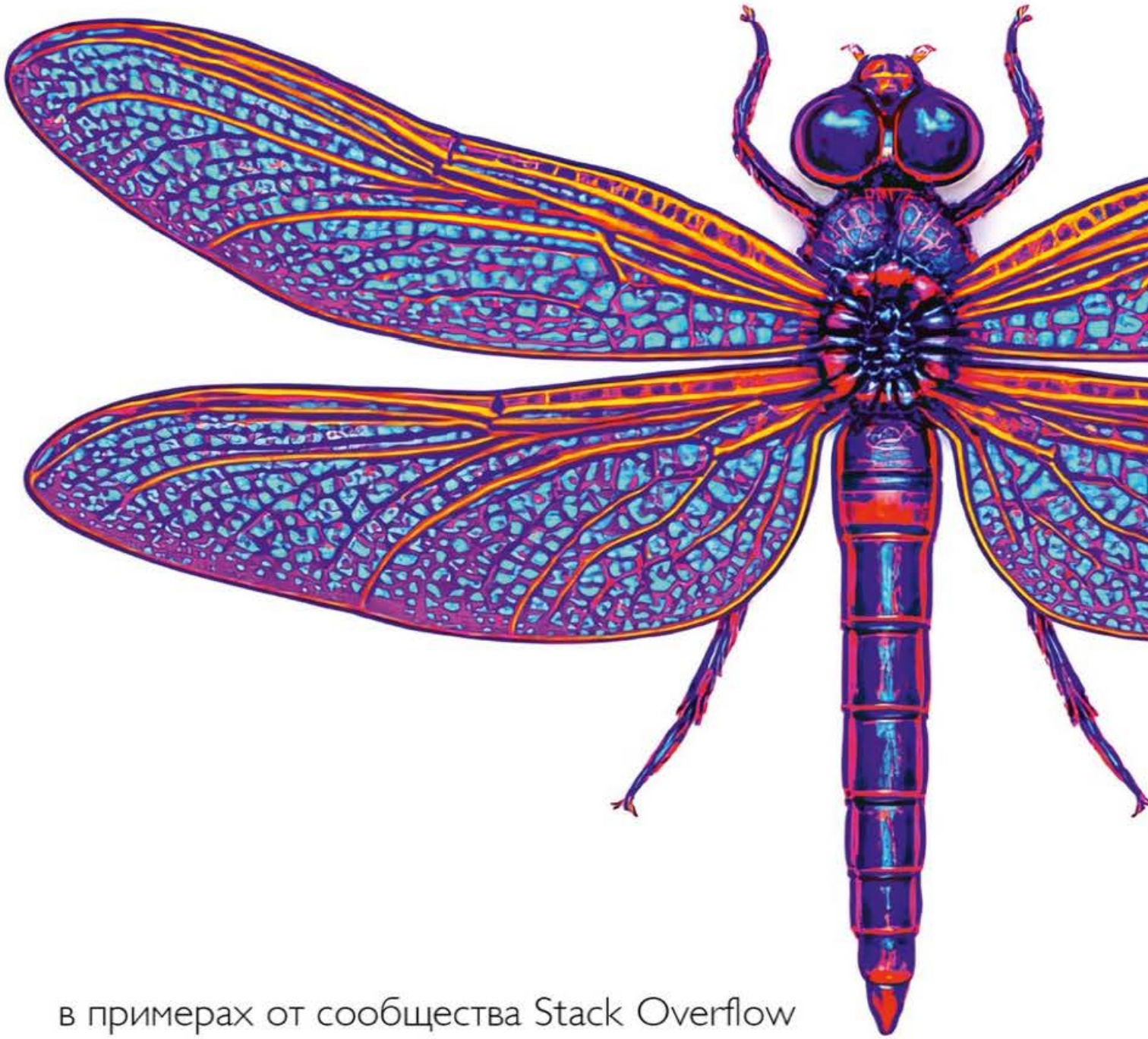
САМОЕ ПОЛНОЕ РУКОВОДСТВО ДЛЯ ВЕБ-РАЗРАБОТЧИКОВ

NODE.JS



в примерах от сообщества Stack Overflow

САМОЕ ПОЛНОЕ РУКОВОДСТВО
ДЛЯ ВЕБ-РАЗРАБОТЧИКОВ

NODE.JS

Москва
2025

УДК 004.43
ББК 32.973.26-018.1
Н85

Node.js. Самое полное руководство для веб-разработчиков в примерах
Н85 от сообщества Stack Overflow. — Москва : Издательство АСТ, 2026. — 480 с. : ил. — (Быстрый старт в программирование).

ISBN 978-5-17-162197-1.

Книга «Node.js. Самое полное руководство для веб-разработчиков в примерах от сообщества Stack Overflow» представляет собой продвинутое учебное пособие по работе на языке JavaScript в кроссплатформенной среде выполнения Node.js — чрезвычайно популярной и важнейшей в области веб-разработки. Вспомогательный контент для программистов с любым опытом и уровнем знаний основан на практических примерах кодов, написанных экспертами широко известного в профессиональных кругах сообщества Stack Overflow, где лучшие ИТ-специалисты со всего мира делятся своими наработками, отвечая на многие технические вопросы.

Издание содержит подробное объяснение важнейших концепций и приемов работы с Node.js, дополненное примерами их практического применения, что даст возможность молодым разработчикам программного обеспечения быстро повысить уровень своих компетенций, а опытным — найти актуальные решения для сложных задач. В книге приведен мощный и современный инструментарий для создания серверов, API, разработки веб-приложений, работы с базами данных, использования WebSocket. Отдельные главы посвящены работе с менеджерами пакетов NPM и Yarn, эффективной обработке событий и повышению производительности.

УДК 004.43
ББК 32.973.26-018.1

ISBN 978-5-17-162197-1

Перевод на русский язык: ООО «Интеджер»
Издание на русском языке: ООО «Издательство АСТ»

Содержание

Глава 1.
Начало работы с Node.js

Примечания

Node.js — это основанная на событиях неблокирующая асинхронная платформа ввода-вывода, использующая JavaScript-движок V8 от Google. Она применяется для разработки приложений, которые активно задействуют возможность выполнения JavaScript как на стороне клиента, так и на стороне сервера, что позволяет выгодно применять повторное использование кода и отсутствие переключения контекста. Node.js является открытым кроссплатформенным решением. Приложения Node.js пишутся на языке JavaScript и могут быть запущены в среде Node.js на Windows, Linux и т. д.

Версии

Версия	Дата выхода
v22.8.0	2024-09-03
v21.7.3	2024-04-10
v20.17.0	2024-08-21
v19.9.0	2023-04-10
v18.20.4	2024-07-08
v17.9.1	2022-06-01
v16.20.2	2023-08-08
v15.14.0	2021-04-06
v14.21.3	2023-02-16
v13.14.0	2020-04-29
v12.22.12	2022-04-05

v11.15.0	2019-04-30
v10.24.1	2021-04-06
v9.11.2	2018-06-12
v8.2.1	2017-07-20
v8.2.0	2017-07-19
v8.1.4	2017-07-11
v8.1.3	2017-06-29
v8.1.2	2017-06-15
v8.1.1	2017-06-13
v8.1.0	2017-06-08
v8.0.0	2017-05-30
v7.10.0	2017-05-02
v7.9.0	2017-04-11
v7.8.0	2017-03-29
v7.7.4	2017-03-21
v7.7.3	2017-03-14
v7.7.2	2017-03-08
v7.7.1	2017-03-02
v7.7.0	2017-02-28
v7.6.0	2017-02-21
v7.5.0	2017-01-31
v7.4.0	2017-01-04
v7.3.0	2016-12-20
v7.2.1	2016-12-06
v7.2.0	2016-11-22
v7.1.0	2016-11-08
v7.0.0	2016-10-25
v6.11.0	2017-06-06
v6.10.3	2017-05-02
v6.10.2	2017-04-04
v6.10.1	2017-03-21

v6.10.0	2017-02-21
v6.9.5	2017-01-31
v6.9.4	2017-01-05
v6.9.3	2017-01-05
v6.9.2	2016-12-06
v6.9.1	2016-10-19
v6.9.0	2016-10-18
v6.8.1	2016-10-14
v6.8.0	2016-10-12
v6.7.0	2016-09-27
v6.6.0	2016-09-14
v6.5.0	2016-08-26
v6.4.0	2016-08-12
v6.3.1	2016-07-21
v6.3.0	2016-07-06
v6.2.2	2016-06-16
v6.2.1	2016-06-02
v6.2.0	2016-05-17
v6.1.0	2016-05-05
v6.0.0	2016-04-26
v5.12.0	2016-06-23
v5.11.1	2016-05-05
v5.11.0	2016-04-21
v5.10.1	2016-04-05
v5.10	2016-04-01
v5.9	2016-03-16
v5.8	2016-03-09
v5.7	2016-02-23
v5.6	2016-02-09
v5.5	2016-01-21
v5.4	2016-01-06

v5.3	2015-12-15
v5.2	2015-12-09
v5.1	2015-11-17
v5.0	2015-10-29
v4.4	2016-03-08
v4.3	2016-02-09
v4.2	2015-10-12
v4.1	2015-09-17
v4.0	2015-09-08
io.js v3.3	2015-09-02
io.js v3.2	2015-08-25
io.js v3.1	2015-08-19
io.js v3.0	2015-08-04
io.js v2.5	2015-07-28
io.js v2.4	2015-07-17
io.js v2.3	2015-06-13
io.js v2.2	2015-06-01
io.js v2.1	2015-05-24
io.js v2.0	2015-05-04
io.js v1.8	2015-04-21
io.js v1.7	2015-04-17
io.js v1.6	2015-03-20
io.js v1.5	2015-03-06
io.js v1.4	2015-02-27
io.js v1.3	2015-02-20
io.js v1.2	2015-02-11
io.js v1.1	2015-02-03
io.js v1.0	2015-01-14
v0.12	2016-02-09
v0.11	2013-03-28
v0.10	2013-03-11

v0.9	2012-07-20
v0.8	2012-06-22
v0.7	2012-01-17
v0.6	2011-11-04
v0.5	2011-08-26
v0.4	2011-08-26
v0.3	2011-08-26
v0.2	2011-08-26
v0.1	2011-08-26

Примеры

HTTP-сервер «Hello World»

Сначала установите Node.js для вашей платформы.

В этом примере мы создадим HTTP-сервер, который будет слушать на порту 1337 и отправлять «`Hello, World!`» в браузер. Обратите внимание, что вместо порта 1337 вы можете использовать любой другой номер порта по вашему выбору, который в данный момент не задействован другими службами.

Модуль `http` является основным модулем Node.js (модулем, который включен в исходный код Node.js и не требует установки дополнительных ресурсов). Он предоставляет функциональность для создания HTTP-сервера с использованием метода `http.createServer()`. Чтобы начать разработку приложения, создайте файл, содержащий следующий JavaScript-код:

```
const http = require('http'); // Загружает модуль http

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

    // 1. Сообщаем браузеру, что все в порядке (Код состояния 200)
и данные в обычном тексте
    response.writeHead(200, {
        'Content-Type': 'text/plain'
    });

    // 2. Записываем объявленный текст в тело страницы
    response.write('Hello, World!\n');

    // 3. Сообщаем серверу, что все заголовки ответа и тело отправлены
    response.end();
```

```
}).listen(1337);
// 4. Сообщаем серверу, на каком порту слушать
```

Сохраните файл под любым именем. В данном случае, если мы назовем его `hello.js`, мы можем запустить приложение, перейдя в каталог, где находится файл, и используя следующую команду:

```
node hello.js
```

Созданный сервер затем будет доступен в браузере по URL `http://localhost:1337` или `http://127.0.0.1:1337`.

Появится простая веб-страница с текстом «Hello, World!» вверху, как показано на скриншоте ниже:

Пример «Hello World» в командной строке

Node.js также может быть использован для создания утилит командной строки. Пример ниже считывает первый аргумент из командной строки и выводит сообщение «Hello».

Чтобы запустить этот код на Unix-системе:

1. Создайте новый файл и вставьте в него код, представленный ниже. Имя файла не имеет значения.
2. Сделайте этот файл исполняемым с помощью команды `chmod 700 FILE_NAME`.
3. Запустите приложение командой `./APP_NAME David`.

На Windows выполните шаг 1 и запустите приложение командой `node APP_NAME David`.

```
#!/usr/bin/env node
'use strict';

/*
 Командные аргументы сохраняются в массиве `process.argv`,
 который имеет следующую структуру:
 [0] Путь к исполняемому файлу, который запустил процесс Node.js
 [1] Путь к этому приложению
 [2-n] Аргументы командной строки
 Пример: [ '/bin/node', '/path/to/yourscript', 'arg1', 'arg2', ... ]
 src: https://nodejs.org/api/process.html#process_process_argv
*/

// Сохраняем первый аргумент как имя пользователя.
var username = process.argv[2];

// Проверяем, предоставлено ли имя пользователя.
if (!username) {
    // Извлекаем имя файла.

    var appName = process.argv[1].split(require('path').sep).pop();
    // Даем пользователю пример, как использовать приложение.

    console.error('Missing argument! Example: %s YOUR_NAME', appName);
    // Завершаем выполнение приложения (успех: 0, ошибка: 1).
    // Ошибка остановит выполнение цепочки команд. Например:
    // ./app.js && ls -> команда ls не выполнится
    // ./app.js David && ls -> команда ls выполнится

    process.exit(1);
}

// Выводим сообщение в консоль.
console.log('Hello %s!', username);
```

Установка и запуск Node.js

Для начала установите Node.js на ваш рабочий компьютер.

