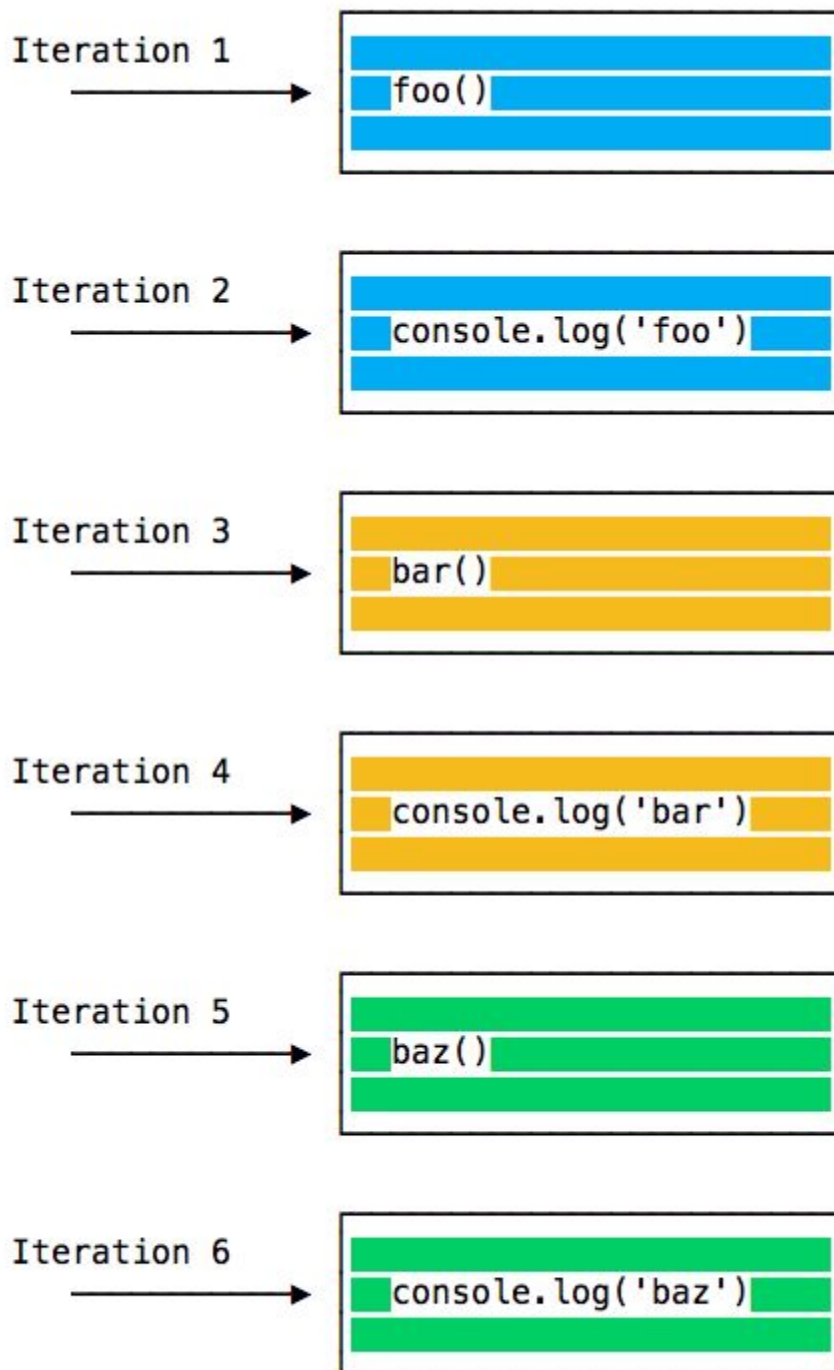
```
baz
```

Такой результат вполне ожидаем. А именно, когда этот код запускают, сначала вызывается функция `foo()`. Внутри этой функции мы сначала вызываем функцию `bar()`, а потом — `baz()`. При этом стек вызовов в ходе выполнения этого кода претерпевает изменения, показанные на следующем рисунке.



Изменение состояния стека вызовов при выполнении исследуемого кода

Цикл событий, на каждой итерации, проверяет, есть ли что-нибудь в стеке вызовов, и если это так — выполняет это до тех пор, пока стек вызовов не опустеет.



Итерации цикла событий

Постановка функции в очередь на выполнение

Вышеприведённый пример выглядит вполне обычным, в нём нет ничего особенного: JavaScript находит код, который надо выполнить, и выполняет его по порядку. Поговорим о том, как отложить выполнение функции до момента очистки стека вызовов. Для того чтобы это сделать, используется такая конструкция:

```
setTimeout(() => {}), 0)
```

Она позволяет выполнить функцию, переданную функции `setTimeout()`, после того, как будут выполнены все остальные функции, вызванные в коде программы.

Рассмотрим пример:

```
const bar = () => console.log('bar')

const baz = () => console.log('baz')

const foo = () => {

  console.log('foo')

  setTimeout(bar, 0)

  baz()

}

foo()
```

То, что выведет этот код, возможно, покажется неожиданным:

```
foo
baz
bar
```

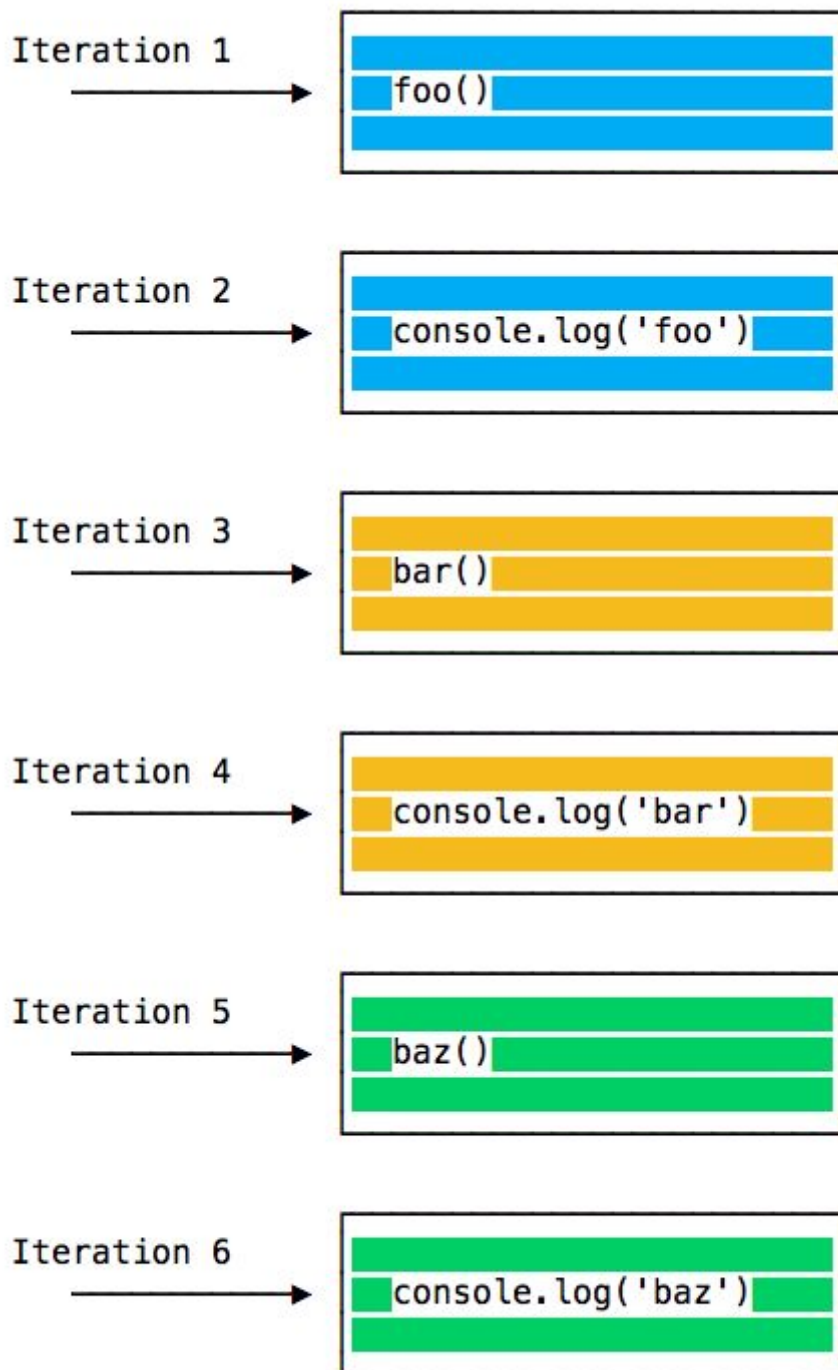
Когда мы запускаем этот пример, сначала вызывается функция `foo()`. В ней мы вызываем `setTimeout()`, передавая этой функции, в качестве первого аргумента, `bar`. Передав ей в качестве второго аргумента `0`, мы сообщаем системе о том, что эту функцию следует выполнить как можно скорее. Затем мы вызываем функцию `baz()`.

Вот как теперь будет выглядеть стек вызовов.



Изменение состояния стека вызовов при выполнении исследуемого кода

Вот в каком порядке теперь будут выполняться функции в нашей программе.



И т е р а ц и и ц и к л а с о б ы т и й

Почему всё происходит именно так?

Очередь событий

Когда вызывается функция `setTimeout()`, браузер или платформа Node.js запускает таймер. После того, как таймер сработает (в нашем случае это происходит немедленно, так как мы установили его на 0), функция обратного вызова, переданная `setTimeout()`, попадает в очередь событий (Event Queue).

В очередь событий, если речь идёт о браузере, попадают и события, инициированные пользователем — события, вызванные щелчками мышью по элементам страницы, события, вызываемые при вводе данных с клавиатуры. Тут же оказываются обработчики событий DOM вроде `onload`, функции,

вызываемые при получении ответов на асинхронные запросы по загрузке данных. Здесь они ждут своей очереди на обработку.

Цикл событий отдаёт приоритет тому, что находится в стеке вызовов. Сначала он выполняет всё, что ему удастся найти в стеке, а после того, как стек оказывается пустым, переходит к обработке того, что находится в очереди событий.

Нам не нужно ждать, пока функция, наподобие `setTimeout()`, завершит работу, так как подобные функции предоставляются браузером и они используют собственные потоки. Так, например, установив с помощью функции `setTimeout()` таймер на 2 секунды, вы не должны, остановив выполнение другого кода, ждать эти 2 секунды, так как таймер работает за пределами вашего кода.

Очередь заданий ES6

В ECMAScript 2015 (ES6) была введена концепция очереди заданий (Job Queue), которой пользуются промисы (они тоже появились в ES6). Благодаря очереди заданий результатом выполнения асинхронной функции можно воспользоваться настолько быстро, насколько это возможно, без необходимости ожидания очищения стека вызовов.

Если промис разрешается до окончания выполнения текущей функции, соответствующий код будет выполнен сразу после того, как текущая функция завершит работу.

Я обнаружил интересную аналогию для того, о чём мы сейчас говорим. Это можно сравнить с американскими горками в парке развлечений. После того, как вы прокатились на горке и хотите сделать это ещё раз, вы берёте билет и становитесь в хвост очереди. Так работает очередь событий. А вот очередь заданий выглядит иначе. Эта концепция похожа на льготный билет, который даёт вам право совершить следующую поездку сразу после того, как вы закончили предыдущую.

Рассмотрим следующий пример:

```
const bar = () => console.log('bar')

const baz = () => console.log('baz')

const foo = () => {
  console.log('foo')

  setTimeout(bar, 0)

  new Promise((resolve, reject) =>
    resolve('should be right after baz, before bar')
  ).then(resolve => console.log(resolve))

  baz()
}

foo()
```

Вот что будет выведено после его выполнения:

```
foo
baz
```

```
should be right after foo, before bar
```

```
bar
```

То, что тут можно видеть, демонстрирует серьёзное различие промисов (и конструкции `async/await`, которая на них основана) и традиционных асинхронных функций, выполнение которых организуется посредством `setTimeout()` или других API используемой платформы.

`process.nextTick()`

Метод `process.nextTick()` по-особому взаимодействует с циклом событий. Тиком (tick) называют один полный проход цикла событий. Передавая функцию методу `process.nextTick()`, мы сообщаем системе о том, что эту функцию нужно вызвать после завершения текущей итерации цикла событий, до начала следующей. Использование данного метода выглядит так:

```
process.nextTick(() => {  
  
    //выполнить какие-то действия  
  
})
```

Предположим, цикл событий занят выполнением кода текущей функции. Когда эта операция завершается, JavaScript-движок выполнит все функции, переданные `process.nextTick()` в ходе выполнения предыдущей операции. Используя этот механизм, мы стремимся к тому, чтобы некая функция была бы выполнена асинхронно (после текущей функции), но как можно скорее, без постановки её в очередь.

Например, если воспользоваться конструкцией `setTimeout(() => {}, 0)` функция будет выполнена на следующей итерации цикла событий, то есть — гораздо позже, чем при использовании в такой же ситуации `process.nextTick()`. Этот метод стоит использовать тогда, когда нужно обеспечить выполнение некоего кода в самом начале следующей итерации цикла событий.

`setImmediate()`

Ещё одной функцией, предоставляемой Node.js для асинхронного выполнения кода, является `setImmediate()`. Вот как ей пользоваться:

```
setImmediate(() => {  
  
    //выполнить некий код  
  
})
```

Функция обратного вызова, переданная `setImmediate()`, будет выполнена на следующей итерации цикла событий.

Чем `setImmediate()` отличается от `setTimeout(() => {}, 0)` (то есть, от таймера, который должен сработать как можно скорее) и от `process.nextTick()`?

Функция, переданная `process.nextTick()` выполнится после завершения текущей итерации цикла событий. То есть, такая функция всегда будет выполняться до функции, выполнение которой запланировано с помощью `setTimeout()` или `setImmediate()`.

Вызов функции `setTimeout()` с установленной задержкой в 0 мс очень похож на вызов `setImmediate()`. Порядок выполнения функций, переданных им, зависит от различных факторов, но и в том и в другом случаях коллбэки будут вызваны на следующей итерации цикла событий.

Таймеры

Выше мы уже говорили о функции `setTimeout()`, которая позволяет планировать вызовы передаваемых ей коллбэков. Уделим некоторое время более подробному описанию её особенностей и рассмотрим ещё одну функцию, `setInterval()`, схожую с ней. В Node.js функции для работы с таймерами входят в модуль [timer](#), но пользоваться ими можно, не подключая этот модуль в коде, так как они являются глобальными.

Функция `setTimeout()`

Напомним, что при вызове функции `setTimeout()` ей передают коллбэк и время, в миллисекундах, по прошествии которого будет вызван коллбэк. Рассмотрим пример:

```
setTimeout(() => {  
    // выполняется через 2 секунды  
}, 2000)  
  
setTimeout(() => {  
    // выполняется через 50 миллисекунд  
}, 50)
```

Здесь мы передаём `setTimeout()` новую функцию, тут же описываемую, но здесь можно использовать и существующую функцию, передавая `setTimeout()` её имя и набор параметров для её запуска. Выглядит это так:

```
const myFunction = (firstParam, secondParam) => {  
    //выполнить некий код  
}  
  
// выполняется через 2 секунды  
  
setTimeout(myFunction, 2000, firstParam, secondParam)
```

Функция `setTimeout()` возвращает идентификатор таймера. Обычно он не используется, но его можно сохранить, и, при необходимости, удалить таймер, если в запланированном выполнении коллбэка больше нет необходимости:

```
const id = setTimeout(() => {  
    // этот код должен выполняться через 2 секунды  
}, 2000)  
  
// Программист передумал, выполнять этот код больше не нужно  
  
clearTimeout(id)
```

Нулевая задержка

В предыдущих разделах мы использовали `setTimeout()`, передавая ей, в качестве времени, по истечении которого надо вызвать коллбэк, 0. Это означало, что коллбэк будет вызван так скоро, как это возможно, но после завершения выполнения текущей функции:

```
setTimeout(() => {
```

```
    console.log('after ')\n  }, 0)\n\n  console.log(' before ')
```

Такой код выведет следующее:

```
before\n\nafter
```

Этот приём особенно полезен в ситуациях, когда, при выполнении тяжёлых вычислительных задач, не хотелось бы блокировать главный поток, позволяя выполняться и другим функциям, разбивая подобные задачи на несколько этапов, оформляемых в виде вызовов `setTimeout()`.

Если вспомнить о вышеупомянутой функции `setImmediate()`, то в Node.js она является стандартной, чего нельзя сказать о браузерах (в IE и Edge она [реализована](#), в других — нет).

Функция `setInterval()`

Функция `setInterval()` похожа на `setTimeout()`, но между ними есть и различия. Вместо однократного выполнения переданного ей коллбэка `setInterval()` будет периодически, с заданным интервалом, вызывать этот коллбэк. Продолжаться это будет, в идеале, до того момента, пока программист явным образом не остановит этот процесс. Вот как пользоваться этой функцией:

```
setInterval(() => {\n\n    // выполняется каждые 2 секунды\n\n}, 2000)
```

Коллбэк, переданный функции, показанной выше, будет вызываться каждые 2 секунды. Для того чтобы предусмотреть возможность остановки этого процесса, нужно получить идентификатор таймера, возвращаемый `setInterval()` и воспользоваться командой `clearInterval()`:

```
const id = setInterval(() => {\n\n    // выполняется каждые 2 секунды\n\n}, 2000)\n\nclearInterval(id)
```

Распространённой методикой является вызов `clearInterval()` внутри коллбэка, переданного `setInterval()` при выполнении некоего условия. Например, следующий код будет периодически запускаться до тех пор, пока свойство `App.somethingIWait` не примет значение `arrived`:

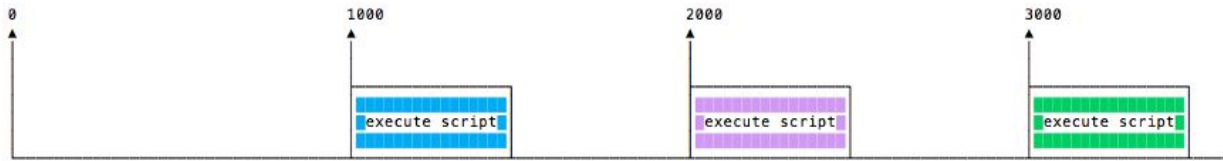
```
const interval = setInterval(function() {\n\n    if (App.somethingIWait === 'arrived') {\n\n        clearInterval(interval)\n\n        // если условие выполняется - удалим таймер, если нет - выполним некие\n        действия\n\n    }\n\n}, 2000)
```

}, 100)

Рекурсивная установка setTimeout()

Функция `setInterval()` будет вызывать переданный ей коллбэк каждые n миллисекунд, не заботясь о том, завершилось ли выполнение этого коллбэка после его предыдущего вызова.

Если на каждый вызов этого коллбэка всегда требуется одно и то же время, меньшее n , то никаких проблем тут не возникает.



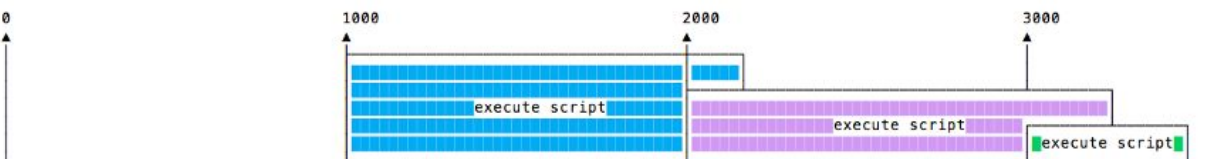
Периодически вызываемый коллбэк, каждый сеанс выполнения которого занимает одно и то же время, укладывающееся в промежуток между вызовами

Возможно, для выполнения коллбэка каждый раз требуется разное время, которое всё ещё меньше n . Если, например, речь идёт о выполнении неких сетевых операций, то такая ситуация вполне ожидаема.



Периодически вызываемый коллбэк, каждый сеанс выполнения которого занимает разное время, укладывающееся в промежуток между вызовами

При использовании `setInterval()` может возникнуть ситуация, когда выполнение коллбэка занимает время, превышающее n , что приводит к тому, что следующий вызов осуществляется до завершения предыдущего.



Периодически вызываемый коллбэк, каждый сеанс выполнения которого занимает разное время, которое иногда не укладывается в промежуток между вызовами

Для того чтобы избежать подобной ситуации, можно воспользоваться методикой рекурсивной установки таймера с помощью `setTimeout()`. Речь идёт о том, что следующий вызов коллбэка планируется после завершения его предыдущего вызова:

```
const myFunction = () => {  
  // выполнить некие действия  
  
  setTimeout(myFunction, 1000)
```

```

}

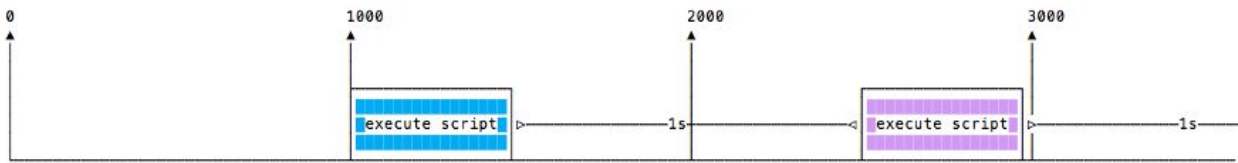
setTimeout (

    myFunction ()

}, 1000)

```

При таком подходе можно реализовать следующий сценарий:



Р е к у р с и в н ы й в ы з о в `setTimeout()` д л я п л а н и р о в а н и я в ы п о л н е н и я к о л л б э к а

Часть 7: асинхронное программирование

Асинхронность в языках программирования

Сам по себе JavaScript — это синхронный однопоточный язык программирования. Это означает, что в коде нельзя создавать новые потоки, выполняющиеся параллельно. Однако компьютеры, по своей природы, асинхронны. То есть некие действия могут выполняться независимо от главного потока выполнения программы. В современных компьютерах каждой программе выделяется некое количество процессорного времени, когда это время истекает, система отдаёт ресурсы другой программе, тоже на некоторое время. Подобные переключения выполняются циклически, делается это настолько быстро, что человек попросту не может этого заметить, в результате мы думаем, что наши компьютеры выполняют множество программ одновременно. Но это иллюзия (если не говорить о многопроцессорных машинах).

В недрах программ используются прерывания — сигналы, передаваемые процессору и позволяющие привлечь внимание системы. Не будем вдаваться в детали, самое главное — помните о том, что асинхронное поведение, когда выполнение программы приостанавливается до того момента, когда ей понадобятся ресурсы процессора, это совершенно нормально. В то время, когда программа не нагружает систему работой, компьютер может решать другие задачи. Например, при таком подходе, когда программа ждёт ответа на выполненный ей сетевой запрос, она не блокирует процессор до момента получения ответа.

Как правило, языки программирования являются асинхронными, некоторые из них дают программисту возможность управлять асинхронными механизмами, пользуясь либо встроенными средствами языка, либо специализированными библиотеками. Речь идёт о таких языках, как C, Java, C#, PHP, Go, Ruby, Swift, Python. Некоторые из них позволяют программировать в асинхронном стиле, используя потоки, запуская новые процессы.

Асинхронность в JavaScript

Как уже было сказано, JavaScript — однопоточный синхронный язык. Строки кода, написанного на JS, выполняются в том порядке, в котором они присутствуют в тексте, друг за другом. Например, вот вполне обычная программа на JS, демонстрирующая такое поведение:

```

const a = 1

const b = 2

```

```
const c = a * b

console.log(c)

doSomething()
```

Но JavaScript был создан для использования в браузерах. Его основной задачей, в самом начале, была организация обработки событий, связанных с деятельностью пользователя. Например — это такие события, как `onClick`, `onMouseOver`, `onChange`, `onSubmit`, и так далее. Как решать подобные задачи в рамках синхронной модели программирования?

Ответ кроется в окружении, в котором работает JavaScript. А именно, эффективно решать подобные задачи позволяет браузер, давая в распоряжение программиста соответствующие API.

В окружении Node.js имеются средства для выполнения неблокирующих операций ввода-вывода, таких, как работа с файлами, организация обмена данными по сети и так далее.

Коллбэки

Если говорить о браузерном JavaScript, то можно отметить, что нельзя заранее узнать, когда пользователь щёлкнет по некоей кнопке. Для того чтобы обеспечить реакцию системы на подобное событие, для него создают обработчик.

Обработчик события принимает функцию, которая будет вызвана при возникновении события. Выглядит это так:

```
document.getElementById('button').addEventListener('click', () => {

    //пользователь щёлкнул по элементу

})
```

Такие функции ещё называют функциями обратного вызова или коллбэками.

Коллбэк — это обычная функция, которая передаётся, как значение, другой функции. Вызвана она будет только в том случае, когда произойдёт некое событие. В JavaScript реализована концепция функций первого класса. Такие функции можно назначать переменным и передавать другим функциям (называемым функциями высшего порядка).

В клиентской JavaScript-разработке распространён подход, когда весь клиентский код оборачивают в прослушиватель события `load` объекта `window`, который вызывает переданный ему коллбэк после того, как страница будет готова к работе:

```
window.addEventListener('load', () => {

    //страница загружена

    //теперь с ней можно работать

})
```

Коллбэки используются повсеместно, а не только для обработки событий DOM. Например, мы уже встречались с их использованием в таймерах:

```
setTimeout(() => {

    // выполнится через 2 секунды

})
```

```
}, 2000)
```

В [XHR-запросах](#) тоже используются коллбэки. В данном случае это выглядит как назначение функции соответствующему свойству. Подобная функция будет вызвана при возникновении определённого события. В следующем примере таким событием является изменение состояния запроса:

```
const xhr = new XMLHttpRequest()

xhr.onreadystatechange = () => {

  if (xhr.readyState === 4) {

    xhr.status === 200 ? console.log(xhr.responseText) : console.error('error')

  }

}

xhr.open('GET', 'https://yoursite.com')

xhr.send()
```

Обработка ошибок в коллбэках

Поговорим о том, как обрабатывать ошибки в коллбэках. Существует одна распространённая стратегия обработки подобных ошибок, которая применяется и в Node.js. Она заключается в том, что первым параметром любой функции обратного вызова делают объект ошибки. При отсутствии ошибок в этот параметр будет записано значение `null`. В противном случае тут будет объект ошибки, содержащий её описание и дополнительные сведения о ней. Вот как это выглядит:

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

  if (err !== null) {

    //обрабатываем ошибку

    console.log(err)

    return

  }

  //ошибок нет, обрабатываем данные

  console.log(data)

})
```

Проблема коллбэков

Коллбэками удобно пользоваться в простых ситуациях. Однако, каждый коллбэк — это дополнительный уровень вложенности кода. Если используется несколько вложенных коллбэков, это быстро приводит к значительному усложнению структуры кода:

```
window.addEventListener('load', () => {

  document.getElementById('button').addEventListener('click', () => {

    setTimeout(() => {
```

```

    items.forEach(item => {

        //код, делающий что-то полезное

    })

    }, 2000)

})

})

```

В этом примере показано всего лишь 4 уровня кода, но на практике можно столкнуться и с большим количеством уровней, обычно это называют «адом коллбэков». Справиться с этой проблемой можно, используя другие языковые конструкции.

Промисы и `async/await`

Начиная со стандарта ES6 в JavaScript появляются новые возможности, которые облегчают написание асинхронного кода, позволяя обходиться без коллбэков. Речь идёт о промисах, которые появились в ES6, и о конструкции `async/await`, появившейся в ES8.

Промисы

Промисы (promise-объекты) — это один из способов работы с асинхронными программными конструкциями в JavaScript, который, в целом, позволяет сократить использование коллбэков.

Знакомство с промисами

Промисы обычно определяют как прокси-объекты для неких значений, появление которых ожидается в будущем. Промисы ещё называют «обещаниями» или «обещанными результатами». Хотя эта концепция существует уже многие годы, промисы были стандартизированы и добавлены в язык лишь в ES2015. В ES2017 появилась конструкция `async/await`, которая основана на промисах, и которую можно рассматривать в качестве их удобной замены. Поэтому, даже если не планируется пользоваться обычными промисами, понимание того, как они работают, важно для эффективного использования конструкции `async/await`.

Как работают промисы

После вызова промиса он переходит в состояние ожидания (pending). Это означает, что функция, вызвавшая промис, продолжает выполняться, при этом в промисе производятся некие вычисления, по завершении которых промис сообщает об этом. Если операция, которую выполняет промис, завершается успешно, то промис переводится в состояние «выполнено» (fulfilled). О таком промисе говорят, что он успешно разрешён. Если операция завершается с ошибкой, промис переводится в состояние «отклонено» (rejected).

Поговорим о работе с промисами.

Создание промисов

API для работы с промисами даёт нам соответствующий конструктор, который вызывают командой вида `new Promise()`. Вот как создают промисы:

```

let done = true

const isItDoneYet = new Promise(

    (resolve, reject) => {

        if (done) {

```

```

    const workDone = 'Here is the thing I built'

    resolve(workDone)

  } else {

    const why = 'Still working on something else'

    reject(why)

  }

}

)

```

Промис проверяет глобальную константу `done`, и, если её значение равно `true`, он успешно разрешается. В противном случае промис отклоняется. Используя параметры `resolve` и `reject`, являющиеся функциями, мы можем возвращать из промиса значения. В данном случае мы возвращаем строку, но тут может использоваться и объект.

Работа с промисами

Выше мы создали промис, теперь рассмотрим работу с ним. Выглядит это так:

```

const isItDoneYet = new Promise(

  //...