Windows: Перейдите на страницу загрузки по ссылке https://nodejs.org/en/download/package-manager или по QR-коду ниже и скачайте/запустите установочный файл.

Mac: Перейдите на страницу загрузки по ссылке https://nodejs.org/en/download/package-manager или по QR-коду ниже и скачайте/запустите установочный файл.

Альтернативно вы можете установить Node через Homebrew, используя команду `brew install node`. Homebrew — это менеджер пакетов командной строки для Macintosh, и дополнительную информацию о нем можно найти на сайте Homebrew, перейдя по ссылке https://brew.sh/ или по QR-коду ниже.

Linux: Следуйте инструкциям для вашей дистрибуции на странице установки из командной строки, перейдя по ссылке https://nodejs.org/en/download/package-manager/ или по QR-коду на следующей странице.

Запуск программы Node.js

Чтобы запустить программу на Node.js, выполните команду `node app.js` или `nodejs app.js`, где `app.js` — это имя файла с исходным кодом вашего Node-приложения. Вам не нужно включать суффикс `.js`, чтобы Node нашел скрипт, который вы хотите запустить.

Альтернативно в операционных системах на базе UNIX программа Node может быть выполнена как скрипт терминала. Для этого она должна начинаться с «шейбэнга», указывающего на интерпретатор Node, такого как `#!/usr/bin/env node`. Файл также должен быть помечен как исполняемый, что можно сделать с помощью команды `chmod`. Теперь скрипт можно запускать непосредственно из командной строки.

Развертывание вашего приложения в онлайн-среде

Когда вы разворачиваете ваше приложение в среде хостинга (специфичной для Node.js), эта среда обычно предоставляет переменную окружения PORT, которую вы можете использовать для запуска вашего сервера. Изменение номера порта на `process.env.PORT` позволяет получить доступ к приложению.

Например:

```
http.createServer(function(request, response) {
    // ваш код сервера
}).listen(process.env.PORT);
```

Также если вы хотите получить доступ к приложению офлайн во время отладки, вы можете использовать следующее:

```
http.createServer(function(request, response) {
    // ваш код сервера
}).listen(process.env.PORT || 3000);
```

...где `3000` — это номер порта для офлайн-доступа.

Отладка вашего Node.js-приложения

Вы можете использовать `node-inspector`. Выполните следующую команду, чтобы установить его через NPM:

```
npm install -g node-inspector
```

Затем вы можете отлаживать ваше приложение, используя:

```
node-debug app.js
```

Репозиторий на GitHub можно найти здесь: https://github.com/node-inspector/node-inspector или по QR-коду ниже.

Отладка нативно

Вы также можете отлаживать Node.js нативно, запустив его следующим образом:

```
node debug your-script.js
```

Чтобы установить точку останова (breakpoint) в нужной строке кода, используйте это:

```
debugger;
```

В Node.js версии 8 используйте следующую команду:

```
node --inspect-brk your-script.js
```

Затем откройте `about://inspect` в последней версии Google Chrome и выберите свой Node-скрипт, чтобы получить опыт отладки с помощью DevTools Chrome.

«Hello World» с использованием Express

В следующем примере используется Express для создания HTTP-сервера, который слушает порт 3000 и отвечает «`Hello, World!`». Express — это широко применяемый веб-фреймворк, который полезен для создания HTTP API.

Сначала создайте новую папку, например `myApp`. Перейдите в папку `myApp` и создайте новый JavaScript-файл, содержащий следующий код (назовем его, например, `hello.js`). Затем установите модуль `express`, используя команду `npm install --save express` в командной строке. Ознакомьтесь с документацией по ссылке https://riptutorial.com/node-js/example/1588/installing-packages или по QR-коду ниже для получения дополнительной информации о том, как устанавливать пакеты.

```
// Импортируем верхнеуровневую функцию express
const express = require('express');

// Создаем приложение Express, используя верхнеуровневую функцию
const app = express();

// Определяем номер порта как 3000
const port = 3000;

// Обрабатываем HTTP GET-запросы по указанному пути "/" с указанной
функцией обратного вызова
app.get('/', function(request, response) {
    response.send('Hello, World!');
});

// Заставляем приложение слушать на порту 3000
app.listen(port, function() {
    console.log('Server listening on http://localhost:' + port);
});
```

Из командной строки выполните следующую команду:

```
node hello.js
```

Откройте браузер и перейдите по адресу `http://localhost:3000` или `http://127.0.0.1:3000`, чтобы увидеть ответ.

«Hello World» с базовой маршрутизацией

Как только вы поймете, как создать HTTP-сервер с Node.js, важно понять, как заставить его «делать» что-то в зависимости от пути, по которому пользователь перешел. Этот механизм называется маршрутизацией.

Самый простой пример — проверка `if (request.url === 'some/path/here')` и затем вызов функции, которая отвечает новым файлом.

Пример этого можно увидеть здесь:

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

function index (request, response) {
    response.writeHead(200);
    response.end('Hello, World!');
}

http.createServer(function (request, response) {
    if (request.url === '/') {
        return index(request, response);
    }
    response.writeHead(404);
    response.end(http.STATUS_CODES[404]);
}).listen(1337);
```

Однако если продолжать определять «маршруты» таким образом, вы в конечном итоге получите одну огромную функцию обратного вызова, что может создать беспорядок. Давайте попробуем оптимизировать это.

Сначала сохраним все наши маршруты в объекте:

```
var routes = {
    '/': function index (request, response) {
        response.writeHead(200);
        response.end('Hello, World!');
    },
    '/foo': function foo (request, response) {
        response.writeHead(200);
        response.end('You are now viewing "foo"');
    }
}
```

Теперь, когда мы сохранили два маршрута в объекте, мы можем проверить их наличие в нашей основной функции обратного вызова:

```
http.createServer(function (request, response) {
    if (request.url in routes) {
        return routes[request.url](request, response);
    }
```

```
    response.writeHead(404);
    response.end(http.STATUS_CODES[404]);

}).listen(1337);
```

Теперь каждый раз, когда вы пытаетесь перейти по какому-либо пути на вашем сайте, сервер будет проверять наличие этого пути в ваших маршрутах и вызовет соответствующую функцию. Если маршрут не найден, сервер ответит кодом 404 (`Not Found`).

Вот и все — маршрутизация с использованием API HTTP-сервера очень проста.

TLS Socket: сервер и клиент

Единственные значительные отличия между этим и обычным TCP-соединением — это закрытый ключ и публичный сертификат, которые вам нужно будет установить в объекте опций.

Как создать ключ и сертификат

Первым шагом по обеспечению безопасности является создание приватного ключа. Что же такое приватный ключ? По сути, это набор случайного шума, который используется для шифрования информации. Теоретически вы могли бы создать один ключ и использовать его для шифрования чего угодно. Но лучшая практика — иметь разные ключи для разных целей, потому что если кто-то украдет ваш приватный ключ, это похоже на кражу ключей от вашего дома. Представьте, что могло бы произойти, если бы вы использовали один и тот же ключ для запирания машины, гаража, офиса и т. д.

```
openssl genrsa -out private-key.pem 1024
```

Теперь, когда у нас есть приватный ключ, мы можем создать CSR (запрос на подпись сертификата), который является нашим запросом на подпись приватного ключа авторитетной организацией. Именно поэтому нужно вводить информацию, связанную с вашей компанией.

Эта информация будет видна подписывающему органу и использована для вашей верификации. В нашем случае неважно, что вы введете, так как на следующем шаге мы сами подпишем наш сертификат:

```
openssl req -new -key private-key.pem -out csr.pem
```

Теперь, когда у нас есть все документы, пришло время притвориться, будто мы — авторитетный подписывающий орган власти:

```
openssl x509 -req -in csr.pem -signkey private-key.pem -out public-
cert.pem
```

Теперь, когда у вас есть приватный ключ и публичный сертификат, вы можете установить защищенное соединение между двумя приложениями Node.js. Как видно в примере кода, это очень простой процесс.

Важно!

Поскольку мы создали публичный сертификат сами, честно говоря, он бесполезен, потому что мы на самом деле никем не являемся. Node.js-сервер по умолчанию не будет доверять такому сертификату, и именно поэтому нам нужно указать ему доверять сертификату с помощью опции `rejectUnauthorized: false`. Очень важно: никогда не устанавливайте эту переменную в `true` в производственной среде.

TLS Socket Server

```
'use strict';
var tls = require('tls');
var fs = require('fs');
const PORT = 1337;
const HOST = '127.0.0.1'
var options = {
    key: fs.readFileSync('private-key.pem'),
    cert: fs.readFileSync('public-cert.pem')
};
var server = tls.createServer(options, function(socket) {
    // Отправить дружеское сообщение
    socket.write("I am the server sending you a message.");
    // Вывести данные, которые мы получили
    socket.on('data', function(data) {
        console.log('Received: %s [it is %d bytes long]',
        data.toString().replace(/(\n)/gm,""),
        data.length);
    });
    // Сообщить нам, когда передача закончена
    socket.on('end', function() {
        console.log('EOT (End Of Transmission)');
    });
});
// Начать прослушивание на конкретном порту и адресе
server.listen(PORT, HOST, function() {
    console.log("I'm listening at %s, on port %s", HOST, PORT);
});
// При возникновении ошибки отобразить ее.
server.on('error', function(error) {
    console.error(error);
```

```
    // Закрыть соединение после возникновения ошибки
    server.destroy();
});
TLS Socket Client
'use strict';
var tls = require('tls');
var fs = require('fs');
const PORT = 1337;
const HOST = '127.0.0.1'
// Передать сертификаты серверу и указать ему обрабатывать даже неав-
торизованные сертификаты
var options = {
    key: fs.readFileSync('private-key.pem'),
    cert: fs.readFileSync('public-cert.pem'),
    rejectUnauthorized: false
};
var client = tls.connect(PORT, HOST, options, function() {
    // Проверить, сработала ли авторизация
    if (client.authorized) {
      console.log("Connection authorized by a Certificate Authority.");
    } else {
              console.log("Connection  not  authorized:  "  +  client.
authorizationError)
    }
    // Отправить дружеское сообщение
    client.write("I am the client sending you a message.");
});
client.on("data", function(data) {
    console.log('Received: %s [it is %d bytes long]',
    data.toString().replace(/(\n)/gm,""),
    data.length);
    // Закрыть соединение после получения сообщения
    client.end();
});
client.on('close', function() {
    console.log("Connection closed");
});
// При возникновении ошибки отобразить ее
client.on('error', function(error) {
    console.error(error);
    // Закрыть соединение после возникновения ошибки
    client.destroy();
});
```

Hello World в REPL

Когда Node.js вызывается без аргументов, он запускает REPL (Read-Eval-Print-Loop), также известный как Node shell.

В командной строке введите node:

```
$ node
>
```

На приглашении Node shell > введите «Hello World!»:

```
$ node
> "Hello World!"
'Hello World!'
```

Core-модули

Node.js — это движок JavaScript (движок V8 от Google для Chrome, написанный на C++), который позволяет запускать JavaScript вне браузера. Хотя для расширения возможностей Node.js доступно множество библиотек, движок поставляется с набором core-модулей, реализующих основные функции.

В настоящее время в Node.js включено 34 core-модуля:

```
[ 'assert',
  'buffer',
  'c/c++_addons',
  'child_process',
  'cluster',
  'console',
  'crypto',
  'deprecated_apis',
  'dns',
  'domain',
  'events',
  'fs',
  'http',
  'https',
  'module',
  'net',
  'os',
  'path',
  'punycode',
  'querystring',
  'readline',
  'repl',
  'stream',
```