)

const checkIfItsDone = () => {

  isItDoneYet

    .then((ok) => {

      console.log(ok)

    })

    .catch((err) => {

      console.error(err)

    })

}

```

```
checkIfItsDone()
```

Вызов `checkIfItsDone()` приведёт к выполнению промиса `isItDoneYet()` и к организации ожидания его разрешения. Если промис разрешится успешно, сработает коллбэк, переданный методу `.then()`. Если возникнет ошибка, то есть промис будет отклонён, обработать её можно будет в функции, переданной методу `.catch()`.

Объединение промисов в цепочки

Методы промисов возвращают промисы, что позволяет объединять их в цепочки. Удачным примером подобного поведения является браузерное [API Fetch](#), представляющее собой уровень абстракции над XMLHttpRequest. Существует довольно популярный npm-пакет для Node.js, реализующий API Fetch, который мы рассмотрим позже. Это API можно использовать для загрузки неких сетевых ресурсов и, благодаря возможности объединения промисов в цепочки, для организации последующей обработки загруженных данных. Фактически, при обращении к API Fetch, выполняемому благодаря вызову функции `fetch()`, создаётся промис.

Рассмотрим следующий пример объединения промисов в цепочки:

```
const fetch = require('node-fetch')

const status = (response) => {

  if (response.status >= 200 && response.status < 300) {

    return Promise.resolve(response)

  }

  return Promise.reject(new Error(response.statusText))

}

const json = (response) => response.json()

fetch('https://jsonplaceholder.typicode.com/todos')

  .then(status)

  .then(json)

  .then((data) => { console.log('Request succeeded with JSON response', data) })

  .catch((error) => { console.log('Request failed', error) })
```

Здесь мы пользуемся npm-пакетом [node-fetch](#) и ресурсом [jsonplaceholder.typicode.com](#) в качестве источника JSON-данных.

В данном примере функция `fetch()` применяется для загрузки элемента TODO-списка с использованием цепочки промисов. После выполнения `fetch()` возвращается [ответ](#), имеющий множество свойств, среди которых нас интересуют следующие:

- `status` — числовое значение, представляющее собой код состояния HTTP.
- `statusText` — текстовое описание кода состояния HTTP, которое представлено строкой OK в том случае, если запрос был выполнен успешно.

У объекта `response` есть метод `json()`, который возвращает промис, при разрешении которого выдаётся обработанное содержимое тела запроса, представленное в формате JSON.

Учитывая вышесказанное, опишем то, что происходит в этом коде. Первый промис в цепочке представлен объявленной нами функцией `status()`, которая проверяет состояние ответа, и, если он свидетельствует о том, что запрос не удался (то есть, код состояния HTTP не находится в диапазоне между 200 и 299), отклоняет промис. Эта операция приводит к тому, что другие выражения `.then()` в

цепочке промисов не выполняются и мы сразу попадаем в метод `.catch()`, выводя в консоль, вместе с сообщением об ошибке, текст `Request failed`.

Если код состояния HTTP нас устраивает, вызывается объявленная нами функция `json()`. Так как предыдущий промис, при его успешном разрешении, возвращает объект `response`, мы используем его в качестве входного значения для второго промиса.

В данном случае мы возвращаем обработанные JSON-данные, поэтому третий промис получает именно их, после чего они, предварённые сообщением о том, что в результате запроса удалось получить нужные данные, выводятся в консоль.

Обработка ошибок

В предыдущем примере у нас был метод `.catch()`, присоединённый к цепочке промисов. Если что-то в цепочке промисов идёт не так и возникает ошибка, либо если один из промисов оказывается отклонённым, управление передаётся в ближайшее выражение `.catch()`. Вот как выглядит ситуация, когда в промисе возникает ошибка:

```
new Promise((resolve, reject) => {  
  
    throw new Error('Error')  
  
}))  
  
    .catch((err) => { console.error(err) })
```

Вот пример срабатывания `.catch()` после отклонения промиса:

```
new Promise((resolve, reject) => {  
  
    reject('Error')  
  
}))  
  
    .catch((err) => { console.error(err) })
```

Каскадная обработка ошибок

Что делать, если в выражении `.catch()` возникнет ошибка? Для обработки такой ошибки можно включить в цепочку промисов ещё одно выражение `.catch()` (а потом можно присоединить к цепочке ещё столько выражений `.catch()`, сколько понадобится):

```
new Promise((resolve, reject) => {  
  
    throw new Error('Error')  
  
}))  
  
    .catch((err) => { throw new Error('Error') })  
  
    .catch((err) => { console.error(err) })
```

Теперь рассмотрим несколько полезных методов, используемых для управления промисами.

`Promise.all()`

Если вам нужно выполнить некое действие после разрешения нескольких промисов, сделать это можно с помощью команды `Promise.all()`. Рассмотрим пример:

```
const f1 = fetch('https://jsonplaceholder.typicode.com/todos/1')
```

```
const f2 = fetch('https://jsonplaceholder.typicode.com/todos/2')

Promise.all([f1, f2]).then((res) => {

    console.log('Array of results', res)

})

.catch((err) => {

    console.error(err)

})
```

В ES2015 появился синтаксис деструктурирующего присваивания, с его использованием можно создавать конструкции следующего вида:

```
Promise.all([f1, f2]).then(([res1, res2]) => {

    console.log('Results', res1, res2)

})
```

Тут мы, в качестве примера, рассматривали API Fetch, но `Promise.all()`, конечно, позволяет работать с любыми промисами.

Promise.race()

Команда `Promise.race()` позволяет выполнить заданное действие после того, как будет разрешён один из переданных ей промисов. Соответствующий коллбэк, содержащий результаты этого первого промиса, вызывается лишь один раз. Рассмотрим пример:

```
const first = new Promise((resolve, reject) => {

    setTimeout(resolve, 500, 'first')

})

const second = new Promise((resolve, reject) => {

    setTimeout(resolve, 100, 'second')

})

Promise.race([first, second]).then((result) => {

    console.log(result) // second

})
```

Об ошибке Uncaught TypeError, которая встречается при работе с промисами

Если, работая с промисами, вы столкнётесь с ошибкой `Uncaught TypeError: undefined is not a promise`, проверьте, чтобы при создании промисов использовалась бы конструкция `new Promise()` а не просто `Promise()`.

Конструкция async/await

Конструкция `async/await` представляет собой современный подход к асинхронному программированию, упрощая его. Асинхронные функции можно представить в виде комбинации промисов и генераторов, и, в целом, эта конструкция представляет собой абстракцию над промисами.

Конструкция `async/await` позволяет уменьшить объём шаблонного кода, который приходится писать при работе с промисами. Когда промисы появились в стандарте ES2015, они были направлены на решение проблемы создания асинхронного кода. Они с этой задачей справились, но за два года, разделяющие выход стандартов ES2015 и ES2017, стало понятно, что считать их окончательным решением проблемы нельзя.

Одной из проблем, которую решали промисы, был знаменитый «ад коллбэков», но они, решая эту проблему, создали собственные проблемы схожего характера.

Промисы представляли собой простые конструкции, вокруг которых можно было бы построить нечто, обладающее более простым синтаксисом. В результате, когда пришло время, появилась конструкция `async/await`. Её использование позволяет писать код, который выглядит как синхронный, но при этом является асинхронным, в частности, не блокирует главный поток.

Как работает конструкция `async/await`

Асинхронная функция возвращает промис, как, например, в следующем примере:

```
const doSomethingAsync = () => {  
  return new Promise((resolve) => {  
    setTimeout(() => resolve('I did something'), 3000)  
  })  
}
```

Когда нужно вызвать подобную функцию, перед командой её вызова нужно поместить ключевое слово `await`. Это приведёт к тому, что вызывающий её код будет ждать разрешения или отклонения соответствующего промиса. Нужно отметить, что функция, в которой используется ключевое слово `await`, должна быть объявлена с использованием ключевого слова `async`:

```
const doSomething = async () => {  
  console.log(await doSomethingAsync())  
}
```

Объединим два вышеприведённых фрагмента кода и исследуем его поведение:

```
const doSomethingAsync = () => {  
  return new Promise((resolve) => {  
    setTimeout(() => resolve('I did something'), 3000)  
  })  
}  
  
const doSomething = async () => {  
  console.log(await doSomethingAsync())  
}  
  
console.log('Before')
```

```
doSomething()

console.log('After')
```

Этот код выведет следующее:

```
Before

After

I did something
```

Текст `I did something` попадёт в консоль с задержкой в 3 секунды.

О промисах и асинхронных функциях

Если объявить некую функцию с использованием ключевого слова `async`, это будет означать, что такая функция возвратит промис даже если в явном виде это не делается. Именно поэтому, например, следующий пример представляет собой рабочий код:

```
const aFunction = async () => {

  return 'test'

}

aFunction().then(console.log) // Будет выведен текст 'test'
```

Эта конструкция аналогична такой:

```
const aFunction = async () => {

  return Promise.resolve('test')

}

aFunction().then(console.log) // Будет выведен текст 'test'
```

Сильные стороны `async/await`

Анализируя вышеприведённые примеры, можно видеть, что код, в котором применяется `async/await`, оказывается проще, чем код, в котором используется объединение промисов в цепочки, или код, основанный на функциях обратного вызова. Здесь мы, конечно, рассмотрели очень простые примеры. В полной мере ощутить вышеозначенные преимущества можно, работая с гораздо более сложным кодом. Вот, например, как загрузить и разобрать JSON-данные с использованием промисов:

```
const getFirstUserData = () => {

  return fetch('/users.json') // загрузить список пользователей

    .then(response => response.json()) // разобрать JSON

    .then(users => users[0]) // выбрать первого пользователя

    .then(user => fetch(`/users/${user.name}`)) // загрузить данные о пользователе

    .then(userResponse => response.json()) // разобрать JSON

}
```

```
getFirstUserData()
```

Вот как выглядит решение той же задачи с использованием `async/await`:

```
const getFirstUserData = async () => {  
  const response = await fetch('/users.json') // загрузить список пользователей  
  const users = await response.json() // разобрать JSON  
  const user = users[0] // выбрать первого пользователя  
  const userResponse = await fetch(`/users/${user.name}`) // загрузить данные о  
пользователе  
  const userData = await user.json() // разобрать JSON  
  return userData  
}
```

```
getFirstUserData()
```

Использование последовательностей из асинхронных функций

Асинхронные функции легко можно объединять в конструкции, напоминающие цепочки промисов. Результаты такого объединения, однако, отличаются гораздо лучшей читабельностью:

```
const promiseToDoSomething = () => {  
  return new Promise(resolve => {  
    setTimeout(() => resolve('I did something'), 10000)  
  })  
}  
  
const watchOverSomeoneDoingSomething = async () => {  
  const something = await promiseToDoSomething()  
  return something + ' and I watched'  
}  
  
const watchOverSomeoneWatchingSomeoneDoingSomething = async () => {  
  const something = await watchOverSomeoneDoingSomething()  
  return something + ' and I watched as well'  
}  
  
watchOverSomeoneWatchingSomeoneDoingSomething().then((res) => {  
  console.log(res)  
})
```

Этот код выведет следующий текст:

I did something and I watched and I watched as well

Упрощённая отладка

Промисы сложно отлаживать, так как при их использовании нельзя эффективно пользоваться обычными инструментами отладчика (наподобие «шага с обходом», `step-over`). Код же, написанный с использованием `async/await`, можно отлаживать с использованием тех же методов, что и обычный синхронный код.

Генерирование событий в Node.js

Если вы работали с JavaScript в браузере, то вы знаете, что события играют огромнейшую роль в обработке взаимодействий пользователей со страницами. Речь идёт об обработке событий, вызываемых щелчками и движениями мыши, нажатиями клавиш на клавиатуре и так далее. В Node.js можно работать с событиями, которые программист создаёт самостоятельно. Здесь можно создать собственную систему событий с использованием модуля [events](#). В частности, этот модуль предлагает нам класс `EventEmitter`, возможности которого можно задействовать для организации работы с событиями. Прежде чем воспользоваться этим механизмом, его нужно подключить:

```
const EventEmitter = require('events').EventEmitter
```

При работе с ним нам доступны, кроме прочих, методы `on()` и `emit()`. Метод `emit` используется для вызова событий. Метод `on` используется для настройки коллбэков, обработчиков событий, которые вызываются при вызове определённого события.

Например, давайте создадим событие `start`. Когда оно происходит, будем выводить что-нибудь в консоль:

```
eventEmitter = new EventEmitter();
```

```
eventEmitter.on('start', () => {  
  console.log('started')  
})
```

Для того чтобы вызвать это событие, используется следующая конструкция:

```
eventEmitter.emit('start')
```

В результате выполнения этой команды вызывается обработчик события и строка `started` попадает в консоль.

Обработчику событий можно передавать аргументы, представляя их в виде дополнительных аргументов метода `emit()`:

```
eventEmitter.on('start', (number) => {  
  console.log(`started ${number}`)  
})  
  
eventEmitter.emit('start', 23)
```

Похожим образом поступают и в случаях, когда обработчику надо передать несколько аргументов:

```
eventEmitter.on('start', (start, end) => {
```

```
    console.log(`started from ${start} to ${end}`)  
  })  
  
  EventEmitter.emit('start', 1, 100)
```

Объекты класса `EventEmitter` имеют и некоторые другие полезные методы:

- `once()` — позволяет зарегистрировать обработчик события, который можно вызвать лишь один раз.
- `removeListener()` — позволяет удалить переданный ему обработчик из массива обработчиков переданного ему события.
- `removeAllListeners()` — позволяет удалить все обработчики переданного ему события.

Часть 8: протоколы HTTP и WebSocket

Что происходит при выполнении HTTP-запросов?

Поговорим о том, как браузеры выполняют запросы к серверам с использованием протокола HTTP/1.1.

Если вы когда-нибудь проходили собеседование в IT-сфере, то вас могли спросить о том, что происходит, когда вы вводите нечто в адресную строку браузера и нажимаете Enter. Пожалуй, это один из самых популярных вопросов, который встречается на подобных собеседованиях. Тот, кто задаёт подобные вопросы, хочет узнать, можете ли вы объяснить некоторые довольно-таки простые концепции и выяснить, понимаете ли вы принципы работы интернета.

Этот вопрос затрагивает множество технологий, понимать общие принципы которых — значит понимать, как устроена одна из самых сложных систем из когда-либо построенных человечеством, которая охватывает весь мир.

Протокол HTTP

Современные браузеры способны отличать настоящие URL-адреса, вводимые в их адресную строку, от поисковых запросов, для обработки которых обычно используется заданная по умолчанию поисковая система. Мы будем говорить именно об URL-адресах. Если вы введёте в строку браузера адрес сайта, вроде `flaviocopes.com`, браузер преобразует этот адрес к виду `http://flaviocopes.com`, исходя из предположения о том, что для обмена данными с указанным ресурсом будет использоваться протокол HTTP. Обратите внимание на то, что в Windows то, о чём мы будем тут говорить, может выглядеть немного иначе, чем в macOS и Linux.

Фаза DNS-поиска

Итак, браузер, начиная работу по загрузке данных с запрошенного пользователем адреса, выполняет операцию DNS-поиска (DNS Lookup) для того, чтобы выяснить IP-адрес соответствующего сервера. Символьные имена ресурсов, вводимые в адресную строку, удобны для людей, но устройство интернета подразумевает возможность обмена данными между компьютерами с использованием IP-адресов, которые представляют собой наборы чисел наподобие `222.324.3.1` (для протокола IPv4).

Сначала, выясняя IP-адрес сервера, браузер заглядывает в локальный DNS-кэш для того, чтобы узнать, не выполнялась ли недавно подобная процедура. В браузере Chrome, например, есть удобный способ посмотреть DNS-кэш, введя в адресной строке следующий адрес:

```
chrome://net-internals/#dns.
```

Если в кэше ничего найти не удаётся, браузер использует системный вызов POSIX `gethostbyname` для того, чтобы узнать IP-адрес сервера.

Функция `gethostbyname`

Функция `gethostbyname` сначала проверяет файл `hosts`, который, в macOS или Linux, можно найти по адресу `/etc/hosts`, для того, чтобы узнать, можно ли, выясняя адрес сервера, обойтись локальными сведениями.

Если локальными средствами разрешить запрос на выяснение IP-адреса сервера не удаётся, система выполняет запрос к DNS-серверу. Адреса таких серверов хранятся в настройках системы.

Вот пара популярных DNS-серверов:

- 8.8.8.8: DNS-сервер Google.
- 1.1.1.1: DNS-сервер CloudFlare.

Большинство людей используют DNS-сервера, предоставляемые их провайдерами. Браузер выполняет DNS-запросы с использованием протокола UDP.

TCP и UDP — это два базовых протокола, применяемых в компьютерных сетях. Они расположены на одном концептуальном уровне, но TCP — это протокол, ориентированный на соединениях, а для обмена UDP-сообщениями, обработка которых создаёт небольшую дополнительную нагрузку на системы, процедура установления соединения не требуется. О том, как именно происходит обмен данными по UDP, мы говорить не будем.

IP-адрес, соответствующий интересующему нас доменному имени, может иметься в кэше DNS-сервера. Если это не так — он обратится к корневому DNS-серверу. Система корневых DNS-серверов состоит из 13 серверов, от которых зависит работа всего интернета.

Надо отметить, что корневому DNS-серверу неизвестны соответствия между всеми существующими в мире доменными именами и IP-адресами. Но подобным серверам известны адреса DNS-серверов верхнего уровня для таких доменов, как `.com`, `.it`, `.pizza`, и так далее.

Получив запрос, корневой DNS-сервер перенаправляет его к DNS-серверу домена верхнего уровня, к так называемому TLD-серверу (от Top-Level Domain).

Предположим, браузер ищет IP-адрес для сервера `flaviocopes.com`. Обратившись к корневому DNS-серверу, браузер получит у него адрес TLD-сервера для зоны `.com`. Теперь этот адрес будет сохранён в кэше, в результате, если будет нужно узнать IP-адрес ещё какого-нибудь URL из зоны `.com`, к корневому DNS-серверу не придётся обращаться снова.

У TLD-серверов есть IP-адреса серверов имён (Name Server, NS), средствами которых и можно узнать IP-адрес по имеющемуся у нас URL. Откуда NS-сервера берут эти сведения? Дело в том, что если вы покупаете домен, доменный регистратор отправляет данные о нём серверам имён. Похожая процедура выполняется и, например, при смене хостинга.

Сервера, о которых идёт речь, обычно принадлежат хостинг-провайдерам. Как правило, для защиты от сбоев, создаются по несколько таких серверов. Например, у них могут быть такие адреса:

- `ns1.dreamhost.com`
- `ns2.dreamhost.com`
- `ns3.dreamhost.com`

Для выяснения IP-адреса по URL, в итоге, обращаются к таким серверам. Именно они хранят актуальные данные об IP-адресах.

Теперь, после того, как нам удалось выяснить IP-адрес, стоящий за введённым в адресную строку браузера URL, мы переходим к следующему шагу нашей работы.

Установление TCP-соединения

Узнав IP-адрес сервера, клиент может инициировать процедуру TCP-подключения к нему. В процессе установления TCP-соединения клиент и сервер передают друг другу некоторые служебные данные, после чего они смогут обмениваться информацией. Это означает, что, после установления соединения, клиент сможет отправить серверу запрос.

Отправка запроса

Запрос представляет собой структурированный в соответствии с правилами используемого протокола фрагмент текста. Он состоит из трёх частей:

- Строка запроса.
- Заголовок запроса.
- Тело запроса.

Строка запроса

Строка запроса представляет собой одну текстовую строку, в которой содержатся следующие сведения:

- Метод HTTP.
- Адрес ресурса.
- Версия протокола.

Выглядеть она, например, может так:

```
GET / HTTP/1.1
```

Заголовок запроса

Заголовок запроса представлен набором пар вида `поле: значение`. Существуют 2 обязательных поля заголовка, одно из которых — `Host`, а второе — `Connection`. Остальные поля необязательны.

Заголовок может выглядеть так:

```
Host: flaviocopes.com
```

```
Connection: close
```

Поле `Host` указывает на доменное имя, которое интересует браузер. Поле `Connection`, установленное в значение `close`, означает, что соединение между клиентом и сервером не нужно держать открытым.

Среди других часто используемых заголовков запросов можно отметить следующие:

- `Origin`
- `Accept`
- `Accept-Encoding`
- `Cookie`
- `Cache-Control`
- `Dnt`

На самом деле, их существует гораздо больше.

Заголовок запроса завершается пустой строкой.

Тело запроса

Тело запроса необязательно, в GET-запросах оно не используется. Тело запроса используется в POST-запросах, а также в других запросах. Оно может содержать, например, данные в формате JSON.

Так как сейчас речь идёт о GET-запросе, тело запроса будет пустым, с ним мы работать не будем.

Ответ

После того, как сервер получает отправленный клиентом запрос, он его обрабатывает и отправляем клиенту ответ.

Ответ начинается с кода состояния и с соответствующего сообщения. Если запрос выполнен успешно, то начало ответа будет выглядеть так:

200 OK

Если что-то пошло не так, тут могут быть и другие коды. Например, следующие:

- 404 Not Found
- 403 Forbidden
- 301 Moved Permanently
- 500 Internal Server Error
- 304 Not Modified
- 401 Unauthorized

Далее в ответе содержится список HTTP-заголовков и тело ответа (которое, так как запрос выполняет браузер, будет представлять собой HTML-код).

Разбор HTML-кода

После того, как браузер получает ответ сервера, в теле которого содержится HTML-код, он начинает его разбирать, повторяя вышеописанный процесс для каждого ресурса, который нужен для формирования страницы. К таким ресурсам относятся, например, следующие:

- CSS-файлы.
- Изображения.
- Значок веб-страницы (favicon).
- JavaScript-файлы.

То, как именно браузер выводит страницу, к нашему разговору не относится. Главное, что нас тут интересует, заключается в том, что вышеописанный процесс запроса и получения данных используется не только для HTML-кода, но и для любых других объектов, передаваемых с сервера в браузер с использованием протокола HTTP.

О создании простого сервера средствами Node.js

Теперь, после того, как мы разобрали процесс взаимодействия браузера и сервера, вы можете по-новому взглянуть на раздел **Первое Node.js-приложение** из [первой части](#) этой серии материалов, в котором мы описывали код простого сервера.

Выполнение HTTP-запросов средствами Node.js

Для выполнения HTTP-запросов средствами Node.js используется соответствующий [модуль](#). В приведённых ниже примерах применяется модуль [https](#). Дело в том, что в современных условиях всегда, когда это возможно, нужно применять именно протокол HTTPS.

Выполнение GET-запросов

Вот пример выполнения GET-запроса средствами Node.js:

```
const https = require('https')

const options = {
```

```
    hostname: 'flaviocopes.com',

    port: 443,

    path: '/todos',

    method: 'GET'

  }

  const req = https.request(options, (res) => {

    console.log(`statusCode: ${res.statusCode}`)

    res.on('data', (d) => {

      process.stdout.write(d)

    })

  })

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

    console.error(error)

  })

  req.end()
```

Выполнение POST-запроса

Вот как выполнить POST-запрос из Node.js:

```
const https = require('https')

const data = JSON.stringify({

  todo: 'Buy the milk'

})

const options = {

  hostname: 'flaviocopes.com',

  port: 443,

  path: '/todos',

  method: 'POST',

  headers: {

    'Content-Type': 'application/json',

    'Content-Length': data.length

  }

}
```

```
const req = https.request(options, (res) => {  
  console.log(`statusCode: ${res.statusCode}`)  
  res.on('data', (d) => {  
    process.stdout.write(d)  
  })  
})  
  
req.on('error', (error) => {  
  console.error(error)  
})  
  
req.write(data)  
  
req.end()
```

Выполнение PUT-запросов и DELETE-запросов

Выполнение таких запросов выглядит так же, как и выполнение POST-запросов. Главное отличие, помимо смыслового наполнения таких операций, заключается в значении свойства `method` объекта `options`.

Выполнение HTTP-запросов в Node.js с использованием библиотеки Axios

Axios — это весьма популярная JavaScript-библиотека, работающая и в браузере (сюда входят все современные браузеры и IE, начиная с IE8), и в среде Node.js, которую можно использовать для выполнения HTTP-запросов.

Эта библиотека основана на промисах, она обладает некоторыми преимуществами перед стандартными механизмами, в частности, перед API Fetch. Среди её преимуществ можно отметить следующие:

- Поддержка старых браузеров (для использования Fetch нужен полифилл).
- Возможность прерывания запросов.
- Поддержка установки тайм-аутов для запросов.
- Встроенная защита от CSRF-атак.
- Поддержка выгрузки данных с предоставлением сведений о ходе этого процесса.
- Поддержка преобразования JSON-данных.
- Работа в Node.js

Установка

Для установки Axios можно воспользоваться npm:

```
npm install axios
```

Того же эффекта можно достичь и при работе с yarn:

```
yarn add axios
```

Подключить библиотеку к странице можно с помощью unpkg.com:

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

API Axios

Выполнить HTTP-запрос можно, воспользовавшись объектом `axios`:

```
axios({
  url: 'https://dog.ceo/api/breeds/list/all',
  method: 'get',
  data: {
    foo: 'bar'
  }
})
```

Но обычно удобнее пользоваться специальными методами:

- `axios.get()`
- `axios.post()`

Это похоже на то, как в jQuery, вместо `$.ajax()` пользуются `$.get()` и `$.post()`.

Axios предлагает отдельные методы и для выполнения других видов HTTP-запросов, которые не так популярны, как GET и POST, но всё-таки используются:

- `axios.delete()`
- `axios.put()`
- `axios.patch()`
- `axios.options()`

В библиотеке имеется метод для выполнения запроса, предназначенного для получения лишь HTTP-заголовков, без тела ответа:

- `axios.head()`

Запросы GET

Axios удобно использовать с применением современного синтаксиса `async/await`. В следующем примере кода, рассчитанном на Node.js, библиотека используется для загрузки списка пород собак из [API Dog](#). Здесь применяется метод `axios.get()` и осуществляется подсчёт пород:

```
const axios = require('axios')

const getBreeds = async () => {
  try {
    return await axios.get('https://dog.ceo/api/breeds/list/all')
  } catch (error) {
    console.error(error)
  }
}
```

```
const countBreeds = async () => {  
  const breeds = await getBreeds()  
  if (breeds.data.message) {  
    console.log(`Got ${Object.entries(breeds.data.message).length} breeds`)  
  }  
}  
  
countBreeds()
```

То же самое можно переписать и без использования async/await, применив промисы:

```
const axios = require('axios')  
  
const getBreeds = () => {  
  try {  
    return axios.get('https://dog.ceo/api/breeds/list/all')  
  } catch (error) {  
    console.error(error)  
  }  
}  
  
const countBreeds = async () => {  
  const breeds = getBreeds()  
  .then(response => {  
    if (response.data.message) {  
      console.log(  
        `Got ${Object.entries(response.data.message).length} breeds`  
      )  
    }  
  })  
  .catch(error => {  
    console.log(error)  
  })  
}  
  
countBreeds()
```

Использование параметров в GET-запросах

GET-запрос может содержать параметры, которые в URL выглядят так:

```
https://site.com/?foo=bar
```

При использовании Axios запрос подобного рода можно выполнить так:

```
axios.get('https://site.com/?foo=bar')
```

Того же эффекта можно достичь, настроив свойство `params` в объекте с параметрами:

```
axios.get('https://site.com/', {  
  params: {  
    foo: 'bar'  
  }  
})
```

Запросы POST

Выполнение POST-запросов очень похоже на выполнение GET-запросов, но тут, вместо метода `axios.get()`, используется метод `axios.post()`:

```
axios.post('https://site.com/')
```

В качестве второго аргумента метод `post` принимает объект с параметрами запроса:

```
axios.post('https://site.com/', {  
  foo: 'bar'  
})
```

Использование протокола WebSocket в Node.js

WebSocket представляет собой альтернативу HTTP, его можно применять для организации обмена данными в веб-приложениях. Этот протокол позволяет создавать долгоживущие двунаправленные каналы связи между клиентом и сервером. После установления соединения канал связи остаётся открытым, что даёт в распоряжение приложения очень быстрое соединение, характеризующееся низкими задержками и небольшой дополнительной нагрузкой на систему.

Протокол WebSocket поддерживают все современные браузеры.

Отличия от HTTP

HTTP и WebSocket — это очень разные протоколы, в которых используются различные подходы к обмену данными. HTTP основан на модели «запрос — ответ»: сервер отправляет клиенту некие данные после того, как они будут запрошены. В случае с WebSocket всё устроено иначе. А именно:

- Сервер может отправлять сообщения клиенту по своей инициативе, не дожидаясь поступления запроса от клиента.
- Клиент и сервер могут обмениваться данными одновременно.
- При передаче сообщения используется крайне малый объём служебных данных. Это, в частности, ведёт к низким задержкам при передаче данных.

Протокол WebSocket очень хорошо подходит для организации связи в режиме реального времени по каналам, которые долго остаются открытыми. HTTP, в свою очередь, отлично подходит для организации эпизодических сеансов связи, инициируемых клиентом. В то же время надо отметить, что,

с точки зрения программирования, реализовать обмен данными по протоколу HTTP гораздо проще, чем по протоколу WebSocket.

Защищённая версия протокола WebSocket

Существует небезопасная версия протокола WebSocket (URI-схема `ws://`), которая напоминает, в плане защищённости, протокол `http://`. Использования `ws://` следует избегать, отдавая предпочтение защищённой версии протокола — `wss://`.

Создание WebSocket-соединения

Для создания WebSocket-соединения нужно воспользоваться соответствующим [конструктором](#):

```
const url = 'wss://myserver.com/something'

const connection = new WebSocket(url)
```

После успешного установления соединения вызывается событие `open`. Организовать прослушивание этого события можно, назначив функцию обратного вызова свойству `onopen` объекта `connection`:

```
connection.onopen = () => {

  //...

}
```

Для обработки ошибок используется обработчик события `onerror`:

```
connection.onerror = error => {

  console.log(`WebSocket error: ${error}`)

}
```

Отправка данных на сервер

После открытия WebSocket-соединения с сервером ему можно отправлять данные. Сделать это можно, например в коллбэке `onopen`:

```
connection.onopen = () => {

  connection.send('hey')

}
```

Получение данных с сервера

Для получения с сервера данных, отправленных с использованием протокола WebSocket, можно назначить коллбэк `onmessage`, который будет вызван при получении события `message`:

```
connection.onmessage = e => {

  console.log(e.data)

}
```

Реализация WebSocket-сервера в среде Node.js

Для того чтобы реализовать WebSocket-сервер в среде Node.js, можно воспользоваться популярной библиотекой [ws](#). Мы применим её для разработки сервера, но она подходит и для создания клиентов, и для организации взаимодействия между двумя серверами.

Установим эту библиотеку, предварительно инициализировав проект:

```
yarn init
```

```
yarn add ws
```

Код WebSocket-сервера, который нам надо написать, довольно-таки компактен:

```
const WebSocket = require('ws')

const wss = new WebSocket.Server({ port: 8080 })

wss.on('connection', ws => {

  ws.on('message', message => {

    console.log(`Received message => ${message}`)

  })

  ws.send('ho!')

})
```

Здесь мы создаём новый сервер, который прослушивает стандартный для протокола WebSocket порт 8080 и описываем коллбэк, который, когда будет установлено соединение, отправляет клиенту сообщение `ho!` и выводит в консоль сообщение, полученное от клиента.

[Bot](#) рабочий пример WebSocket-сервера, а [vot](#) — клиент, который может с ним взаимодействовать.

Часть 9: работа с файловой системой

Работа с файловыми дескрипторами в Node.js

Прежде чем вы сможете взаимодействовать с файлами, находящимися в файловой системе вашего сервера, вам необходимо получить дескриптор файла.

Дескриптор можно получить, воспользовавшись для открытия файла асинхронным методом `open()` из модуля `fs`:

```
const fs = require('fs')

fs.open('/Users/flavio/test.txt', 'r', (err, fd) => {

  //fd - это дескриптор файла

})
```

Обратите внимание на второй параметр, `r`, использованный при вызове метода `fs.open()`. Это — флаг, который сообщает системе о том, что файл открывают для чтения. Вот ещё некоторые флаги, которые часто используются при работе с этим и некоторыми другими методами:

- `r+` — открыть файл для чтения и для записи.
- `w+` — открыть файл для чтения и для записи, установив указатель потока в начало файла. Если файл не существует — он создаётся.
- `a` — открыть файл для записи, установив указатель потока в конец файла. Если файл не существует — он создаётся.
- `a+` — открыть файл для чтения и записи, установив указатель потока в конец файла. Если файл не существует — он создаётся.

Файлы можно открывать и пользуясь синхронным методом `fs.openSync()`, который, вместо того, чтобы предоставить дескриптор файла в коллбэке, возвращает его:

```
const fs = require('fs')

try {

  const fd = fs.openSync('/Users/flavio/test.txt', 'r')

} catch (err) {

  console.error(err)

}
```

После получения дескриптора любым из вышеописанных способов вы можете производить с ним необходимые операции.

Данные о файлах

С каждым файлом связан набор данных о нём, исследовать эти данные можно средствами Node.js. В частности, сделать это можно, используя метод `stat()` из модуля `fs`.

Вызывают этот метод, передавая ему путь к файлу, и, после того, как Node.js получит необходимые сведения о файле, он вызовет коллбэк, переданный методу `stat()`. Вот как это выглядит:

```
const fs = require('fs')

fs.stat('/Users/flavio/test.txt', (err, stats) => {

  if (err) {

    console.error(err)

    return

  }

  //сведения о файле содержатся в аргументе `stats`

})
```

В Node.js имеется возможность синхронного получения сведений о файлах. При таком подходе главный поток блокируется до получения свойств файла:

```
const fs = require('fs')

try {

  const stats = fs.statSync ('/Users/flavio/test.txt')

} catch (err) {

  console.error(err)

}
```

Информация о файле попадёт в константу `stats`. Что это за информация? На самом деле, соответствующий объект предоставляет нам большое количество полезных свойств и методов:

- Методы `.isFile()` и `.isDirectory()` позволяют, соответственно, узнать, является ли исследуемый файл обычным файлом или директорией.
- Метод `.isSymbolicLink()` позволяет узнать, является ли файл символической ссылкой.
- Размер файла можно узнать, воспользовавшись свойством `.size`.

Тут имеются и другие методы, но эти — самые употребимые. Вот как ими пользоваться:

```
const fs = require('fs')

fs.stat('/Users/flavio/test.txt', (err, stats) => {

  if (err) {

    console.error(err)

    return

  }

  stats.isFile() //true

  stats.isDirectory() //false

  stats.isSymbolicLink() //false

  stats.size //1024000 // = 1MB

})
```

Пути к файлам в Node.js и модуль path

Путь к файлу — это адрес того места в файловой системе, где он расположен.

В Linux и macOS путь может выглядеть так:

```
/users/flavio/file.txt
```

В Windows пути выглядят немного иначе:

```
C:\users\flavio\file.txt
```

На различия в форматах записи путей при использовании разных операционных систем следует обращать внимание, учитывая операционную систему, используемую для развёртывания Node.js-сервера.

В Node.js есть стандартный модуль `path`, предназначенный для работы с путями к файлам. Перед использованием этого модуля в программе его надо подключить:

```
const path = require('path')
```

Получение информации о пути к файлу

Если у вас есть путь к файлу, то, используя возможности модуля `path`, вы можете, в удобном для восприятия и дальнейшей обработки виде, узнать подробности об этом пути. Выглядит это так:

```
const notes = '/users/flavio/notes.txt'

path.dirname(notes) // /users/flavio

path.basename(notes) // notes.txt
```

```
path.extname(notes) // .txt
```

Здесь, в строке `notes`, хранится путь к файлу. Для разбора пути использованы следующие методы модуля `path`:

- `dirname()` — возвращает родительскую директорию файла.
- `basename()` — возвращает имя файла.
- `extname()` — возвращает расширение файла.

Узнать имя файла без расширения можно, вызвав метод `.basename()` и передав ему второй аргумент, представляющий расширение:

```
path.basename(notes, path.extname(notes)) //notes
```

Работа с путями к файлам

Несколько частей пути можно объединить, используя метод `path.join()`:

```
const name = 'flavio'
```

```
path.join('/', 'users', name, 'notes.txt') //'users/flavio/notes.txt'
```

Найти абсолютный путь к файлу на основе относительного пути к нему можно с использованием метода `path.resolve()`:

```
path.resolve('flavio.txt')
```

```
 //'Users/flavio/flavio.txt' при запуске из моей домашней папки
```

В данном случае Node.js просто добавляет `/flavio.txt` к пути, ведущем к текущей рабочей директории. Если при вызове этого метода передать ещё один параметр, представляющий путь к папке, метод использует его в качестве базы для определения абсолютного пути:

```
path.resolve('tmp', 'flavio.txt')
```

```
 // '/Users/flavio/tmp/flavio.txt' при запуске из моей домашней папки
```

Если путь, переданный в качестве первого параметра, начинается с косой черты — это означает, что он представляет собой абсолютный путь.

```
path.resolve('/etc', 'flavio.txt')
```

```
 // '/etc/flavio.txt'
```

Вот ещё один полезный метод — `path.normalize()`. Он позволяет найти реальный путь к файлу, используя путь, в котором содержатся спецификаторы относительного пути вроде точки (`.`), двух точек (`..`), или двух косых черт:

```
path.normalize('/users/flavio/../../test.txt')
```

```
 // /users/test.txt
```

Методы `resolve()` и `normalize()` не проверяют существование директории. Они просто находят путь, основываясь на переданном им данным.

Чтение файлов в Node.js

Самый простой способ чтения файлов в Node.js заключается в использовании метода `fs.readFile()` с передачей ему пути к файлу и коллбэка, который будет вызван с передачей ему данных файла (или объекта ошибки):

```
fs.readFile('/Users/flavio/test.txt', (err, data) => {

  if (err) {

    console.error(err)

    return

  }

  console.log(data)

})
```

Если надо, можно воспользоваться синхронной версией этого метода — `fs.readFileSync()`:

```
const fs = require('fs')

try {

  const data = fs.readFileSync('/Users/flavio/test.txt')

  console.log(data)

} catch (err) {

  console.error(err)

}
```

По умолчанию при чтении файлов используется кодировка `utf8`, но кодировку можно задать и самостоятельно, передав методу соответствующий параметр.

Методы `fs.readFile()` и `fs.readFileSync()` считывают в память всё содержимое файла. Это означает, что работа с большими файлами с применением этих методов серьёзно отразится на потреблении памяти вашим приложением и окажет влияние на его производительность. Если с такими файлами нужно работать, лучше всего воспользоваться потоками.

Запись файлов в Node.js

В Node.js легче всего записывать файлы с использованием метода `fs.writeFile()`:

```
const fs = require('fs')

const content = 'Some content!'

fs.writeFile('/Users/flavio/test.txt', content, (err) => {

  if (err) {

    console.error(err)

    return

  }

})
```

```

    }

    //файл записан успешно
  })

```

Есть и синхронная версия того же метода — `fs.writeFileSync()`:

```

const fs = require('fs')

const content = 'Some content!'

try {

  const data = fs.writeFileSync('/Users/flavio/test.txt', content)

  //файл записан успешно

} catch (err) {

  console.error(err)

}

```

Эти методы, по умолчанию, заменяют содержимое существующих файлов. Изменить их стандартное поведение можно, воспользовавшись соответствующим флагом:

```

fs.writeFile('/Users/flavio/test.txt', content, { flag: 'a+' }, (err) => {})

```

Тут могут использоваться флаги, которые мы уже перечисляли в разделе, посвящённом дескрипторам. Подробности о флагах можно узнать [здесь](#).

Присоединение данных к файлу

Метод `fs.appendFile()` (и его синхронную версию — `fs.appendFileSync()`) удобно использовать для присоединения данных к концу файла:

```

const content = 'Some content!'

fs.appendFile('file.log', content, (err) => {

  if (err) {

    console.error(err)

    return

  }

  //готово!