```
'string_decoder',
'timers',
'tls_(ssl)',
'tracing',
'tty',
'dgram',
'url',
'util',
'v8',
'vm',
'zlib' ]
```

Этот список был получен из API-документации Node.js, на который можно перейти по ссылке https://nodejs.org/api/all.html или по QR-коду ниже.

Все core-модули: краткий обзор

assert

Модуль `assert` предоставляет простой набор тестов утверждений, которые можно использовать для проверки инвариантов.

buffer

До введения `TypedArray` в ECMAScript 2015 (ES6) в языке JavaScript не было механизма для чтения или манипулирования потоками бинарных данных. Класс `Buffer` был введен как часть API Node.js, чтобы сделать возможным взаимодействие с октетными потоками в контексте таких вещей, как TCP-потоки и операции с файловой системой.

Теперь, когда `TypedArray` добавлен в ES6, класс `Buffer` реализует API `Uint8Array` таким образом, который более оптимизирован и подходит для случаев использования Node.js.

c/c++_addons

`Addons` в Node.js — это динамически связываемые общие объекты, написанные на C или C++, которые могут быть загружены в Node.js с помощью функции

`require()` и использоваться так же, как обычный модуль Node.js. Они применяются в основном для обеспечения интерфейса между JavaScript, выполняющимся в Node.js, и библиотеками на C/C++.

child_process

Модуль `child_process` предоставляет возможность порождать дочерние процессы аналогично, но не идентично `popen(3)`.

cluster

Одна инстанция Node.js работает в одном потоке. Чтобы воспользоваться преимуществами многопроцессорных систем, иногда нужно запустить кластер процессов Node.js для обработки нагрузки. Модуль `cluster` позволяет легко создавать дочерние процессы, которые разделяют серверные порты.

console

Модуль `console` предоставляет простой отладочный консольный интерфейс, похожий на механизм консоли JavaScript, предоставляемый веб-браузерами.

crypto

Модуль `crypto` предоставляет криптографические функции, включая набор оберток для функций `OpenSSL`, таких как `hash`, `HMAC`, `cipher`, `decipher`, `sign` и `verify`.

deprecated_apis

Node.js может «устаревать» API в следующих случаях: (a) использование API считается небезопасным, (b) доступна улучшенная альтернатива API или (c) ожидаются изменения, несовместимые с предыдущими версиями, в будущем крупном релизе.

dns

Модуль `dns` содержит функции, принадлежащие к двум разным категориям:

1. Функции, использующие возможности операционной системы для выполнения разрешения имен и не обязательно выполняющие сетевое взаимодействие. В эту категорию входит только одна функция: `dns.lookup()`.

2. Функции, которые подключаются к реальному DNS-серверу для выполнения разрешения имен и всегда используют сеть для выполнения DNS-запросов. В эту категорию входят все функции в модуле dns, кроме `dns.lookup()`.

domain

Этот модуль вскоре станет устаревшим. Как только будет окончательно утверждено заменяющее API, этот модуль станет полностью устаревшим. Большинство конечных пользователей не должны использовать этот модуль. Пользователи, которым необходима функциональность, предоставляемая доменами, могут полагаться на него в настоящее время, но должны быть готовы к переходу на другое решение в будущем.

Events

Большая часть основного API Node.js построена вокруг идиоматической асинхронной событийно-ориентированной архитектуры, в которой определенные типы объектов (называемые `emitters`) периодически испускают именованные события, которые вызывают объекты `Function` (называемые `listeners`).

fs

Ввод-вывод файлов осуществляется с помощью простых оберток вокруг стандартных функций POSIX. Чтобы использовать этот модуль, нужно вызвать `require('fs')`. Все методы имеют асинхронные и синхронные формы.

http

Интерфейсы HTTP в Node.js спроектированы таким образом, чтобы поддерживать многие особенности протокола, которые традиционно трудно использовать. В частности, это крупные, возможно, кодируемые фрагментами сообщения. Интерфейс никогда не буферизует целые запросы или ответы — пользователь может передавать данные в потоке.

https

HTTPS — это протокол HTTP поверх TLS/SSL. В Node.js это реализовано как отдельный модуль.

module

Node.js имеет простую систему загрузки модулей. В Node.js файлы и модули находятся в соответствии одно к одному (каждый файл рассматривается как отдельный модуль).

net

Модуль `net` предоставляет вам асинхронную сетевую оболочку. Он содержит функции для создания как серверов, так и клиентов (называемых потоками). Вы можете подключить этот модуль с помощью `require('net')`.

os

Модуль `os` предоставляет множество утилит, связанных с операционной системой.

path

Модуль `path` предоставляет утилиты для работы с путями к файлам и каталогам.

punycode

Версия модуля punycode, включенная в Node.js, устаревает.

querystring

Модуль `querystring` предоставляет утилиты для разбора и форматирования строк запросов URL.

readline

Модуль `readline` предоставляет интерфейс для чтения данных из потока `Readable` (например `process.stdin`) построчно.

repl

Модуль `repl` предоставляет реализацию Read-Eval-Print-Loop (REPL), которая доступна как отдельная программа или может быть включена в другие приложения.

stream

`Stream` — это абстрактный интерфейс для работы с потоковыми данными в Node.js. Модуль `stream` предоставляет базовое API, которое упрощает создание объектов, реализующих интерфейс потока.

В Node.js предоставляется множество объектов `stream`. Например, запрос к HTTP-серверу и `process.stdout` являются экземплярами `stream`.

string_decoder

Модуль `string_decoder` предоставляет API для декодирования объектов `Buffer` в строки таким образом, который сохраняет закодированные многобайтовые символы UTF-8 и UTF-16.

timers

Модуль `timer` предоставляет глобальный API для планирования вызовов функций в определенное будущее время. Поскольку функции таймера являются глобальными, нет необходимости вызывать `require('timers')` для использования API.

Функции таймера в Node.js реализуют API, аналогичное API таймеров, предоставляемому веб-браузерами, но используют другую внутреннюю реализацию, построенную вокруг Event Loop Node.js.

tls (ssl)

Модуль `tls` предоставляет реализацию протоколов Transport Layer Security (TLS) и Secure Socket Layer (SSL), построенную на базе `OpenSSL`.

tracing

Trace Event предоставляет механизм для централизованного сбора информации о трассировке, генерируемой V8, ядром Node и пользовательским кодом.

Трассировка может быть включена с помощью передачи флага `--trace-events-enabled` при запуске приложения Node.js.

tty

Модуль `tty` предоставляет классы `tty.ReadStream` и `tty.WriteStream`. В большинстве случаев не потребуется или не удастся использовать этот модуль напрямую.

dgram

Модуль `dgram` предоставляет реализацию UDP Datagram-сокетов.

url

Модуль `url` предоставляет утилиты для разрешения и разбора URL.

util

Модуль `util` предназначен в первую очередь для поддержки внутренних API Node.js. Однако многие утилиты полезны для разработчиков приложений и модулей.

v8

Модуль `v8` предоставляет API, специфичные для версии V8, встроенной в бинарный файл Node.js.

Примечание: API и реализация могут изменяться в любое время.

vm

Модуль `vm` предоставляет API для компиляции и выполнения кода в контексте виртуальной машины V8. Код JavaScript может быть скомпилирован и выполнен немедленно или скомпилирован, сохранен и выполнен позже.

Примечание: модуль `vm` не является механизмом безопасности. Не используйте его для выполнения ненадежного кода.

zlib

Модуль `zlib` предоставляет функциональность сжатия, реализованную с использованием `Gzip` и `Deflate/Inflate`.

Как запустить базовый HTTPS веб-сервер?

После того как Node.js установлен на вашу систему, вы можете просто следовать процедуре, представленной ниже, чтобы запустить базовый веб-сервер с поддержкой как HTTP, так и HTTPS.

Шаг 1: создание центра сертификации

1. Создайте папку, где вы хотите хранить свой ключ и сертификат:

```
mkdir conf
```

2. Перейдите в эту директорию (папку):

```
cd conf
```

3. Скачайте файл `ca.cnf` для использования в качестве конфигурационного ярлыка:

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/ca.cnf
```

4. Создайте новый центр сертификации, используя эту конфигурацию:

```
openssl req -new -x509 -days 9999 -config ca.cnf -keyout ca-key.pem -out ca-cert.pem
```

5. Теперь, когда у нас есть центр сертификации в `ca-key.pem` и `ca-cert.pem`, сгенерируем приватный ключ для сервера:

```
openssl genrsa -out key.pem 4096
```

6. Скачайте файл `server.cnf` для использования в качестве конфигурационного ярлыка:

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/server.cnf
```

7. Сгенерируйте запрос на подпись сертификата, используя эту конфигурацию:

```
openssl req -new -config server.cnf -key key.pem -out csr.pem
```

8. Подпишите запрос:

```
openssl x509 -req -extfile server.cnf -days 999 -passin "pass:password"
-in csr.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out
cert.pem
```

Шаг 2: установка вашего сертификата в качестве корневого

1. Скопируйте ваш сертификат в папку с корневыми сертификатами:

```
sudo cp ca-crt.pem /usr/local/share/ca-certificates/ca-crt.pem
```

2. Обновите хранилище сертификатов CA:

```
sudo update-ca-certificates
```

Шаг 3: запуск вашего Node-сервера

Сначала создайте файл `server.js`, который будет содержать код вашего сервера.

Минимальная настройка для HTTPS-сервера в Node.js может выглядеть так:

```
var https = require('https');
var fs = require('fs');
var httpsOptions = {
    key: fs.readFileSync('path/to/server-key.pem'),
    cert: fs.readFileSync('path/to/server-crt.pem')
};
var app = function (req, res) {
    res.writeHead(200);
    res.end("hello world\n");
}
https.createServer(httpsOptions, app).listen(4433);
```

Если вы также хотите поддерживать HTTP-запросы, вам нужно внести лишь одно небольшое изменение:

```
var http = require('http');
var https = require('https');
var fs = require('fs');
var httpsOptions = {
    key: fs.readFileSync('path/to/server-key.pem'),
    cert: fs.readFileSync('path/to/server-crt.pem')
};
var app = function (req, res) {
    res.writeHead(200);
    res.end("hello world\n");
}
```

```
http.createServer(app).listen(8888);
https.createServer(httpsOptions, app).listen(4433);
```

1. Перейдите в директорию (или папку), где находится ваш `server.js`:

```
cd /path/to
```

2. Запустите `server.js`:

```
node server.js
```

Глава 2.
Взаимодействие Arduino с Node.js

Введение

В этой главе представлены способы, позволяющие Node.js взаимодействовать с Arduino Uno.

Примеры

Связь Node.js с Arduino через serialport

Код Node.js

Пример, с которого можно начать эту тему — сервер Node.js, взаимодействующий с Arduino через serialport.

```
npm install express --save
npm install serialport -save
```

Пример `app.js`:

```
const express = require('express');
const app = express();
var SerialPort = require("serialport");

var port = 3000;

var arduinoCOMPort = "COM3";

var arduinoSerialPort = new SerialPort(arduinoCOMPort, {
    baudrate: 9600
});

arduinoSerialPort.on('open', function() {
    console.log('Serial Port ' + arduinoCOMPort + ' is opened.');
});
```

```
app.get('/', function (req, res) {

    return res.send('Working');

});

app.get('/:action', function (req, res) {

    var action = req.params.action || req.param('action');

    if(action == 'led'){
        arduinoSerialPort.write("w");
        return res.send('Led light is on!');
    }
    if(action == 'off') {
        arduinoSerialPort.write("t");
        return res.send("Led light is off!");
    }

    return res.send('Action: ' + action);

});

app.listen(port, function () {
    console.log('Example app listening on port http://0.0.0.0:' + port
+ '!');
});
```

Запуск примера Express-сервера:

```
node app.js
```

Код Arduino

```
// Функция setup выполняется один раз при нажатии кнопки сброса
или при включении питания платы
void setup() {
    // инициализируем цифровой вывод LED_BUILTIN как выход.

    Serial.begin(9600); // Начинаем прослушивание на порту 9600

    pinMode(LED_BUILTIN, OUTPUT);

    digitalWrite(LED_BUILTIN, LOW);
}
```

```
// Функция loop выполняется снова и снова
void loop() {

    if(Serial.available() > 0) // Чтение данных из последовательного
порта
    {
        char ReaderFromNode; // Сохраняем текущий символ
        ReaderFromNode = (char) Serial.read();
        convertToState(ReaderFromNode); // Преобразуем символ в состояние
    }
    delay(1000);
}

void convertToState(char chr) {
    if(chr=='o'){
        digitalWrite(LED_BUILTIN, HIGH);
        delay(100);
    }
    if(chr=='f'){
        digitalWrite(LED_BUILTIN, LOW);
        delay(100);
    }
}
```

Запуск

1. Подключите Arduino к вашему компьютеру.
2. Запустите сервер.

Управляйте встроенным светодиодом через сервер Node.js Express.
Чтобы включить светодиод:

```
http://0.0.0.0:3000/led
```

Чтобы выключить светодиод:

```
http://0.0.0.0:3000/off
```

Глава 3.
async.js

Синтаксис

- Каждая функция обратного вызова (callback) должна быть написан с использованием следующего синтаксиса:

```
function callback(err, result [, arg1[, ...]])
```

- Таким образом, вы обязаны сначала вернуть ошибку и не сможете игнорировать их обработку в будущем. В отсутствие ошибок принято использовать `null`.

```
callback(null, myResult);
```

- Ваши `callback` могут содержать больше аргументов, чем `err` и `result`, но это полезно только для определенного набора функций (например, waterfall, seq и т. д.).

```
callback(null, myResult, myCustomArgument);
```

- И, конечно, отправляйте ошибки. Вы должны это делать и обрабатывать их (или, по крайней мере, регистрировать).

```
callback(err);
```

Примеры

Параллельное выполнение: многозадачность

`async.parallel(tasks, afterTasksCallback)` выполнит набор задач параллельно и будет ждать завершения всех задач (что будет отмечено вызовом функции обратного вызова). Когда задачи завершены, `async` вызывает основной `callback` со всеми ошибками и результатами задач:

```
function shortTimeFunction(callback) {
    setTimeout(function() {
        callback(null, 'resultOfShortTime');
    }, 200);
}
function mediumTimeFunction(callback) {
    setTimeout(function() {
        callback(null, 'resultOfMediumTime');
    }, 500);
}

function longTimeFunction(callback) {
    setTimeout(function() {
        callback(null, 'resultOfLongTime');
    }, 1000);
}
async.parallel([
    shortTimeFunction,
    mediumTimeFunction,
    longTimeFunction
    ],
    function(err, results) {
        if (err) {
            return console.error(err);
        }
        console.log(results);
    });
```

Результат: [«resultOfShortTime», «resultOfMediumTime», «resultOfLongTime»].

Вызов async.parallel() с объектом

Вы можете заменить параметр массива задач объектом. В этом случае результаты также будут объектом с теми же ключами, что и задачи.

Это очень удобно для выполнения нескольких задач и простого поиска каждого результата:

```
async.parallel({
    short: shortTimeFunction,
    medium: mediumTimeFunction,
    long: longTimeFunction
    },
    function(err, results) {
```

```
        if (err) {
            return console.error(err);
        }

        console.log(results);
    });
```

Результат: {short: «resultOfShortTime», medium: «resultOfMediumTime», long: «resultOfLongTime»}.

Обработка нескольких значений

Каждой параллельной функции передается callback. Этот callback может вернуть ошибку в качестве первого аргумента или успешные значения после этого. Если в функцию обратного вызова передано несколько успешных значений, эти результаты возвращаются в виде массива:

```
async.parallel({
    short: function shortTimeFunction(callback) {
        setTimeout(function() {
          callback(null, 'resultOfShortTime1', 'resultOfShortTime2');
        }, 200);
    },

    medium: function mediumTimeFunction(callback) {
        setTimeout(function() {
          callback(null, 'resultOfMediumTime1', 'resultOfMeiumTime2');
        }, 500);
    }

    },
    function(err, results) {
        if (err) {
            return console.error(err);
        }
        console.log(results);
    });
```

Результат:

```
{
    short: ["resultOfShortTime1", "resultOfShortTime2"],
    medium: ["resultOfMediumTime1", "resultOfMediumTime2"]
}
```

Последовательное выполнение: независимый монотаскинг

`async.series(tasks, afterTasksCallback)` выполнит набор задач. Каждая задача выполняется последовательно одна за другой. Если какая-то задача не выполнится, async немедленно прекращает выполнение и переходит к основному `callback`.

Когда задачи успешно завершены, async вызывает «главный» `callback` со всеми ошибками и результатами всех задач:

```
function shortTimeFunction(callback) {
    setTimeout(function() {
        callback(null, 'resultOfShortTime');
    }, 200);
}
function mediumTimeFunction(callback) {
    setTimeout(function() {
        callback(null, 'resultOfMediumTime');
    }, 500);
}
function longTimeFunction(callback) {
    setTimeout(function() {
        callback(null, 'resultOfLongTime');
    }, 1000);
}
async.series([
    mediumTimeFunction,
    shortTimeFunction,
    longTimeFunction
    ],
    function(err, results) {
        if (err) {
            return console.error(err);
        }
        console.log(results);
    });
```

Результат: [«resultOfMediumTime», «resultOfShortTime», «resultOfLongTime»].

Вызов async.series() с объектом

Вы можете заменить параметр массива задач объектом. В этом случае результаты также будут объектом с теми же ключами, что и задачи. Это очень удобно для выполнения нескольких задач и простого поиска каждого результата.

```
async.series({
```

```
    short: shortTimeFunction,
    medium: mediumTimeFunction,
    long: longTimeFunction
    },
    function(err, results) {
        if (err) {
            return console.error(err);
        }
        console.log(results);
    });
```

Результат: {short: "resultOfShortTime", medium: "resultOfMediumTime", long: "resultOfLongTime"}.

Водопад (waterfall): зависимый монотаскинг

async.waterfall(tasks, afterTasksCallback) выполняет набор задач. Каждая задача выполняется последовательно, и результат одной задачи передается в следующую задачу. Если какая-то задача не выполнится, async, как и async.series(), прекращает выполнение и немедленно вызывает основной callback.

Когда задачи успешно завершены, async вызывает «главный» callback со всеми ошибками и результатами всех задач.

```
function getUserRequest(callback) {
    // Имитация запроса с помощью тайм-аута
    setTimeout(function() {
        var userResult = {
            name: 'Aamu'
        };
        callback(null, userResult);
    }, 500);
}

function getUserFriendsRequest(user, callback) {
    // Еще одна имитация запроса с помощью тайм-аута
    setTimeout(function() {
        var friendsResult = [];

        if (user.name === "Aamu") {
            friendsResult = [{
                name: 'Alice'
            }, {
                name: 'Bob'
            }];
        }
```

```
        callback(null, friendsResult);
    }, 500);
}

async.waterfall([
    getUserRequest,
    getUserFriendsRequest
    ],
    function(err, results) {
        if (err) {
            return console.error(err);
        }

        console.log(JSON.stringify(results));
    });
```

Результат: `results` содержит второй параметр `callback` последней функции водопада (waterfall), который в данном случае является `friendsResult`.

async.times (для более удобной обработки циклов)

Для выполнения функции в цикле в Node.js можно применять цикл `for` для коротких циклов. Но если цикл длинный, использование `for` может увеличить время обработки, что может привести к зависанию процесса Node. В таких случаях можно задействовать `async.times`:

```
function recursiveAction(n, callback) {
    // Выполните необходимые действия многократно
    callback(err, result);
}
async.times(5, function(n, next) {
    recursiveAction(n, function(err, result) {
        next(err, result);
    });
}, function(err, results) {
    // Теперь у нас должно быть 5 результатов
});
```

Это вызывается параллельно. Если мы хотим вызывать это по одному за раз, используйте `async.timesSeries`.

async.each (для эффективной обработки массива данных)

Когда мы хотим обработать массив данных, лучше использовать `async.each`. Когда нужно выполнить что-то с каждым элементом данных и получить окончательный `callback` после завершения всех операций, этот метод будет полезен. Это выполняется параллельно.

```
function createUser(userName, callback) {
    // Создание пользователя в базе данных
    callback(null); // или ошибка в зависимости от результата создания
}

var arrayOfData = ['Ritu', 'Sid', 'Tom'];
async.each(arrayOfData, function(eachUserName, callback) {
    // Выполняем операцию для каждого пользователя.
    console.log('Creating user ' + eachUserName);
     // Возврат callback обязателен. Иначе мы не получим финальный
callback, даже если пропустим один.
    createUser(eachUserName, callback);

}, function(err) {
    // Если создание любого из пользователей не удалось, может возник-
нуть ошибка.
    if (err) {
        // Одна из итераций вызвала ошибку.
        // Все дальнейшие процессы будут остановлены.
        console.log('unable to create user');
    } else {
        console.log('All user created successfully');
    }
});
```

Если необходимо выполнять одно действие за раз, можно использовать `async.eachSeries`.

async.series (для обработки событий по одному)

В `async.series` все функции выполняются последовательно, и объединенные результаты каждой функции передаются в финальный `callback`.

Например:

```
var async = require('async');
async.series([
    function(callback) {
        console.log('First Execute..');
        callback(null, 'userPersonalData');
    },
    function(callback) {
        console.log('Second Execute.. ');
        callback(null, 'userDependentData');
    }
```

```
    ],
    function(err, result) {
        console.log(result);
    });
```

Вывод:

```
First Execute..
Second Execute..
['userPersonalData', 'userDependentData'] // результат
```

Глава 4.
Async/Await

Введение

Async/await — это набор ключевых слов, позволяющих писать асинхронный код в процедурной манере, без необходимости полагаться на обратные вызовы или цепочки промисов (`.then().then().then()`).

Это работает за счет использования ключевого слова `await`, которое приостанавливает выполнение асинхронной функции до разрешения промиса, и ключевого слова `async`, которое объявляет такие асинхронные функции, возвращающие промис.

Async/await доступен в Node.js по умолчанию, начиная с версии 8, или в версии 7 при использовании флага `--harmony-async-await`.

Примеры

Асинхронные функции с обработкой ошибок через try-catch

Одной из лучших особенностей синтаксиса async/await является возможность использовать стандартный стиль кодирования с try-catch, как если бы вы писали синхронный код.

```
const myFunc = async (req, res) => {
    try {
        const result = await somePromise();
    } catch (err) {
        // обработка ошибок осуществляется здесь
    }
});
```

Вот пример с использованием Express и promise-mySQL:

```
router.get('/flags/:id', async (req, res) => {
    try {
        const connection = await pool.createConnection();
```

```
        try {
            const sql = `SELECT f.id, f.width, f.height, f.code,
f.filename
                         FROM flags f
                         WHERE f.id = ?
                         LIMIT 1`;
            const flags = await connection.query(sql, req.params.id);
            if (flags.length === 0)
             return res.status(404).send({ message: 'flag not found' });
            return res.send({ flags[0] });
        } finally {
            pool.releaseConnection(connection);
        }
    } catch (err) {
        // обработка ошибок осуществляется здесь
    }
});
```

Сравнение Promises и Async/Await

Функция, использующая промисы:

```
function myAsyncFunction() {
    return aFunctionThatReturnsAPromise()
        // doSomething – это синхронная функция
        .then(result => doSomething(result))
        .catch(handleError);
}
```

Вот как async/await делает код более чистым:

```
async function myAsyncFunction() {
    let result;
    try {
        result = await aFunctionThatReturnsAPromise();
    } catch (error) {
        handleError(error);
    }
    // doSomething – это синхронная функция
    return doSomething(result);
}
```

Ключевое слово async можно сравнить с написанием return new Promise((resolve, reject) => {...}. А await аналогичен получению результата в обратном вызове then.

Переход от обратных вызовов

Вначале мы использовали обратные вызовы, и это было вполне приемлемо:

```
const getTemperature = (callback) => {
    http.get('www.temperature.com/current', (res) => {
        callback(res.data.temperature);
    });
}

const getAirPollution = (callback) => {
    http.get('www.pollution.com/current', (res) => {
        callback(res.data.pollution);
    });
}

getTemperature(function(temp) {
    getAirPollution(function(pollution) {
            console.log(`the temp is ${temp} and the pollution is
${pollution}.`);
        // The temp is 27 and the pollution is 0.5.
    });
});
```

Но возникло несколько серьезных проблем, поэтому мы начали использовать промисы:

```
const getTemperature = () => {
    return new Promise((resolve, reject) => {
        http.get('www.temperature.com/current', (res) => {
            resolve(res.data.temperature);
        });
    });
}

const getAirPollution = () => {
    return new Promise((resolve, reject) => {
        http.get('www.pollution.com/current', (res) => {
            resolve(res.data.pollution);
        });
    });
}

getTemperature()
    .then(temp => console.log(`the temp is ${temp}`))
    .then(() => getAirPollution())
```

```
  .then(pollution => console.log(`and the pollution is ${pollution}`));
// the temp is 32
// and the pollution is 0.5
```

Все стало немного лучше. И, наконец, мы нашли async/await, который по-прежнему использует промисы «под капотом»:

```
const temp = await getTemperature();
const pollution = await getAirPollution();
```

Остановка выполнения на await

Если промис ничего не возвращает, асинхронное задание может быть завершено с помощью await:

```
try {
    await User.findByIdAndUpdate(user._id, {
        $push: {
            tokens: token
        }
    }).exec();
} catch (e) {
    handleError(e);
}
```

Глава 5.
Асинхронное программирование

Введение

Когда мы используем Node, все может выполняться асинхронно. Ниже вы найдете несколько примеров и типичных аспектов работы в асинхронном режиме.

Синтаксис

- `doSomething([args], function([argsCB]) { /* выполнение действия по завершении */ });`
- `doSomething([args], ([argsCB]) => { /* выполнение действия по завершении */ });`

Примеры

Функции обратного вызова в JavaScript

Функции обратного вызова (callback functions) широко распространены в JavaScript, так как функции в этом языке являются объектами первого класса.

Синхронные функции обратного вызова

Функции обратного вызова могут быть синхронными или асинхронными. Поскольку асинхронные функции обратного вызова могут быть более сложными, рассмотрим простой пример синхронной функции обратного вызова.