})

```

Об использовании потоков

Выше мы описывали методы, которые, выполняя запись в файл, пишут в него весь объём переданных им данных, после чего, если используются их синхронные версии, возвращают управление программе, а если применяются асинхронные версии — вызывают коллбэки. Если вас такое состояние дел не устраивает — лучше будет воспользоваться потоками.

Работа с директориями в Node.js

Модуль `fs` предоставляет в распоряжение разработчика много удобных методов, которые можно использовать для работы с директориями.

Проверка существования папки

Для того чтобы проверить, существует ли директория и может ли Node.js получить к ней доступ, учитывая разрешения, можно использовать метод `fs.access()`.

Создание новой папки

Для того чтобы создавать новые папки, можно воспользоваться методами `fs.mkdir()` и `fs.mkdirSync()`:

```
const fs = require('fs')

const folderName = '/Users/flavio/test'

try {

  if (!fs.existsSync(dir)){

    fs.mkdirSync(dir)

  }

} catch (err) {

  console.error(err)

}
```

Чтение содержимого папки

Для того чтобы прочесть содержимое папки, можно воспользоваться методами `fs.readdir()` и `fs.readdirSync()`. В этом примере осуществляется чтение содержимого папки — то есть — сведений о том, какие файлы и поддиректории в ней имеются, и возврат их относительных путей:

```
const fs = require('fs')

const path = require('path')

const folderPath = '/Users/flavio'

fs.readdirSync(folderPath)
```

Вот так можно получить полный путь к файлу:

```
fs.readdirSync(folderPath).map(fileName => {

  return path.join(folderPath, fileName)

})
```

Результаты можно отфильтровать для того, чтобы получить только файлы и исключить из вывода директории:

```
const isFile = fileName => {

  return fs.lstatSync(fileName).isFile()

}
```



```
}

fs.readdirSync(folderPath).map(fileName => {

  return path.join(folderPath, fileName).filter(isFile)

})
```

Переименование папки

Для переименования папки можно воспользоваться методами `fs.rename()` и `fs.renameSync()`. Первый параметр — это текущий путь к папке, второй — новый:

```
const fs = require('fs')

fs.rename('/Users/flavio', '/Users/roger', (err) => {

  if (err) {

    console.error(err)

    return

  }

  //готово

})
```

Переименовать папку можно и с помощью синхронного метода `fs.renameSync()`:

```
const fs = require('fs')

try {

  fs.renameSync('/Users/flavio', '/Users/roger')

} catch (err) {

  console.error(err)

}
```

Удаление папки

Для того чтобы удалить папку, можно воспользоваться методами `fs.rmdir()` или `fs.rmdirSync()`. Надо отметить, что удаление папки, в которой что-то есть, задача несколько более сложная, чем удаление пустой папки. Если вам нужно удалять такие папки, воспользуйтесь пакетом [fs-extra](#), который весьма популярен и хорошо поддерживается. Он представляет собой замену модуля `fs`, расширяющую его возможности.

Метод `remove()` из пакета `fs-extra` умеет удалять папки, в которых уже что-то есть.

Установить этот модуль можно так:

```
npm install fs-extra
```

Вот пример его использования:

```
const fs = require('fs-extra')
```

```
const folder = '/Users/flavio'

fs.remove(folder, err => {

  console.error(err)

})
```

Его методами можно пользоваться в виде промисов:

```
fs.remove(folder).then(() => {

  //готово

}).catch(err => {

  console.error(err)

})
```

Допустимо и применение конструкции `async/await`:

```
async function removeFolder(folder) {

  try {

    await fs.remove(folder)

    //готово

  } catch (err) {

    console.error(err)

  }

}
```

```
const folder = '/Users/flavio'

removeFolder(folder)
```

Модуль fs

Выше мы уже сталкивались с некоторыми методами модуля `fs`, применяемыми при работе с файловой системой. На самом деле, он содержит ещё много полезного. Напомним, что он не нуждается в установке, для того, чтобы воспользоваться им в программе, его достаточно подключить:

```
const fs = require('fs')
```

После этого у вас будет доступ к его методам, среди которых отметим следующие, некоторые из которых вам уже знакомы:

- `fs.access()`: проверяет существование файла и возможность доступа к нему с учётом разрешений.
- `fs.appendFile()`: присоединяет данные к файлу. Если файл не существует — он будет создан.
- `fs.chmod()`: изменяет разрешения для заданного файла. Похожие методы: `fs.lchmod()`, `fs.fchmod()`.

- `fs.chown()`: изменяет владельца и группу для заданного файла. Похожие методы: `fs.fchown()`, `fs.lchown()`.
- `fs.close()`: закрывает дескриптор файла.
- `fs.copyFile()`: копирует файл.
- `fs.createReadStream()`: создаёт поток чтения файла.
- `fs.createWriteStream()`: создаёт поток записи файла.
- `fs.link()`: создаёт новую жёсткую ссылку на файл.
- `fs.mkdir()`: создаёт новую директорию.
- `fs.mkdtemp()`: создаёт временную директорию.
- `fs.open()`: открывает файл.
- `fs.readdir()`: читает содержимое директории.
- `fs.readFile()`: считывает содержимое файла. Похожий метод: `fs.read()`.
- `fs.readlink()`: считывает значение символической ссылки.
- `fs.realpath()`: разрешает относительный путь к файлу, построенный с использованием символов `.` и `..`, в полный путь.
- `fs.rename()`: переименовывает файл или папку.
- `fs.rmdir()`: удаляет папку.
- `fs.stat()`: возвращает сведения о файле. Похожие методы: `fs.fstat()`, `fs.lstat()`.
- `fs.symlink()`: создаёт новую символическую ссылку на файл.
- `fs.truncate()`: обрезает файл до заданной длины. Похожий метод: `fs.ftruncate()`.
- `fs.unlink()`: удаляет файл или символическую ссылку.
- `fs.unwatchFile()`: отключает наблюдение за изменениями файла.
- `fs.utimes()`: изменяет временную отметку файла. Похожий метод: `fs.futimes()`.
- `fs.watchFile()`: включает наблюдение за изменениями файла. Похожий метод: `fs.watch()`.
- `fs.writeFile()`: записывает данные в файл. Похожий метод: `fs.write()`.

Интересной особенностью модуля `fs` является тот факт, что все его методы, по умолчанию, являются асинхронными, но существуют и их синхронные версии, имена которых получаются путём добавления слова `Sync` к именам асинхронных методов.

Например:

- `fs.rename()`
- `fs.renameSync()`
- `fs.write()`
- `fs.writeSync()`

Использование синхронных методов seriously влияет на то, как работает программа.

В Node.js 10 имеется экспериментальная поддержка этих [API](#), основанных на промисах.

Исследуем метод `fs.rename()`. Вот асинхронная версия этого метода, использующая коллбэки:

```
const fs = require('fs')

fs.rename('before.json', 'after.json', (err) => {

  if (err) {

    return console.error(err)

  }

  // Готово
```

```
}}
```

При использовании его синхронной версии для обработки ошибок используется конструкция try/catch:

```
const fs = require('fs')

try {

  fs.renameSync('before.json', 'after.json')

  //готово

} catch (err) {

  console.error(err)

}
```

Основное различие между этими вариантами использования данного метода заключается в том, что во втором случае выполнение скрипта будет заблокировано до завершения файловой операции.

Модуль path

Модуль path, о некоторых возможностях которого мы тоже уже говорили, содержит множество полезных инструментов, позволяющих взаимодействовать с файловой системой. Как уже было сказано, устанавливать его не нужно, так как он является частью Node.js. Для того чтобы пользоваться им, его достаточно подключить:

```
const path = require('path')
```

Свойство path.sep этого модуля предоставляет символ, использующийся для разделения сегментов пути (\ в Windows и / в Linux и macOS), а свойство path.delimiter даёт символ, используемый для отделения друг от друга нескольких путей (; в Windows и : в Linux и macOS).

Рассмотрим и проиллюстрируем примерами некоторые методы модуля path.

path.basename()

Возвращает последний фрагмент пути. Передав второй параметр этому методу можно убрать расширение файла.

```
require('path').basename('/test/something') //something

require('path').basename('/test/something.txt') //something.txt

require('path').basename('/test/something.txt', '.txt') //something
```

path.dirname()

Возвращает ту часть пути, которая представляет имя директории:

```
require('path').dirname('/test/something') // /test

require('path').dirname('/test/something/file.txt') // /test/something
```

path.extname()

Возвращает ту часть пути, которая представляет расширение файла:

```
require('path').dirname('/test/something') // ''
```

```
require('path').dirname('/test/something/file.txt') // '.txt'
```

path.isAbsolute()

Возвращает истинное значение если путь является абсолютным:

```
require('path').isAbsolute('/test/something') // true  
require('path').isAbsolute('./test/something') // false
```

path.join()

Соединяет несколько частей пути:

```
const name = 'flavio'  
  
require('path').join('/', 'users', name, 'notes.txt')  
// '/users/flavio/notes.txt'
```

path.normalize()

Пытается выяснить реальный путь на основе пути, который содержит символы, используемые при построении относительных путей вроде `..`, `...` и `///`:

```
require('path').normalize('/users/flavio/../../test.txt') ///users/test.txt
```

path.parse()

Преобразует путь в объект, свойства которого представляют отдельные части пути:

- `root`: корневая директория.
- `dir`: путь к файлу, начиная от корневой директории
- `base`: имя файла и расширение.
- `name`: имя файла.
- `ext`: расширение файла.

Вот пример использования этого метода:

```
require('path').parse('/users/test.txt')
```

В результате его работы получается такой объект:

```
{  
  root: '/',  
  dir: '/users',  
  base: 'test.txt',  
  ext: '.txt',  
  name: 'test'  
}
```

path.relative()

Принимает, в качестве аргументов, 2 пути. Возвращает относительный путь из первого пути ко второму, основываясь на текущей рабочей директории:

```
require('path').relative('/Users/flavio', '/Users/flavio/test.txt')  
// 'test.txt'
```

```
require('path').relative('/Users/flavio', '/Users/flavio/something/test.txt')  
// 'something/test.txt'
```

`path.resolve()`

Находит абсолютный путь на основе переданного ему относительного пути:

```
path.resolve('flavio.txt')
```

// '/Users/flavio/flavio.txt' при запуске из моей домашней папки.

Часть 10: стандартные модули, потоки, базы данных, NODE_ENV

Модуль `Node.js os`

Модуль `os` даёт доступ ко многим функциям, которые можно использовать для получения информации об операционной системе и об аппаратном обеспечении компьютера, на котором работает Node.js. Это стандартный модуль, устанавливать его не надо, для работы с ним из кода его достаточно подключить:

```
const os = require('os')
```

Здесь имеются несколько полезных свойств, которые, в частности, могут пригодиться при работе с файлами.

Так, свойство `os.EOL` позволяет узнать используемый в системе разделитель строк (признак конца строки). В Linux и macOS это `\n`, в Windows — `\r\n`.

Надо отметить, что упоминая тут «Linux и macOS», мы говорим о POSIX-совместимых платформах. Ради краткости изложения менее популярные платформы мы тут не упоминаем.

Свойство `os.constants.signals` даёт сведения о константах, используемых для обработки сигналов процессов наподобие `SIGHUP`, `SIGKILL`, и так далее. [Здесь](#) можно найти подробности о них.

Свойство `os.constants.errno` содержит константы, используемые для сообщений об ошибках — наподобие `EADDRINUSE`, `E_OVERFLOW`.

Теперь рассмотрим основные методы модуля `os`.

`os.arch()`

Этот метод возвращает строку, идентифицирующую архитектуру системы, например — `arm`, `x64`, `arm64`.

`os.cpus()`

Возвращает информацию о процессорах, доступных в системе. Например, эти сведения могут выглядеть так:

```
[ { model: 'Intel(R) Core(TM)2 Duo CPU      P8600  @ 2.40GHz',  
  speed: 2400,  
  times:  
    { user: 281685380,  
      nice: 0,  
      sys: 187986530,
```

```
    idle: 685833750,

    irq: 0 } },

{ model: 'Intel(R) Core(TM)2 Duo CPU      P8600   @ 2.40GHz',

  speed: 2400,

  times:

    { user: 282348700,

      nice: 0,

      sys: 161800480,

      idle: 703509470,

      irq: 0 } } ]
```

os.endianness()

Возвращает **BE** или **LE** в зависимости от того, какой [порядок байтов](#) (Big Engian или Little Endian) был использован для компиляции бинарного файла Node.js.

os.freemem()

Возвращает количество свободной системной памяти в байтах.

os.homedir()

Возвращает путь к домашней директории текущего пользователя. Например — '/Users/flavio'.

os.hostname()

Возвращает имя хоста.

os.loadavg()

Возвращает, в виде массива, данные о средних значениях нагрузки, вычисленные операционной системой. Эта информация имеет смысл только в Linux и macOS. Выглядеть она может так:

```
[ 3.68798828125, 4.00244140625, 11.1181640625 ]
```

os.networkInterfaces()

Возвращает сведения о сетевых интерфейсах, доступных в системе. Например:

```
{ lo0:

  [ { address: '127.0.0.1',

      netmask: '255.0.0.0',

      family: 'IPv4',

      mac: 'fe:82:00:00:00:00',

      internal: true },

    { address: '::1',

      netmask: 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff',

      family: 'IPv6',
```

```

    mac: 'fe:82:00:00:00:00',
    scopeid: 0,
    internal: true },
{ address: 'fe80::1',
  netmask: 'ffff:ffff:ffff:ffff::',
  family: 'IPv6',
  mac: 'fe:82:00:00:00:00',
  scopeid: 1,
  internal: true } ],
en1:
[ { address: 'fe82::9b:8282:d7e6:496e',
  netmask: 'ffff:ffff:ffff:ffff::',
  family: 'IPv6',
  mac: '06:00:00:02:0e:00',
  scopeid: 5,
  internal: false },
{ address: '192.168.1.38',
  netmask: '255.255.255.0',
  family: 'IPv4',
  mac: '06:00:00:02:0e:00',
  internal: false } ],
utun0:
[ { address: 'fe80::2513:72bc:f405:61d0',
  netmask: 'ffff:ffff:ffff:ffff::',
  family: 'IPv6',
  mac: 'fe:80:00:20:00:00',
  scopeid: 8,
  internal: false } ] }

```

os.platform()

Возвращает сведения о платформе, для которой был скомпилирован Node.js. Вот некоторые из возможных возвращаемых значений:

- darwin

- `freebsd`
- `linux`
- `openbsd`
- `win32`

`os.release()`

Возвращает строку, идентифицирующую номер релиза операционной системы.

`os.tmpdir()`

Возвращает путь к заданной в системе директории для хранения временных файлов.

`os.totalmem()`

Возвращает общее количество системной памяти в байтах.

`os.type()`

Возвращает сведения, позволяющие идентифицировать операционную систему. Например:

- `Linux` — `Linux`.
- `Darwin` — `macOS`.
- `Windows_NT` — `Windows`.

`os.uptime()`

Возвращает время работы системы в секундах с последней перезагрузки.

Модуль `Node.js events`

Модуль `events` предоставляет нам класс `EventEmitter`, который предназначен для работы с событиями на платформе `Node.js`. Мы уже немного говорили об этом модуле в [седьмой](#) части этой серии материалов. [Вот](#) документация к нему. Здесь рассмотрим API этого модуля. Напомним, что для использования его в коде нужно, как это обычно бывает со стандартными модулями, его подключить. После этого надо создать новый объект `EventEmitter`. Выглядит это так:

```
const EventEmitter = require('events')

const door = new EventEmitter()
```

Объект класса `EventEmitter` пользуется стандартными механизмами, в частности — следующими событиями:

- `newListener` — это событие вызывается при добавлении обработчика событий.
- `removeListener` — вызывается при удалении обработчика.

Рассмотрим наиболее полезные методы объектов класса `EventEmitter` (подобный объект в названиях методов обозначен как `emitter`).

`emitter.addListener()`

Псевдоним для метода `emitter.on()`.

`emitter.emit()`

Генерирует событие. Синхронно вызывает все обработчики события в том порядке, в котором они были зарегистрированы.

`emitter.eventNames()`

Возвращает массив, который содержит зарегистрированные события.

emitter.getMaxListeners()

Возвращает максимальное число обработчиков, которые можно добавить к объекту класса `EventEmitter`. По умолчанию это 10. При необходимости этот параметр можно увеличить или уменьшить с использованием метода `setMaxListeners()`.

emitter.listenerCount()

Возвращает количество обработчиков события, имя которого передаётся данному методу в качестве параметра:

```
door.listenerCount('open')
```

emitter.listeners()

Возвращает массив обработчиков события для соответствующего события, имя которого передано этому методу:

```
door.listeners('open')
```

emitter.off()

Псевдоним для метода `emitter.removeListener()`, появившийся в Node 10.

emitter.on()

Регистрирует коллбэк, который вызывается при генерировании события. Вот как им пользоваться:

```
door.on('open', () => {  
    console.log('Door was opened')  
})
```

emitter.once()

Регистрирует коллбэк, который вызывается только один раз — при первом возникновении события, для обработки которого зарегистрирован этот коллбэк. Например:

```
const EventEmitter = require('events')  
  
const ee = new EventEmitter()  
  
ee.once('my-event', () => {  
    //вызвать этот коллбэк один раз при первом возникновении события  
})
```

emitter.prependListener()

При регистрации обработчика с использованием методов `on()` или `addListener()` этот обработчик добавляется в конец очереди обработчиков и вызывается для обработки соответствующего события последним. При использовании метода `prependListener()` обработчик добавляется в начало очереди, что приводит к тому, что он будет вызываться для обработки события первым.

emitter.prependOnceListener()

Этот метод похож на предыдущий. А именно, когда обработчик, предназначенный для однократного вызова, регистрируется с помощью метода `once()`, он оказывается последним в очереди обработчиков и последним вызывается. Метод `prependOnceListener()` позволяет добавить такой обработчик в начало очереди.

emitter.removeAllListeners()

Данный метод удаляет все обработчики для заданного события, зарегистрированные в соответствующем объекте. Пользуются им так:

```
door.removeAllListeners('open')
```

emitter.removeListener()

Удаляет заданный обработчик, который нужно передать данному методу. Для того чтобы сохранить обработчик для последующего удаления соответствующий коллбэк можно назначить переменной. Выглядит это так:

```
const doSomething = () => {}
```

```
door.on('open', doSomething)
```

```
door.removeListener('open', doSomething)
```

emitter.setMaxListeners()

Этот метод позволяет задать максимальное количество обработчиков, которые можно добавить к отдельному событию в экземпляре класса `EventEmitter`. По умолчанию, как уже было сказано, можно добавить до 10 обработчиков для конкретного события. Это значение можно изменить. Пользуются данным методом так:

```
door.setMaxListeners(50)
```

Модуль Node.js http

В [восьмой](#) части этой серии материалов мы уже говорили о стандартном модуле Node.js `http`. Он даёт в распоряжение разработчика механизмы, предназначенные для создания HTTP-серверов. Он является основным модулем, применяемым для решения задач обмена данными по сети в Node.js. Подключить его в коде можно так:

```
const http = require('http')
```

В его состав входят свойства, методы и классы. Поговорим о них.

Свойства

http.METHODS

В этом свойстве перечисляются все поддерживаемые методы HTTP:

```
> require('http').METHODS
```

```
[ 'ACL',  
  
  'BIND',  
  
  'CHECKOUT',  
  
  'CONNECT',  
  
  'COPY',  
  
  'DELETE',  
  
  'GET',  
  
  'HEAD',
```

```
'LINK',  
'LOCK',  
'M-SEARCH',  
'MERGE',  
'MKACTIVITY',  
'MKCALENDAR',  
'MKCOL',  
'MOVE',  
'NOTIFY',  
'OPTIONS',  
'PATCH',  
'POST',  
'PROPFIND',  
'PROPPATCH',  
'PURGE',  
'PUT',  
'REBIND',  
'REPORT',  
'SEARCH',  
'SUBSCRIBE',  
'TRACE',  
'UNBIND',  
'UNLINK',  
'UNLOCK',  
'UNSUBSCRIBE' ]
```

http.STATUS_CODES

Здесь содержатся коды состояния HTTP и их описания:

```
> require('http').STATUS_CODES  
  
{ '100': 'Continue',  
  '101': 'Switching Protocols',  
  '102': 'Processing',
```

'200': 'OK',
'201': 'Created',
'202': 'Accepted',
'203': 'Non-Authoritative Information',
'204': 'No Content',
'205': 'Reset Content',
'206': 'Partial Content',
'207': 'Multi-Status',
'208': 'Already Reported',
'226': 'IM Used',
'300': 'Multiple Choices',
'301': 'Moved Permanently',
'302': 'Found',
'303': 'See Other',
'304': 'Not Modified',
'305': 'Use Proxy',
'307': 'Temporary Redirect',
'308': 'Permanent Redirect',
'400': 'Bad Request',
'401': 'Unauthorized',
'402': 'Payment Required',
'403': 'Forbidden',
'404': 'Not Found',
'405': 'Method Not Allowed',
'406': 'Not Acceptable',
'407': 'Proxy Authentication Required',
'408': 'Request Timeout',
'409': 'Conflict',
'410': 'Gone',
'411': 'Length Required',

'412': 'Precondition Failed',
'413': 'Payload Too Large',
'414': 'URI Too Long',
'415': 'Unsupported Media Type',
'416': 'Range Not Satisfiable',
'417': 'Expectation Failed',
'418': 'I\'m a teapot',
'421': 'Misdirected Request',
'422': 'Unprocessable Entity',
'423': 'Locked',
'424': 'Failed Dependency',
'425': 'Unordered Collection',
'426': 'Upgrade Required',
'428': 'Precondition Required',
'429': 'Too Many Requests',
'431': 'Request Header Fields Too Large',
'451': 'Unavailable For Legal Reasons',
'500': 'Internal Server Error',
'501': 'Not Implemented',
'502': 'Bad Gateway',
'503': 'Service Unavailable',
'504': 'Gateway Timeout',
'505': 'HTTP Version Not Supported',
'506': 'Variant Also Negotiates',
'507': 'Insufficient Storage',
'508': 'Loop Detected',
'509': 'Bandwidth Limit Exceeded',
'510': 'Not Extended',
'511': 'Network Authentication Required' }

`http.globalAgent`

Данное свойство указывает на глобальный экземпляр класса `http.Agent`. Он используется для управления соединениями. Его можно считать ключевым компонентом HTTP-подсистемы Node.js. Подробнее о классе `http.Agent` мы поговорим ниже.

Методы

`http.createServer()`

Возвращает новый экземпляр класса `http.Server`. Вот как пользоваться этим методом для создания HTTP-сервера:

```
const server = http.createServer((req, res) => {  
  
  //в этом коллбэке будут обрабатываться запросы  
  
})
```

`http.request()`

Позволяет выполнить HTTP-запрос к серверу, создавая экземпляр класса `http.ClientRequest`.

`http.get()`

Этот метод похож на `http.request()`, но он автоматически устанавливает метод HTTP в значение GET и автоматически же вызывает команду вида `req.end()`.

Классы

Модуль HTTP предоставляет 5 классов — `Agent`, `ClientRequest`, `Server`, `ServerResponse` и `IncomingMessage`. Рассмотрим их.

`http.Agent`

Глобальный экземпляр класса `http.Agent`, создаваемый Node.js, используется для управления соединениями. Он применяется в качестве значения по умолчанию всеми HTTP-запросами и обеспечивает постановку запросов в очередь и повторное использование сокетов. Кроме того, он поддерживает пул сокетов, что позволяет обеспечить высокую производительность сетевой подсистемы Node.js. При необходимости можно создать собственный объект `http.Agent`.

`http.ClientRequest`

Объект класса `http.ClientRequest`, представляющий собой выполняющийся запрос, создаётся при вызове методов `http.request()` или `http.get()`. При получении ответа на запрос вызывается событие `response`, в котором передаётся ответ — экземпляр `http.IncomingMessage`. Данные, полученные после выполнения запроса, можно обработать двумя способами:

- Можно вызвать метод `response.read()`.
- В обработчике события `response` можно настроить прослушиватель для события `data`, что позволяет работать с потоковыми данными.

`http.Server`

Экземпляры этого класса используются для создания серверов с применением команды `http.createServer()`. После того, как у нас имеется объект сервера, мы можем воспользоваться его методами:

- Метод `listen()` используется для запуска сервера и организации ожидания и обработки входящих запросов.
- Метод `close()` останавливает сервер.

http.ServerResponse

Этот объект создаётся классом `http.Server` и передаётся в качестве второго параметра событию `request` при его возникновении. Обычно подобным объектам в коде назначают имя `res`:

```
const server = http.createServer((req, res) => {  
  
  //res - это объект http.ServerResponse  
  
})
```

В таких обработчиках, после того, как ответ сервера будет готов к отправке клиенту, вызывают метод `end()`, завершающий формирование ответа. Этот метод необходимо вызывать после завершения формирования каждого ответа.

Вот методы, которые используются для работы с HTTP-заголовками:

- `getHeaderNames()` — возвращает список имён установленных заголовков.
- `getHeaders()` — возвращает копию установленных HTTP-заголовков.
- `setHeader('headername', value)` — устанавливает значение для заданного заголовка.
- `getHeader('headername')` — возвращает установленный заголовок.
- `removeHeader('headername')` — удаляет установленный заголовок.
- `hasHeader('headername')` — возвращает `true` если в ответе уже есть заголовок, имя которого передано этому методу.
- `headersSent()` — возвращает `true` если заголовки уже отправлены клиенту.

После обработки заголовков их можно отправить клиенту, вызвав метод `response.writeHead()`, который, в качестве первого параметра, принимает код состояния. В качестве второго и третьего параметров ему можно передать сообщение, соответствующее коду состояния, и заголовки.

Для отправки данных клиенту в теле ответа используют метод `write()`. Он отправляет буферизованные данные в поток HTTP-ответа.

Если до этого заголовки ещё не были установлены командой `response.writeHead()`, сначала будут отправлены заголовки с кодом состояния и сообщением, которые заданы в запросе. Задавать их значения можно, устанавливая значения для свойств `statusCode` и `statusMessage`:

```
response.statusCode = 500
```

```
response.statusMessage = 'Internal Server Error'
```

http.IncomingMessage

Объект класса `http.IncomingMessage` создаётся в ходе работы следующих механизмов:

- `http.Server` — при обработке события `request`.
- `http.ClientRequest` — при обработке события `response`.

Его можно использовать для работы с данными ответа. А именно:

- Для того чтобы узнать код состояния ответа и соответствующее сообщение используются свойства `statusCode` и `statusMessage`.
- Заголовки ответа можно посмотреть, обратившись к свойству `headers` или `rawHeaders` (для получения списка необработанных заголовков).
- Метод запроса можно узнать, воспользовавшись свойством `method`.
- Узнать используемую версию HTTP можно с помощью свойства `httpVersion`.
- Для получения URL предназначено свойство `url`.
- Свойство `socket` позволяет получить объект `net.Socket`, связанный с соединением.

Данные ответа представлены в виде потока так как объект `http.IncomingMessage` реализует интерфейс `Readable Stream`.

Работа с потоками в Node.js

Потоки — это одна из фундаментальных концепций, используемых в Node.js-приложениях. Потоки — это инструменты, которые позволяют выполнять чтение и запись файлов, организовывать сетевое взаимодействие систем, и, в целом — эффективно реализовывать операции обмена данными.

Концепция потоков не уникальна для Node.js. Они появились в ОС семейства Unix десятки лет назад. В частности, программы могут взаимодействовать друг с другом, передавая потоки данных с использованием конвейеров (с применением символа конвейера — `|`).

Если представить себе, скажем, чтение файла без использования потоков, то, в ходе выполнения соответствующей команды, содержимое файла будет целиком считано в память, после чего с этим содержимым можно будет работать.

Благодаря использованию механизма потоков файлы можно считывать и обрабатывать по частям, что избавляет от необходимости хранить в памяти большие объёмы данных.

Модуль Node.js [stream](#) представляет собой основу, на которой построены все API, поддерживающие работу с потоками.

О сильных сторонах использования потоков

Потоки, в сравнении с другими способами обработки данных, отличаются следующими преимуществами:

- Эффективное использование памяти. Работа с потоком не предполагает хранения в памяти больших объёмов данных, загружаемых туда заранее, до того, как появится возможность их обработать.
- Экономия времени. Данные, получаемые из потока, можно начать обрабатывать гораздо быстрее, чем в случае, когда для того, чтобы приступить к их обработке, приходится ждать их полной загрузки.

Пример работы с потоками

Традиционный пример работы с потоками демонстрирует чтение файла с диска.

Сначала рассмотрим код, в котором потоки не используются. Стандартный модуль Node.js `fs` позволяет прочитать файл, после чего его можно передать по протоколу HTTP в ответ на запрос, полученный HTTP-сервером:

```
const http = require('http')

const fs = require('fs')

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

  fs.readFile(__dirname + '/data.txt', (err, data) => {

    res.end(data)

  })

})

server.listen(3000)
```

Метод `readFile()`, использованный здесь, позволяет прочесть файл целиком. Когда чтение будет завершено, он вызывает соответствующий коллбэк.

Метод `res.end(data)`, вызываемый в коллбэке, отправляет содержимое файла клиенту.

Если размер файла велик, то эта операция займёт немало времени. Вот тот же пример переписанный с использованием потоков:

```
const http = require('http')

const fs = require('fs')

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

  const stream = fs.createReadStream(__dirname + '/data.txt')

  stream.pipe(res)

})

server.listen(3000)
```

Вместо того, чтобы ждать того момента, когда файл будет полностью прочитан, мы начинаем передавать его данные клиенту сразу после того, как первая порция этих данных будет готова к отправке.

Метод `pipe()`

В предыдущем примере мы использовали конструкцию вида `stream.pipe(res)`, в которой вызывается метод файлового потока `pipe()`. Этот метод берёт данные из их источника и отправляет их в место назначения.

Его вызывают для потока, представляющего собой источник данных. В данном случае это — файловый поток, который отправляют в HTTP-ответ.

Возвращаемым значением метода `pipe()` является целевой поток. Это очень удобно, так как позволяет объединять в цепочки несколько вызовов метода `pipe()`:

```
src.pipe(dest1).pipe(dest2)
```

Это равносильно такой конструкции:

```
src.pipe(dest1)

dest1.pipe(dest2)
```

API Node.js, в которых используются потоки

Потоки — полезный механизм, в результате многие модули ядра Node.js предоставляют стандартные возможности по работе с потоками. Перечислим некоторые из них:

- `process.stdin` — возвращает поток, подключённый к `stdin`.
- `process.stdout` — возвращает поток, подключённый к `stdout`.
- `process.stderr` — возвращает поток, подключённый к `stderr`.
- `fs.createReadStream()` — создаёт читаемый поток для работы с файлом.
- `fs.createWriteStream()` — создаёт записываемый поток для работы с файлом.
- `net.connect()` — иницирует соединение, основанное на потоке.
- `http.request()` — возвращает экземпляр класса `http.ClientRequest`, предоставляющий доступ к записываемому потоку.

- `zlib.createGzip()` — сжимает данные с использованием алгоритма `gzip` и отправляет их в поток.
- `zlib.createGunzip()` — выполняет декомпрессию `gzip`-потока.
- `zlib.createDeflate()` — сжимает данные с использованием алгоритма `deflate` и отправляет их в поток.
- `zlib.createInflate()` — выполняет декомпрессию `deflate`-потока.

Разные типы потоков

Существует четыре типа потоков:

- Поток для чтения (`Readable`) — это поток, из которого можно читать данные. Записывать данные в такой поток нельзя. Когда в такой поток поступают данные, они буферизуются до того момента пока потребитель данных не приступит к их чтению.
- Поток для записи (`Writable`) — это поток, в который можно отправлять данные. Читать из него данные нельзя.
- Дуплексный поток (`Duplex`) — в такой поток можно и отправлять данные и читать их из него. По существу это — комбинация потока для чтения и потока для записи.
- Трансформирующий поток (`Transform`) — такие потоки похожи на дуплексные потоки, разница заключается в том, что то, что поступает на вход этих потоков, преобразует то, что из них можно прочитать.

Создание потока для чтения

Поток для чтения можно создать и инициализировать, воспользовавшись возможностями модуля `stream`:

```
const Stream = require('stream')

const readableStream = new Stream.Readable()
```

Теперь в поток можно поместить данные, которые позже сможет прочесть потребитель этих данных:

```
readableStream.push('hi!')

readableStream.push('ho!')
```

Создание потока для записи

Для того чтобы создать записываемый поток нужно расширить базовый объект `Writable` и реализовать его метод `_write()`. Для этого сначала создадим соответствующий поток:

```
const Stream = require('stream')

const writableStream = new Stream.Writable()
```

Затем реализуем его метод `_write()`:

```
writableStream._write = (chunk, encoding, next) => {

  console.log(chunk.toString())

  next()

}
```

Теперь к такому потоку можно подключить поток, предназначенный для чтения:

```
process.stdin.pipe(writableStream)
```

Получение данных из потока для чтения

Для того чтобы получить данные из потока, предназначенного для чтения, воспользуемся потоком для записи:

```
const Stream = require('stream')

const readableStream = new Stream.Readable()

const writableStream = new Stream.Writable()

writableStream._write = (chunk, encoding, next) => {

  console.log(chunk.toString())

  next()

}

readableStream.pipe(writableStream)

readableStream.push('hi!')

readableStream.push('ho!')

readableStream.push(null)
```

Команда `readableStream.push(null)` сообщает об окончании вывода данных.

Работать с потоками для чтения можно и напрямую, обрабатывая событие `readable`:

```
readableStream.on('readable', () => {

  console.log(readableStream.read())

})
```

Отправка данных в поток для записи

Для отправки данных в поток для записи используется метод `write()`:

```
writableStream.write('hey!\n')
```

Сообщение потоку для записи о том, что запись данных завершена

Для того чтобы сообщить потоку для записи о том, что запись данных в него завершена, можно воспользоваться его методом `end()`:

```
writableStream.end()
```

Этот метод принимает несколько необязательных параметров. В частности, ему можно передать последнюю порцию данных, которые надо записать в поток.

Основы работы с MySQL в Node.js

MySQL является одной из самых популярных СУБД в мире. В экосистеме Node.js имеется несколько пакетов, которые позволяют взаимодействовать с MySQL-базами, то есть — сохранять в них данные, получать данные из баз и выполнять другие операции.

Мы будем использовать пакет [mysqljs/mysql](https://www.npmjs.com/package/mysqljs/mysql). Этот проект, который существует уже очень давно, собрал более 12000 звёзд на GitHub. Для того чтобы воспроизвести следующие примеры, вам понадобится MySQL-сервер.

Установка пакета

Для установки этого пакета воспользуйтесь такой командой:

```
npm install mysql
```

Инициализация подключения к базе данных

Сначала подключим пакет в программе:

```
const mysql = require('mysql')
```

После этого создадим соединение:

```
const options = {  
  user: 'the_mysql_user_name',  
  password: 'the_mysql_user_password',  
  database: 'the_mysql_database_name'  
}  
  
const connection = mysql.createConnection(options)
```

Теперь попытаемся подключиться к базе данных:

```
connection.connect(err => {  
  if (err) {  
    console.error('An error occurred while connecting to the DB')  
    throw err  
  }  
})
```

Параметры соединения

В вышеприведённом примере объект `options` содержал три параметра соединения:

```
const options = {  
  user: 'the_mysql_user_name',  
  password: 'the_mysql_user_password',  
  database: 'the_mysql_database_name'  
}
```

На самом деле этих параметров существует гораздо больше. В том числе — следующие:

- `host` — имя хоста, на котором расположен MySQL-сервер, по умолчанию — `localhost`.
- `port` — номер порта сервера, по умолчанию — `3306`.
- `socketPath` — используется для указания сокета Unix вместо хоста и порта.
- `debug` — позволяет работать в режиме отладки, по умолчанию эта возможность отключена.
- `trace` — позволяет выводить сведения о трассировке стека при возникновении ошибок, по умолчанию эта возможность включена.

- `ssl` — используется для настройки SSL-подключения к серверу.

Выполнение запроса SELECT

Теперь всё готово к выполнению SQL-запросов к базе данных. Для выполнения запросов используется метод соединения `query`, который принимает запрос и коллбэк. Если операция завершится успешно — коллбэк будет вызван с передачей ему данных, полученных из базы. В случае ошибки в коллбэк попадёт соответствующий объект ошибки. Вот как это выглядит при выполнении запроса на выборку данных:

```
connection.query('SELECT * FROM todos', (error, todos, fields) => {  
  if (error) {  
    console.error('An error occurred while executing the query')  
    throw error  
  }  
  console.log(todos)  
})
```

При формировании запроса можно использовать значения, которые будут автоматически встроены в строку запроса:

```
const id = 223  
  
connection.query('SELECT * FROM todos WHERE id = ?', [id], (error, todos, fields) => {  
  if (error) {  
    console.error('An error occurred while executing the query')  
    throw error  
  }  
  console.log(todos)  
})
```

Для передачи в запрос нескольких значений можно, в качестве второго параметра, использовать массив:

```
const id = 223  
  
const author = 'Flavio'  
  
connection.query('SELECT * FROM todos WHERE id = ? AND author = ?', [id, author], (error, todos, fields) => {  
  if (error) {  
    console.error('An error occurred while executing the query')  
    throw error  
  }  
})
```

```
}

console.log(todos)

})
```

Выполнение запроса INSERT

Запросы `INSERT` используются для записи данных в базу. Например, запишем в базу данных объект:

```
const todo = {

  thing: 'Buy the milk'

  author: 'Flavio'

}

connection.query('INSERT INTO todos SET ?', todo, (error, results, fields) => {

  if (error) {

    console.error('An error occurred while executing the query')

    throw error

  }

})
```

Если у таблицы, в которую добавляются данные, есть первичный ключ со свойством `auto_increment`, его значение будет возвращено в виде `results.insertId`:

```
const todo = {

  thing: 'Buy the milk'

  author: 'Flavio'

}

connection.query('INSERT INTO todos SET ?', todo, (error, results, fields) => {

  if (error) {

    console.error('An error occurred while executing the query')

    throw error

  }}

  const id = results.insertId

  console.log(id)

})
```

Заккрытие соединения с базой данных

После того как работа с базой данных завершена и пришло время закрыть соединение — воспользуйтесь его методом `end()`:

```
connection.end()
```

Это приведёт к правильному завершению работы с базой данных.

О разнице между средой разработки и продакшн-средой

Создавая приложения в среде Node.js можно использовать различные конфигурации для окружения разработки и продакшн-окружения.

По умолчанию платформа Node.js работает в окружении разработки. Для того чтобы указать ей на то, что код выполняется в продакшн-среде, можно настроить переменную окружения `NODE_ENV`:

```
NODE_ENV=production
```

Обычно это делается в командной строке. В Linux, например, это выглядит так:

```
export NODE_ENV=production
```

Лучше, однако, поместить подобную команду в конфигурационный файл наподобие `.bash_profile` (при использовании Bash), так как в противном случае такие настройки не сохраняются после перезагрузки системы.

Настроить значение переменной окружения можно, воспользовавшись следующей конструкцией при запуске приложения:

```
NODE_ENV=production node app.js
```

Эта переменная окружения широко используется во внешних библиотеках для Node.js.

Установка `NODE_ENV` в значение `production` обычно означает следующее:

- До минимума сокращается логирование.
- Используется больше уровней кэширования для оптимизации производительности.

Например, [Pug](#) — библиотека для работы с шаблонами, используемая Express, готовится к работе в режиме отладки в том случае, если переменная `NODE_ENV` не установлена в значение `production`. Представления Express, в режиме разработки, генерируются при обработке каждого запроса. В продакшн-режиме они кэшируются. Есть и множество других подобных примеров.

Express предоставляет конфигурационные хуки для каждого окружения. То, какой именно будет вызван, зависит от значения `NODE_ENV`:

```
app.configure('development', () => {  
  //...  
})  
  
app.configure('production', () => {  
  //...  
})  
  
app.configure('production', 'staging', () => {  
  //...  
})
```


Например, с их помощью можно использовать различные обработчики событий для разных режимов:

```
app.configure('development', () => {  
    app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));  
})  
  
app.configure('production', () => {  
    app.use(express.errorHandler())  
})
```

Надеемся, освоив это руководство, вы узнали о платформе Node.js достаточно много для того, чтобы приступить к работе с ней. Полагаем, теперь вы, даже если начали читать первую статью этого цикла, совершенно не разбираясь в Node.js, сможете начать писать что-то своё, с интересом читать чужой код и с толком пользоваться [документацией](#) к Node.js.

На всякий случай :)

Habrahabr10

Промо-код для скидки в 10% на наши виртуальные сервера