```
// функция, которая использует функцию обратного вызова с именем `cb`
в качестве параметра
function getSyncMessage(cb) {
```

```
    cb("Hello World!");
}
console.log("Before getSyncMessage call");
// вызов функции и передача функции обратного вызова в качестве аргу-
мента
getSyncMessage(function(message) {
    console.log(message);
});
console.log("After getSyncMessage call");
```

Вывод для приведенного выше кода будет следующим:

```
> Before getSyncMessage call
> Hello World!
> After getSyncMessage call
```

Рассмотрим пошагово, как выполняется этот код. Это больше для тех, кто еще не знаком с концепцией функций обратного вызова; если вы уже понимаете ее, можете пропустить этот абзац. Сначала код анализируется, затем выполняется строка 6, которая выводит в консоль `Before getSyncMessage call`. Далее строка 8 вызывает функцию `getSyncMessage`, передавая анонимную функцию в качестве аргумента для параметра `cb` в функции `getSyncMessage`. Теперь выполнение происходит внутри функции `getSyncMessage` на строке 3, где вызывается функция `cb`, которая только что была передана. Этот вызов передает строку «`Hello World`» в качестве аргумента для параметра `message` в переданной анонимной функции. Выполнение затем переходит на строку 9, где в консоль выводится `Hello World!`. После этого выполнение проходит через процесс выхода из стека вызовов, достигая строки 10, затем строки 4 и, наконец, возвращаясь на строку 11.

Вот что нужно знать о функциях обратного вызова в целом:

- Функция, которую вы передаете в качестве функции обратного вызова, может быть вызвана ноль, один или несколько раз. Все зависит от реализации.
- Функция обратного вызова может быть вызвана синхронно или асинхронно, а может и одновременно синхронно и асинхронно.
- Как и в обычных функциях, имена, которые вы даете параметрам вашей функции, не важны, но порядок важен. Например, на строке 8 параметр `message` мог бы называться `statement`, `msg` или как-нибудь нелепо вроде `jellybean`. Поэтому вам следует знать, какие параметры передаются в вашу функцию обратного вызова, чтобы получить их в правильном порядке с подходящими именами.

Асинхронные функции обратного вызова

Одно из важных замечаний о JavaScript заключается в том, что этот язык по умолчанию синхронный, но в среде (браузер, Node.js и т. д.) существуют API, которые могут сделать его асинхронным.

Перечислим некоторые распространенные вещи в средах JavaScript, которые принимают функции обратного вызова:

- События (Events)
- setTimeout
- setInterval
- fetch API
- Промисы (Promises)

Также любая функция, использующая одну из вышеуказанных функций, может быть обернута в функцию, которая принимает функцию обратного вызова, и эта функция обратного вызова тогда станет асинхронной (хотя оборачивание промисов в функцию, принимающую функцию обратного вызова, скорее всего, будет считаться антипаттерном, так как существуют более предпочтительные способы обработки промисов).

Владея этой информацией, мы можем построить асинхронную функцию, аналогичную синхронной.

```
// функция, которая использует функцию обратного вызова с именем `cb`
в качестве параметра
function getAsyncMessage(cb) {
    setTimeout(function () { cb("Hello World!") }, 1000);
}
console.log("Before getSyncMessage call");
// вызов функции и передача функции обратного вызова в качестве аргу-
мента

getAsyncMessage(function(message) {
    console.log(message);
});

console.log("After getSyncMessage call");
```

В результате в консоль будет выведено:

```
> Before getSyncMessage call
> After getSyncMessage call
// пауза в 1000 мс без вывода
> Hello World!
```

Выполнение переходит к строке 6, выводя `Before getSyncMessage call`. Затем выполнение переходит к строке 8, вызывая `getAsyncMessage` с функцией обратного вызова для параметра `cb`. Далее выполняется строка 3, которая вызывает `setTimeout` с функцией обратного вызова в качестве первого аргумента и числом 300 в качестве второго аргумента. `setTimeout` удерживает эту функцию обратного вызова, чтобы вызвать ее позже, через 1000 миллисекунд, но после установки тайм-аута и до паузы в 1000 миллисекунд он передает выполнение обратно — туда, где оно было прервано, то есть на строку 4, затем на строку 11. После этого следует пауза в 1 секунду, и затем `setTimeout` вызывает свою функцию обратного вызова, что возвращает выполнение к строке 3, где вызывается функция обратного вызова `getAsyncMessage` с аргументом `«Hello World»` для ее параметра `message`, который затем выводится в консоль на строке 9.

Функции обратного вызова (callbacks) в Node.js

Node.js поддерживает асинхронные функции обратного вызова и часто передает два параметра в ваши функции, которые иногда условно называют `err` и `data`. Приведем пример с чтением текстового файла:

```
const fs = require("fs");
fs.readFile("./test.txt", "utf8", function(err, data) {
    if(err) {
        // обработка ошибки
    } else {
        // обработка текста файла, переданного с `data`
    }
});
```

Это пример функции обратного вызова, которая вызывается один раз.

Хорошей практикой является обработка ошибки каким-либо образом, даже если вы просто записываете ее в лог или выбрасываете. Конструкция `else` не обязательна, если вы выбрасываете ошибку или используете `return`, и ее можно удалить для уменьшения отступов, при условии, что вы прекращаете выполнение текущей функции в блоке `if`, делая что-то вроде выбрасывания ошибки или возврата. Хотя `err`, `data` — часто встречающийся паттерн, не всегда ваши функции обратного вызова будут использовать его. Лучше всего обращаться к документации.

Приведем другой пример функции обратного вызова из библиотеки `express` (`express 4.x`):

```
// этот фрагмент кода был на http://expressjs.com/en/4x/api.html
const express = require('express');
const app = express();
// метод app.get принимает URL-маршрут для отслеживания и функцию обратного вызова,
```

```
// которая вызывается каждый раз, когда этот маршрут запрашивается
пользователем.
app.get('/', function(req, res){
    res.send('hello world');
});
app.listen(3000);
```

Этот пример показывает функцию обратного вызова, которая вызывается многократно. Она получает два объекта в качестве параметров, названных здесь `req` и `res`, эти имена соответствуют запросу и ответу соответственно, и они предоставляют способы просмотреть входящий запрос и настроить ответ, который будет отправлен пользователю.

Существует множество способов использования функций обратного вызова для выполнения синхронного и асинхронного кода в JavaScript, и функции обратного вызова широко распространены в этом языке.

Пример кода

Вопрос: каков вывод кода ниже и почему?

```
setTimeout(function() {
    console.log("A");
}, 1000);
setTimeout(function() {
    console.log("B");
}, 0);
getDataFromDatabase(function(err, data) {
    console.log("C");
    setTimeout(function() {
        console.log("D");
    }, 1000);
});
console.log("E");
```

Вывод

Известно точно: `EBAD`. Неизвестно, когда C будет выведено в лог.

Пояснение: компилятор не остановится на `setTimeout` и методах `getDataFromDatabase`. Поэтому первой будет выведена строка E. Функции обратного вызова (первый аргумент `setTimeout`) будут выполнены асинхронно после истечения времени, установленного в `setTimeout`.

Подробнее:

1. E не имеет `setTimeout`.
2. B имеет `setTimeout` на 0 миллисекунд.

3. A имеет `setTimeout` на 1000 миллисекунд.

4. D должна запросить данные из базы данных, и после этого D должна подождать 1000 миллисекунд, поэтому она выполнится после A.

5. C неизвестна, потому что неизвестно, когда данные из базы данных будут запрошены. Это может произойти как до, так и после A.

Обработка ошибок в асинхронном коде

Ошибки всегда должны обрабатываться. Если вы используете синхронное программирование, можно применить `try-catch`. Но это не работает при асинхронном программировании!

Mongoose

```
try {
    setTimeout(function() {
          throw new Error("I'm an uncaught error and will stop the
server!");
    }, 100);
} catch (ex) {
   console.error("This error will not work in an asynchronous situation:
" + ex);
}
```

Асинхронные ошибки будут обрабатываться только внутри функции обратного вызова!

Обработчики событий

В первых версиях Node.js был доступен обработчик событий:

```
process.on("UncaughtException", function(err, data) {
    if (err) {
        // обработка ошибок
    }
});
```

v0.8

Домены

Внутри домена ошибки обрабатываются через эмиттеры событий. При их использовании все ошибки, таймеры и методы обратного вызова регистрируются

только внутри домена. При возникновении ошибки генерируется событие ошибки, и приложение не «падает»:

```
var domain = require("domain");
var d1 = domain.create();
var d2 = domain.create();
d1.run(function() {
    d2.add(setTimeout(function() {
        throw new Error("error on the timer of domain 2");
    }, 0));
});

d1.on("error", function(err) {
    console.log("error at domain 1: " + err);
});
d2.on("error", function(err) {
    console.log("error at domain 2: " + err);
});
```

Callback hell

«Ад обратных вызовов» (callback hell) (также известный как «пирамида судьбы» или «эффект бумеранга») возникает, когда вы вкладываете слишком много функций обратного вызова друг в друга. Вот пример чтения файла (в ES6):

```
const fs = require('fs');
let filename = `${__dirname}/myfile.txt`;

fs.exists(filename, exists => {
    if (exists) {
        fs.stat(filename, (err, stats) => {
            if (err) {
                throw err;
            }
            if (stats.isFile()) {
                fs.readFile(filename, null, (err, data) => {
                    if (err) {
                        throw err;
                    }
                    console.log(data);
                });
            } else {
                throw new Error("This location contains not a file");
            }
        });
```

```
    } else {
        throw new Error("404: file not found");
    }
});
```

Как избежать callback hell

Рекомендуется не вкладывать более двух функций обратного вызова. Это поможет сохранить читаемость кода и упростит его поддержку в будущем. Если вам нужно вложить больше двух функций обратного вызова, попробуйте использовать распределенные события.

Существует также библиотека async, которая помогает управлять обратными вызовами и их выполнением. Она улучшает читаемость кода обратных вызовов и дает вам больше контроля над их потоком, включая возможность выполнять их параллельно или последовательно.

Нативные Promise (Промисы)

v6.0.0

Промис (Promise) — это инструмент для асинхронного программирования. В JavaScript промисы известны своими методами then. У промисов, то есть обещаний (promises), есть два основных состояния — «ожидание» (pending) и «завершено» (settled). Как только промис становится «завершенным», он не может вернуться в «ожидание». Это означает, что промисы в основном хороши для событий, которые происходят только один раз. Состояние «завершено» также делится на два состояния — «разрешено» (resolved) и «отклонено» (rejected). Вы можете создать новый промис, используя ключевое слово new и передав функцию в конструктор new Promise(function(resolve, reject) {}).

Функция, переданная в конструктор Promise, всегда получает первый и второй параметры, обычно называемые resolve и reject соответственно. Названия этих двух параметров условны, но они переводят promise либо в «разрешенное», либо «отклоненное» состояние. Когда вызывается одна из этих функций, promise переходит из состояния «ожидания» в «завершено». resolve вызывается, когда желаемое действие, которое часто является асинхронным, было выполнено, а reject используется, если действие привело к ошибке.

В приведенном ниже примере timeout — это функция, которая возвращает Promise:

```
function timeout (ms) {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            resolve("It was resolved!");
        }, ms)
```

```
    });
}

timeout(1000).then(function (dataFromPromise) {
    // выводит "It was resolved!"
    console.log(dataFromPromise);
})
console.log("waiting...");
```

Вывод в консоль:

```
waiting...
// << пауза на одну секунду >>
It was resolved!
```

Когда вызывается `timeout`, функция, переданная в конструктор `Promise`, выполняется без задержки. Затем выполняется метод `setTimeout`, и его функция обратного вызова запланирована на выполнение через ms миллисекунд, в данном случае `ms=1000`. Поскольку функция обратного вызова `setTimeout` еще не была вызвана, функция `timeout` возвращает управление вызывающему коду. Цепочка методов `then` затем сохраняется для последующего вызова, когда/если `Promise` будет разрешен. Если бы здесь были методы `catch`, они бы также были сохранены, но были бы вызваны, когда/если `Promise` будет отклонен.

Скрипт затем выводит «`waiting...`». Через секунду `setTimeout` вызывает свою функцию обратного вызова, которая вызывает функцию `resolve` со строкой «`It was resolved!`». Эта строка затем передается в функцию обратного вызова метода `then` и выводится пользователю.

Точно так же, как вы можете обернуть асинхронную функцию `setTimeout`, которая требует функции обратного вызова, вы можете обернуть любое отдельное асинхронное действие с помощью `Promise`.

Глава 6.
Автоматическая перезагрузка при изменениях

Примеры

Автоперезагрузка при изменении исходного кода с использованием nodemon

Пакет nodemon позволяет автоматически перезагружать вашу программу при изменении любого файла в исходном коде.

Установка nodemon глобально

```
npm install -g nodemon (или npm i -g nodemon)
```

Установка nodemon локально

```
npm install --save-dev nodemon (или npm i -D nodemon)
```

Использование nodemon

Запустите вашу программу с nodemon:

```
nodemon entry.js (или nodemon entry)
```

Это заменяет обычное использование:

```
node entry.js (или node entry)
```

Вы также можете добавить запуск nodemon как npm-скрипт, что может быть полезно, если вы хотите передавать параметры и не вводить их каждый раз вручную.

Добавьте в package.json:

```
"scripts": {
    "start": "nodemon entry.js -devmode -something 1"
}
```

Таким образом, вы сможете просто использовать команду:

```
npm start
```

из вашей консоли.

Browsersync

Обзор

Browsersync — это инструмент, который позволяет следить за изменениями файлов в реальном времени и перезагружать браузер. Доступен в виде NPM-пакета.

Установка

Чтобы установить Browsersync, вам сначала нужно установить Node.js и NPM. Для получения дополнительной информации см. документацию по установке и запуску Node.js по ссылке https://riptutorial.com/node-js или по QR-коду ниже.

После того как ваш проект настроен, вы можете установить Browsersync с помощью следующей команды:

```
$ npm install browser-sync -D
```

Это установит Browsersync в локальный каталог node_modules и сохранит его в ваших зависимостях для разработчиков.

Если вы предпочитаете установить его глобально, используйте флаг `-g` вместо `-D`.

Пользователи Windows

Если у вас возникли проблемы с установкой `Browsersync` на Windows, вам может потребоваться установить Visual Studio, чтобы получить доступ к инструментам сборки для установки `Browsersync`. Затем вам нужно будет указать версию Visual Studio, которую вы используете, следующим образом:

```
$ npm install browser-sync --msvs_version=2013 -D
```

Эта команда указывает версию Visual Studio именно 2013 года.

Основное использование

Чтобы автоматически перезагружать ваш сайт при изменении файла JavaScript в проекте, используйте следующую команду:

```
$ browser-sync start --proxy "myproject.dev" --files "**/*.js"
```

Замените `myproject.dev` на веб-адрес, который вы используете для доступа к вашему проекту. `Browsersync` выдаст альтернативный адрес, который можно использовать для доступа к вашему сайту через прокси.

Продвинутое использование

Помимо интерфейса командной строки, описанного выше, `Browsersync` также может использоваться с `Grunt.js` и `Gulp.js`.

Grunt.js

Для использования с `Grunt.js` требуется плагин, который можно установить следующим образом:

```
$ npm install grunt-browser-sync -D
```

Затем добавьте эту строку в ваш `gruntfile.js`:

```
grunt.loadNpmTasks('grunt-browser-sync');
```

Gulp.js

`Browsersync` работает как модуль `CommonJS`, поэтому нет необходимости в плагине для `Gulp.js`. Просто подключите модуль следующим образом:

```
var browserSync = require('browser-sync').create();
```

Теперь вы можете использовать API `Browsersync` для настройки в соответствии с вашими потребностями.

API

API Browsersync можно найти здесь: https://browsersync.io/docs/api или по QR-коду ниже.

Глава 7.
Избегайте ада обратных вызовов (callback hell)

Примеры

Модуль Async

Исходный код доступен для загрузки с GitHub по ссылке https://github.com/caolan/async или по QR-коду ниже.

Кроме того, вы можете установить его с помощью NPM:

```
$ npm install --save async
```

...а также с помощью Bower:

```
$ bower install async
```

Mongoose

```
var async = require("async");
async.parallel([
    function(callback) { ... },
    function(callback) { ... }
```

```
], function(err, results) {
    // необязательная функция обратного вызова
});
```

Еще о модуле Async

Библиотеки, такие как Async.js, очень полезны. Async добавляет тонкий слой функций поверх вашего кода, что может значительно снизить сложность, избегая вложенности функций обратного вызова. В Async есть множество вспомогательных методов, которые можно использовать в различных ситуациях, таких как `series`, `parallel`, `waterfall` и другие. Каждая функция имеет свое конкретное применение, поэтому уделите время изучению того, какая функция поможет в той или иной ситуации.

Как бы ни был хорош Async, он не идеален. Очень легко увлечься, комбинируя `series`, `parallel`, `forever` и другие методы, и в итоге вернуться к исходной точке с неаккуратным кодом. Будьте осторожны и не занимайтесь преждевременной оптимизацией. То, что несколько асинхронных задач можно выполнить параллельно, не всегда означает, что это нужно делать. В реальности, поскольку Node.js однопоточен, выполнение задач параллельно с использованием Async дает незначительное или вообще никакого улучшения производительности.

Пример использования Async waterfall

```
var fs = require('fs');
var async = require('async');
var myFile = '/tmp/test';
async.waterfall([
    function(callback) {
        fs.readFile(myFile, 'utf8', callback);
    },
    function(txt, callback) {
        txt = txt + '\nAppended something!';
        fs.writeFile(myFile, txt, callback);
    }
], function (err, result) {
    if(err) return console.log(err);
    console.log('Appended text!');
});
```

Глава 8.
Bluebird Promises (промисы)

Примеры

Преобразование библиотеки nodeback в Promises

```
const Promise = require('bluebird'),
    fs = require('fs')
Promise.promisifyAll(fs)
// теперь вы можете использовать методы на основе promise для 'fs'
с суффиксом Async
fs.readFileAsync('file.txt').then(contents => {
    console.log(contents)
}).catch(err => {
    console.error('error reading', err)
})
```

Функциональные Promises

Пример использования `map`:

```
Promise.resolve([ 1, 2, 3 ]).map(el => {

    return Promise.resolve(el * el) // в реальной ситуации возвращает-
ся некоторая асинхронная операция
})
```

Пример использования `filter`:

```
Promise.resolve([ 1, 2, 3 ]).filter(el => {

    return Promise.resolve(el % 2 === 0)
// в реальной ситуации возвращается некоторая асинхронная операция
}).then(console.log)
```

Пример использования reduce:

```
Promise.resolve([ 1, 2, 3 ]).reduce((prev, curr) => {

    return Promise.resolve(prev + curr) // в реальной ситуации возвра-
щается некоторая асинхронная операция
}).then(console.log)
```

Корутины (coroutines)

```
const promiseReturningFunction = Promise.coroutine(function* (file) {
    const data = yield fs.readFileAsync(file) // это возвращает Promise
и разрешается в содержимое файла
    return data.toString().toUpperCase()
})
promiseReturningFunction('file.txt').then(console.log)
```

Автоматическое освобождение ресурсов (Promise.using)

```
function somethingThatReturnsADisposableResource() {
    return getSomeResourceAsync(...).disposer(resource => {
        resource.dispose()
    })
}
Promise.using(somethingThatReturnsADisposableResource(), resource =>
{
     // используйте ресурс здесь, disposer автоматически закроет его
при выходе из Promise.using
})
```

Последовательное выполнение

```
Promise.resolve([1, 2, 3])
    .mapSeries(el => Promise.resolve(el * el)) // в реальном примене-
нии используйте асинхронную функцию, возвращающую Promise.
    .then(console.log)
```

Глава 9.
Преобразование функций обратного вызова в Promises (промисах)

Примеры

Промисификация функций обратного вызова

Использование на основе функций обратного вызова:

```
db.notification.email.find({subject: 'promisify callback'}, (error,
result) => {
    if (error) {
        console.log(error);
    }
    // обычный код здесь
});
```

Этот код использует метод `promisifyAll` из библиотеки `bluebird` для преобразования кода, основанного на функциях обратного вызова, как в приведенном выше примере. `bluebird` создаст версию с `promise` для всех методов в объекте, и имена этих методов на основе `promise` будут иметь суффикс `Async`:

```
let email = bluebird.promisifyAll(db.notification.email);

email.findAsync({subject: 'promisify callback'}).then(result => {
    // обычный код здесь
})
.catch(console.error);
```

Если необходимо промисифицировать только определенные методы, используйте `promisify`:

```
let find = bluebird.promisify(db.notification.email.find);

find({locationId: 168}).then(result => {
```

```
    // обычный код здесь
})
.catch(console.error);
```

Существуют некоторые библиотеки (например MassiveJS), которые нельзя промисифицировать, если объект метода не передан во втором параметре. В таком случае просто передайте объект метода, который нужно промисифицировать, во втором параметре и заключите его в свойство context:

```
let find = bluebird.promisify(db.notification.email.find, { context:
db.notification.email });

find({locationId: 168}).then(result => {
    // обычный код здесь
})
.catch(console.error);
```

Ручная промисификация функции обратного вызова

Иногда может потребоваться вручную промисифицировать функцию обратного вызова. Это может быть необходимо в случае, когда она не следует стандартному формату «функция обратного вызова с первым аргументом-ошибкой» (error-first callback) или если требуется дополнительная логика для промисификации.

Пример с fs.exists(path, callback):

```
var fs = require('fs');

var existsAsync = function(path) {
    return new Promise(function(resolve, reject) {
        fs.exists(path, function(exists) {
            // exists – это булево значение

            if (exists) {
                // Успешное разрешение
                resolve();
            } else {
                // Отклонение с ошибкой
                reject(new Error('path does not exist'));
            }
        });
    });
};
// Теперь используйте как promise
existsAsync('/path/to/some/file').then(function() {
```

```
    console.log('file exists!');
}).catch(function(err) {
    // файл не существует
    console.error(err);
});
```

Промисификация setTimeout

```
function wait(ms) {
    return new Promise(function (resolve, reject) {
        setTimeout(resolve, ms)
    })
}
```

Глава 10.
Интеграция с Cassandra

Примеры

Hello world

Для доступа к Cassandra можно использовать модуль `cassandra-driver` от DataStax. Он поддерживает все функции и легко настраивается.

Модуль `cassandra-driver` доступен на GitHub по ссылке https://github.com/datastax/nodejs-driver.

Также его можно скачать по QR-коду ниже.

```
const cassandra = require("cassandra-driver");
const clientOptions = {
    contactPoints: ["host1", "host2"],
    keyspace: "test"
};

const client = new cassandra.Client(clientOptions);

const query = "SELECT hello FROM world WHERE name = ?";
```

```
client.execute(query, ["John"], (err, results) => {
    if (err) {
        // Вывод ошибки в консоль
        return console.error(err);
    }

    // Вывод строк результата в консоль
    console.log(results.rows);
});
```

Глава 11.
Интерфейс командной строки (CLI)

Синтаксис

- `node [options] [v8 options] [script.js | -e "script"] [arguments]`

Примеры

Опции командной строки

`-v, --version`

Добавлено в v0.1.3
Вывод версии Node.js.

`-h, --help`

Добавлено в v0.1.3
Вывод опций командной строки Node.js. Вывод данной опции менее детализирован, чем этот документ.

`-e, --eval "script"`

Добавлено в v0.5.2
Выполняет следующий аргумент как JavaScript. Модули, которые предопределены в REPL, также могут быть использованы в скрипте.

`-p, --print "script"`

Добавлено в v0.6.4
Идентично `-e`, но выводит результат.

`-c, --check`

Добавлено в v5.0.0
Проверяет синтаксис скрипта без выполнения.

`-i, --interactive`

Добавлено в v0.7.7
Открывает REPL, даже если `stdin` не выглядит как терминал.

`-r, --require module`

Добавлено в v1.6.0
Предварительная загрузка указанного модуля при запуске.
Следует правилам разрешения модулей `require()`. Модуль может быть либо путем к файлу, либо именем модуля Node.js.

`--no-deprecation`

Добавлено в v0.8.0
Отключение предупреждений об устаревших (депрецированных) функциях.

`--trace-deprecation`

Добавлено в v0.8.0
Вывод трассировок стека для предупреждений об устаревших функциях.

`--throw-deprecation`

Добавлено в v0.11.14
Вызов ошибок для устаревших функций.

`--no-warnings`

Добавлено в v6.0.0
Отключение всех предупреждений обработки (в том числе об устаревших функциях).

`--trace-warnings`

Добавлено в v6.0.0
Вывод трассировок стека для предупреждений обработки (в том числе об устаревших функциях).

`--trace-sync-io`

Добавлено в v2.1.0
Выводит трассировку стека при обнаружении синхронного ввода-вывода после первого выполнения цикла событий.

`--zero-fill-buffers`

Добавлено в v6.0.0
Автоматическое заполнение нулями всех вновь выделенных экземпляров `Buffer` и `SlowBuffer`.

`--preserve-symlinks`

Добавлено в v6.3.0

Указывает загрузчику модулей сохранять символические ссылки при разрешении и кешировании модулей.

По умолчанию, когда Node.js загружает модуль из пути, который символически ссылается на другое местоположение на диске, Node.js будет разыменовывать ссылку и использовать фактический «настоящий путь» модуля на диске как идентификатор и как корневой путь для поиска других зависимых модулей. В большинстве случаев такое поведение по умолчанию является приемлемым. Однако при использовании символически связанных зависимостей, как показано в примере ниже, поведение по умолчанию вызывает исключение, если `moduleA` пытается подключить `moduleB` как зависимость:

```
{appDir}
├── app
|  ├── index.js
|  └── node_modules
|    ├── moduleA -> {appDir}/moduleA
|    └── moduleB
|      ├── index.js
|      └── package.json
└── moduleA
  ├── index.js
  └── package.json
```

Флаг командной строки `--preserve-symlinks` указывает Node.js использовать путь символической ссылки для модулей вместо реального пути, что позволяет найти символически связанные зависимости.

Однако использование `--preserve-symlinks` может иметь побочные эффекты. В частности, символически связанные нативные модули могут не загрузиться, если они связаны из более чем одного места в дереве зависимостей (Node.js воспримет их как два отдельных модуля и попытается загрузить модуль несколько раз, что вызовет исключение):

`--track-heap-objects`

Добавлено в v2.4.0

Отслеживание выделений объектов кучи (heap) для снимков кучи.

`--prof-process`

Добавлено в v6.0.0

Обработка выходных данных профилировщика v8, сгенерированных с использованием опции v8 `--prof`.

`--v8-options`

Добавлено в v0.1.3
Вывод опций командной строки v8.
Примечание: опции v8 допускают разделение слов как дефисами (-), так и подчеркиваниями (_). Например, `--stack-trace-limit` эквивалентен `--stack_trace_limit`.

`--tls-cipher-list=list`

Добавлено в v4.0.0
Указание альтернативного списка шифров TLS по умолчанию (требует сборки Node.js с поддержкой криптографии (по умолчанию)).

`--enable-fips`

Добавлено в v6.0.0
Включение криптографии, соответствующей стандарту FIPS, при запуске (требует сборки Node.js с `./configure --openssl-fips`).

`--force-fips`

Добавлено в v6.0.0
Принудительное включение криптографии, соответствующей стандарту FIPS, при запуске (невозможно отключить из скрипта). (Те же требования, что и для `--enable-fips`).

`--icu-data-dir=file`

Добавлено в v0.11.15
Указание пути загрузки данных ICU (переопределяет NODE_ICU_DATA).

Переменные окружения

`NODE_DEBUG=module[,…]`

Добавлено в v0.1.32
‘,’-разделенный список основных модулей, которые должны выводить отладочную информацию.

`NODE_PATH=path[:…]`

Добавлено в v0.1.32
‘:’-разделенный список директорий (папок), которые добавляются к пути поиска модулей.
Примечание: в Windows это ‘;’-разделенный список.

`NODE_DISABLE_COLORS=1`

Добавлено в v0.3.0
Когда установлено в 1, цвета не будут использоваться в REPL.

`NODE_ICU_DATA=file`

Добавлено в v0.11.15

Путь к данным ICU (объект Intl). Расширяет встроенные данные при компиляции с поддержкой малого ICU:

`NODE_REPL_HISTORY=file`

Добавлено в v5.0.0

Путь к файлу, используемому для сохранения истории REPL. Путь по умолчанию — `~/.node_repl_history`, который может быть переопределен этой переменной. Установка значения в пустую строку ("" или " ") отключает постоянную историю REPL.

Глава 12.
Взаимодействие «клиент — сервер»

Примеры

Использование с Express, jQuery and Jade

```
//client.jade

// Кнопка размещена внизу; аналогично HTML
button(type='button', id='send_by_button') Modify data

    #modify Lorem ipsum Sender

    // Подключение jQuery; также можно сделать это из онлайн-источника
    script(src='./js/jquery-2.2.0.min.js')

    // AJAX-запрос с использованием jQuery
    script
        $(function () {
            $('#send_by_button').click(function (e) {
                e.preventDefault();
                 // тест: текст в скобках должен появляться при клике
на кнопку
                //window.alert('You clicked on me. - jQuery');
                // переменная и JSON инициализированы в коде
                var predeclared = "Katamori";
                var data = {
                    Title: "Name_SenderTest",
                    Nick: predeclared,
                    FirstName: "Zoltan",
                    Surname: "Schmidt"
                };

                // AJAX-запрос с указанными параметрами
```

```
                $.ajax({
                    type: 'POST',
                    data: JSON.stringify(data),
                    contentType: 'application/json',
                    url: 'http://localhost:7776/domaintest',
                     // при успешном выполнении полученные данные ис-
пользуются в качестве входных данных функции 'data'
                    success: function (data) {
                        window.alert('Request sent; data received.');

                        var jsonstr = JSON.stringify(data);
                        var jsonobj = JSON.parse(jsonstr);

                       // если элемент JSON 'Nick' не равен предопре-
деленной строке (как было инициализировано), то
                        // скрипт backend был выполнен, что означает,
что связь была установлена
                        if(data.Nick != predeclared){
                          document.getElementById("modify").innerHTML
= "JSON changed!\n" +
                                  jsonstr;
                        };
                    }
                });
            });
        });

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

// Маршрут Express, прослушивающий GET-запросы – в данном случае
он пуст, а это означает, что ничего не отображается при обращении
к 'localhost/domaintest'
router.get('/', function(req, res, next) {
});

// То же самое для POST-запросов – обратите внимание, что AJAX-запрос
выше был определен как POST
router.post('/', function(req, res) {
    res.setHeader('Content-Type', 'application/json');

    // сгенерированное содержимое
    var some_json = {
        Title: "Test",
```

```
        Item: "Crate"
    };
    var result = JSON.stringify(some_json);
    // содержимое, переданное 'client.jade'
    var sent_data = req.body;
    sent_data.Nick = "ttony33";
    res.send(sent_data);

});

module.exports = router;
```

Глава 13.
Модуль Cluster

Синтаксис

- `const cluster = require("cluster")`
- `cluster.fork()`
- `cluster.isMaster`
- `cluster.isWorker`
- `cluster.schedulingPolicy`
- `cluster.setupMaster(settings)`
- `cluster.settings`
- `cluster.worker // в worker`
- `cluster.workers // в master`

Примечания

Обратите внимание, что `cluster.fork()` порождает дочерний процесс, который начинает выполнение текущего скрипта с начала, в отличие от системного вызова `fork()` в языке C, который клонирует текущий процесс и продолжает выполнение с инструкции после системного вызова как в родительском, так и в дочернем процессе.

Более полное руководство по кластерам можно найти в документации Node.js по ссылке https://nodejs.org/api/cluster.html или по QR-коду справа.

Примеры

Hello World

Это ваш `cluster.js`:

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;
if (cluster.isMaster) {
    // Порождаем рабочие процессы.
    for (let i = 0; i < numCPUs; i++) {
        cluster.fork();
    }
    cluster.on('exit', (worker, code, signal) => {
        console.log(`worker ${worker.process.pid} died`);
    });
} else {
    // Рабочие процессы могут использовать любое TCP-соединение
    // В данном случае это HTTP-сервер
    require('./server.js')();
}
```

Это ваш основной `server.js`:

```
const http = require('http');
function startServer() {
    const server = http.createServer((req, res) => {
        res.writeHead(200);
        res.end('Hello Http');
    });
    server.listen(3000);
}
if (!module.parent) {
    // Запускаем сервер, если файл запущен напрямую
    startServer();
} else {
    // Экспортируем сервер, если файл подключается через cluster
    module.exports = startServer;
}
```

В этом примере мы хостим базовый веб-сервер, однако запускаем рабочие (дочерние) процессы с использованием встроенного модуля `cluster`. Количество порожденных процессов зависит от количества доступных ядер процессора. Это позволяет приложению Node.js использовать многопроцессорные системы, поскольку один экземпляр Node.js работает в одном потоке. Приложение

будет использовать порт 8000 одновременно для всех процессов. Нагрузка автоматически распределяется между рабочими процессами по методу Round-Robin по умолчанию.

Пример кластера

Один экземпляр Node.js работает в одном потоке. Чтобы использовать преимущества многопроцессорных систем, приложение может быть запущено в кластере процессов Node.js для обработки нагрузки.

Модуль `cluster` позволяет легко создавать дочерние процессы, которые все разделяют серверные порты.

Следующий пример создает дочерний рабочий процесс в основном процессе, который распределяет нагрузку между несколькими ядрами.

Пример

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length; // количество CPU

if (cluster.isMaster) {
    // Порождаем рабочие процессы.
    for (var i = 0; i < numCPUs; i++) {
        cluster.fork(); // создание дочернего процесса
    }
    // при завершении работы кластера
    cluster.on('exit', (worker, code, signal) => {
        if (signal) {
            console.log(`worker was killed by signal: ${signal}`);
        } else if (code !== 0) {
            console.log(`worker exited with error code: ${code}`);
        } else {
            console.log('worker success!');
        }
    });
} else {
    // Рабочие процессы могут использовать любое TCP-соединение
    // В данном случае это HTTP-сервер
    http.createServer((req, res) => {
        res.writeHead(200);
        res.end('hello world\n');
    }).listen(3000);
}
```

Глава 14.
Подключение к MongoDB

Введение

MongoDB — это бесплатная кросс-платформенная документно-ориентированная база данных с открытым исходным кодом. Классифицируемая как база данных NoSQL, MongoDB использует документы, похожие на JSON, со схемами.

Для получения более подробной информации перейдите на официальный сайт MongoDB по ссылке https://www.mongodb.com/ или по QR-коду ниже.

Синтаксис

```
MongoClient.connect('mongodb://127.0.0.1:27017/crud', function (err,
db) {
    // действия здесь
});
```

Примеры

Простой пример подключения к MongoDB из Node.js

```
MongoClient.connect('mongodb://localhost:27017/myNewDB', function
(err, db) {
```

```
    if (err)
        console.log("Unable to connect DB. Error: " + err)
    else
        console.log('Connected to DB');
    db.close();
});
```

myNewDB — это имя базы данных. Если она не существует, то будет автоматически создана этим вызовом.

Простой способ подключения к MongoDB с использованием чистого Node.js

```
var MongoClient = require('mongodb').MongoClient;
// подключение к MongoDB
MongoClient.connect("mongodb://localhost:27017/MyDb", function (err,
db) {
    // проверка соединения
    if (err) {
        console.log("connection failed.");
    } else {
        console.log("successfully connected to mongoDB.");
    }
});
```

Глава 15.
Создание библиотеки Node.js, поддерживающей как Promises, так и функции обратного вызова с первым аргументом-ошибкой

Введение

Многим нравится работать с синтаксисом promise и/или async/await, но при написании модуля полезно поддерживать также классический стиль методов с обратными вызовами (callback). Вместо создания двух модулей или двух наборов функций, либо использования promisify, ваш модуль может поддерживать оба способа программирования, используя asCallback() из библиотеки Bluebird или nodeify() из библиотеки Q.

Примеры

Пример модуля и соответствующей программы с использованием Bluebird

math.js:

```
'use strict';
const Promise = require('bluebird');
module.exports = {
    // пример метода, работающего только с обратными вызовами (callback)
    callbackSum: function(a, b, callback) {
        if (typeof a !== 'number')
            return callback(new Error('"a" must be a number'));
        if (typeof b !== 'number')
            return callback(new Error('"b" must be a number'));
        return callback(null, a + b);
    },
```

```
    // пример метода, работающего только с промисами (promises)
    promiseSum: function(a, b) {
        return new Promise(function(resolve, reject) {
            if (typeof a !== 'number')
                return reject(new Error('"a" must be a number'));
            if (typeof b !== 'number')
                return reject(new Error('"b" must be a number'));
            resolve(a + b);
        });
    },
    // метод, который можно использовать как с промисами (promises),
так и с обратными вызовами (callback)
    sum: function(a, b, callback) {
        return new Promise(function(resolve, reject) {
            if (typeof a !== 'number')
                return reject(new Error('"a" must be a number'));
            if (typeof b !== 'number')
                return reject(new Error('"b" must be a number'));
            resolve(a + b);
        }).asCallback(callback);
    },
};
```

index.js:

```
'use strict';
const math = require('./math');

// классические обратные вызовы (callbacks)
math.callbackSum(1, 3, function(err, result) {
    if (err)
        console.log('Test 1: ' + err);
    else
        console.log('Test 1: the answer is ' + result);
});

math.callbackSum(1, 'd', function(err, result) {
    if (err)
        console.log('Test 2: ' + err);
    else
        console.log('Test 2: the answer is ' + result);
});

// промисы (promises)
math.promiseSum(2, 5)
```

```
.then(function(result) {
    console.log('Test 3: the answer is ' + result);
})
.catch(function(err) {
    console.log('Test 3: ' + err);
});

math.promiseSum(1)
.then(function(result) {
    console.log('Test 4: the answer is ' + result);
})
.catch(function(err) {
    console.log('Test 4: ' + err);
});

// метод, использующийся как промис (promise)
math.sum(8, 2)
.then(function(result) {
    console.log('Test 5: the answer is ' + result);
})
.catch(function(err) {
    console.log('Test 5: ' + err);
});

// метод, использующийся с обратными вызовами (callback)
math.sum(7, 11, function(err, result) {
    if (err)
        console.log('Test 6: ' + err);
    else
        console.log('Test 6: the answer is ' + result);
});

// метод, использующийся как промис (promise) с синтаксисом async/await
(async () => {
    try {
        let x = await math.sum(6, 3);
        console.log('Test 7a: ' + x);
        let y = await math.sum(4, 's');
        console.log('Test 7b: ' + y);
    } catch(err) {
        console.log(err.message);
    }
})();
```

Глава 16.
Создание API с помощью Node.js

Примеры

GET API с использованием Express

API на Node.js можно легко создавать с помощью веб-фреймворка Express. Следующий пример создает простой GET API для вывода списка всех пользователей.

Пример

```
var express = require('express');
var app = express();
var users = [{
    id: 1,
    name: "John Doe",
    age : 23,
    email: "john@doe.com"
}];

// GET /api/users
app.get('/api/users', function(req, res){
    return res.json(users); // вернуть ответ в формате JSON
});

app.listen('3000', function(){
    console.log('Server listening on port 3000');
});
```

POST API с использованием Express

Следующий пример создает POST API с использованием Express. Этот пример аналогичен примеру GET, за исключением использования `body-parser`, который разбирает данные POST-запроса и добавляет их в `req.body`.

Пример

```
var express = require('express');
var app = express();
// для разбора тела запроса в POST-запросах
var bodyParser = require('body-parser');
var users = [{
    id: 1,
    name: "John Doe",
    age: 23,
    email: "john@doe.com"
}];

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

// GET /api/users
app.get('/api/users', function(req, res) {
    return res.json(users); // вернуть список пользователей в форма-
те JSON
});

/* POST /api/users
{
    "user": {
        "id": 3,
        "name": "Test User",
        "age": 20,
        "email": "test@test.com"
    }
}
*/
app.post('/api/users', function(req, res) {
    var user = req.body.user;
    users.push(user); // добавить пользователя в список
   return res.send('User has been added successfully'); // вернуть со-
общение об успешном добавлении
});

app.listen('3000', function() {
    console.log('Server listening on port 3000'); // сообщение о запу-
ске сервера на порту 3000
});
```

Глава 17.
Парсер CSV в Node.js

Введение

Чтение данных из CSV может быть выполнено различными способами. Одно из решений — это считать CSV-файл в массив. После этого можно работать с массивом.

Примеры

Использование FS для чтения CSV

`fs` — это API файловой системы в Node.js. Мы можем использовать метод `readFile` на нашей переменной `fs`, передав ему файл `data.csv`, формат и функцию, которые читают и разбивают CSV для дальнейшей обработки.

Это предполагает наличие файла data.csv в той же папке.

```
'use strict'
const fs = require('fs');
fs.readFile('data.csv', 'utf8', function (err, data) {
    var dataArray = data.split(/\r?\n/);
    console.log(dataArray);
});
```

Теперь вы можете использовать этот массив как любой другой.

Глава 18.
Работа с базой данных (MongoDB с Mongoose)

Примеры

Подключение Mongoose

Убедитесь, что MongoDB запущен: `mongod --dbpath data/.`

package.json

```
"dependencies": {
    "mongoose": "^4.5.5"
}
```

server.js (ECMA 6)

```
import mongoose from 'mongoose';
mongoose.connect('mongodb://localhost:27017/stackoverflow-example');
const db = mongoose.connection;
db.on('error', console.error.bind(console, 'DB connection error!'));
// обработка ошибки подключения к БД
```

server.js (ECMA 5.1)

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/stackoverflow-example');
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'DB connection error!'));
// обработка ошибки подключения к БД
```

Модель

Определите свою модель (или модели):

app/models/user.js (ECMA 6)

```
import mongoose from 'mongoose';
const userSchema = new mongoose.Schema({
```

```
    name: String,
    password: String
});
const User = mongoose.model('User', userSchema);
export default User;
```

app/model/user.js (ECMA 5.1)

```
var mongoose = require('mongoose');
var userSchema = new mongoose.Schema({
    name: String,
    password: String
});
var User = mongoose.model('User', userSchema);
module.exports = User;
```

Вставка данных

ECMA 6:

```
const user = new User({
    name: 'Stack',
    password: 'Overflow'
});
user.save((err) => {
    if (err) throw err;
    console.log('User saved!'); // сообщение о сохранении пользователя
});
```

ECMA 5.1:

```
var user = new User({
    name: 'Stack',
    password: 'Overflow'
});
user.save(function (err) {
    if (err) throw err;
    console.log('User saved!');
});
```

Чтение данных

ECMA 6:

```
User.findOne({
    name: 'stack'
```

```
}, (err, user) => {
    if (err) throw err;
    if (!user) {
        console.log('No user was found');
    } else {
        console.log('User was found');
    }
});
```

ECMA 5.1:

```
User.findOne({
    name: 'stack'
}, function (err, user) {
    if (err) throw err;
    if (!user) {
        console.log('No user was found');
    } else {
        console.log('User was found');
    }
});
```

Глава 19.
Отладка Node.js-приложения

Встроенный отладчик Node.js и node inspector

Использование встроенного отладчика

Node.js предоставляет встроенную неграфическую утилиту для отладки. Чтобы запустить встроенный отладчик, запустите приложение с этой командой:

```
node debug filename.js
```

Рассмотрим следующий простой Node.js-скрипт, содержащийся в файле `debugDemo.js`:

```
'use strict';
function addTwoNumber(a, b){
// функция возвращает сумму двух чисел
debugger
    return a + b;
}
var result = addTwoNumber(5, 9);
console.log(result);
```

Ключевое слово `debugger` остановит выполнение кода в отладчике на этой строке.

Команды отладки

1. Шаги по коду

```
cont, c – Продолжить выполнение
next, n – Перейти на следующую строку
step, s – Войти в функцию
out, o – Выйти из функции
```

2. Точки останова

`setBreakpoint(), sb()` – Установить точку останова на текущей строке
`setBreakpoint(line), sb(line)` – Установить точку останова на определенной строке

Чтобы отладить код, выполните следующую команду:

```
node debug debugDemo.js
```

После запуска команды вы увидите следующий вывод. Чтобы выйти из интерфейса отладчика, введите `process.exit()`.

```
ankuranand:~/workspace/nodejs/nodejsDebugging $ node debug debugDemo.js
< Debugger listening on port 5858
debug> . ok
break in debugDemo.js:3
  1 // A Demo Code Showing the basic capabilities of the nodejs  debugging module
  2
> 3 'use strict';
  4
  5 function addTwoNumber(a, b){
debug> n
break in debugDemo.js:11
  9 }
 10
>11 let result = addTwoNumber(5, 9);
 12 console.log(result);
 13
debug> c
break in debugDemo.js:7
  5 function addTwoNumber(a, b){
  6 // function returns the sum of the two numbers
> 7 debugger
  8   return a + b;
  9 }
debug> c
< 14
debug> process.exit()
ankuranand:~/workspace/nodejs/nodejsDebugging $
```

Используйте команду `watch(expression)` для добавления переменной или выражения, за значением которых вы хотите наблюдать. Команда `restart` перезапускает приложение и отладку.

Используйте команду `repl` для интерактивного ввода кода. Режим `repl` имеет такой же контекст, как и строка, на которой вы находитесь в процессе отладки. Это позволяет вам исследовать содержимое переменных и тестировать строки кода. Нажмите Ctrl+C, чтобы выйти из режима `repl`.

Использование встроенного инспектора Node

v6.3.0

Вы можете использовать встроенный v8 инспектор Node.js. Плагин `node-inspector` больше не нужен. Просто передайте флаг инспектора, и вам будет предоставлен URL для доступа к нему:

```
node --inspect server.js
```

Установите `node-inspector`:

```
npm install -g node-inspector
```

Запустите ваше приложение с командой `node-debug`:

```
node-debug filename.js
```

После этого откройте в Chrome:

```
http://localhost:8080/debug?port=5858
```

Иногда порт 8080 может быть недоступен на вашем компьютере. В этом случае может появиться следующая ошибка:

```
Cannot start the server at 0.0.0.0:8080. Error: listen EACCES.
```

В этом случае запустите инспектор Node на другом порту, используя следующую команду:

```
$node-inspector --web-port=6500
```

Вы увидите что-то подобное:

Глава 20.
Доставка HTML или других типов файлов

Синтаксис

- `response.sendFile(fileName, options, function (err) {});`

Примеры

Доставка HTML по указанному пути

Рассмотрим, как создать сервер на Express и обслуживать `index.html` по умолчанию (пустой путь /) и `page1.html` по пути `/page1`.

Структура папок

```
project root
| server.js
|____views
     | index.html
     | page1.html
```

server.js

```
var express = require('express');
var path = require('path');
var app = express();

// доставка index.html, если файл не запрашивается
app.get("/", function (request, response) {
    response.sendFile(path.join(__dirname, 'views/index.html'));
});

// доставка page1.html, если запрашивается page1
```

```
app.get('/page1', function(request, response) {
       response.sendFile(path.join(__dirname, 'views', 'page1.html',
function(error) {
        if (error) {
            // что-то делаем в случае ошибки
            console.log(err);
            response.end(JSON.stringify({error:"page not found"}));
        }
    }));
});

app.listen(8080);
```

Обратите внимание, что sendFile() просто передает статический файл в качестве ответа, не предоставляя возможности его изменить. Если вы обслуживаете HTML-файл и хотите включить в него динамические данные, вам нужно будет использовать шаблонизатор (template engine), такой как Pug, Mustache или EJS.

Глава 21.
Внедрение зависимостей

Почему следует использовать внедрение зависимостей

Быстрый процесс разработки

При использовании внедрения зависимостей разработчики Node.js могут ускорить процесс разработки, так как после внедрения зависимости становится меньше конфликтов кода и проще управлять всеми модулями.

Ослабление связей между модулями

Модули становятся менее связанными, что упрощает их поддержку.

Написание модульных тестов

Жестко заданные зависимости можно передать в модуль, что упрощает написание модульных тестов для каждого модуля.

Глава 22.
Развертывание приложения Node.js без простоя

Примеры

Развертывание с использованием PM2 без простоя

ecosystem.json

```
{
    "name": "app-name",
    "script": "server",
    "exec_mode": "cluster",
    "instances": 0,
    "wait_ready": true,
    "listen_timeout": 10000,
    "kill_timeout": 5000
}
```

`wait_ready`

Вместо ожидания события прослушивания (listen event) ждет, пока не будет выполнено `process.send('ready');`.

`listen_timeout`

Время в миллисекундах перед принудительной перезагрузкой, если приложение не начало слушать.

`kill_timeout`

Время в миллисекундах перед отправкой финального сигнала `SIGKILL`.

server.js

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

```
const app = express();
const server = http.Server(app);
const port = 80;

server.listen(port, function() {
    process.send('ready');
});

process.on('SIGINT', function() {
    server.close(function() {
        process.exit(0);
    });
});
```

Возможно, вам нужно будет подождать, пока ваше приложение не установит соединения с базами данных, кешами, рабочими процессами и т. д. PM2 нужно подождать, прежде чем считать ваше приложение онлайн. Для этого вам нужно указать `wait_ready: true` в файле конфигурации процесса. Это заставит PM2 ждать этого события. В приложении вам нужно будет добавить `process.send('ready');`, когда вы захотите, чтобы оно считалось готовым.

Когда процесс останавливается или перезапускается PM2, вашему процессу отправляются некоторые системные сигналы в заданном порядке. Сначала отправляется сигнал `SIGINT`, который можно поймать, чтобы узнать, что ваш процесс собирается остановиться. Если ваше приложение не завершится само по себе до 1.6 секунды (кастомизируемо), оно получит сигнал `SIGKILL`, чтобы принудительно завершить процесс. Таким образом, если вашему приложению нужно что-то очистить (состояния или задания), вы можете поймать сигнал `SIGINT`, чтобы подготовить его к выходу.

Глава 23. Развертывание приложений Node.js в продакшене

Примеры

Установка NODE_ENV="production"

Развертывание в продакшене может значительно варьироваться, но стандартной практикой являются определение переменной окружения `NODE_ENV` и установка ее значения на `"production"`.

Флаги выполнения

Любой код, выполняемый в вашем приложении (включая внешние модули), может проверять значение переменной `NODE_ENV`:

```
if (process.env.NODE_ENV === 'production') {
    // Запускаем приложение в продакшене
} else {
    // Запускаем приложение в режиме разработки
}
```

Зависимости

Когда переменная окружения `NODE_ENV` установлена на `'production'`, все `devDependencies` в вашем файле `package.json` будут полностью игнорироваться при выполнении команды `npm install`. Вы также можете принудительно задать это с помощью флага `--production`:

```
npm install -production
```

Методы установки NODE_ENV

Метод 1. Установка NODE_ENV для всех Node.js-приложений

Windows:

```
set NODE_ENV=production
```

Linux или другая Unix-подобная система:

```
export NODE_ENV=production
```

Это установит NODE_ENV для текущей bash-сессии, так что любые приложения, запущенные после этого, будут иметь NODE_ENV, установленный на "production".

Метод 2. Установка NODE_ENV для текущего приложения

```
NODE_ENV=production node app.js
```

Это установит NODE_ENV только для текущего приложения. Это полезно, когда мы хотим протестировать наши приложения в разных окружениях.

Метод 3. Создание файла .env и его использование

Этот метод использует идею, объясненную здесь: https://stackoverflow.com/questions/22312671/setting-environment-variables-for-node-to-retrieve/28821696#28821696.

По сути, вы создаете файл .env и запускаете скрипт bash для установки переменных окружения.

Чтобы избежать написания bash-скрипта, можно использовать пакет env-cmd для загрузки переменных окружения, определенных в файле .env.

```
env-cmd .env node app.js
```

Метод 4. Использование пакета cross-env

Этот пакет позволяет устанавливать переменные окружения одинаковым способом для всех платформ.

После установки с помощью npm вы можете просто добавить его в ваш скрипт развертывания в package.json следующим образом:

```
"build:deploy": "cross-env NODE_ENV=production webpack"
```

Управление приложением с помощью менеджера процессов

Хорошей практикой является управление приложениями Node.js с помощью менеджеров процессов. Менеджер процессов помогает поддерживать работу приложения постоянно, перезапускать его при сбое, перезагружать без простоев и упрощать администрирование. Наиболее мощные из них (например PM2) имеют встроенный балансировщик нагрузки. PM2 также позволяет управлять логированием, мониторингом и кластеризацией приложений.

Менеджер процессов PM2

Установка PM2

```
npm install pm2 -g
```

Процесс можно запустить в кластерном режиме, используя встроенный балансировщик нагрузки для распределения нагрузки между процессами:

```
pm2 start app.js -i 0 --name "api"
```

(`-i` используется для указания количества порождаемых процессов. Если указано 0, количество процессов будет основано на количестве ядер CPU.)

При наличии нескольких пользователей в продакшене важно иметь единую точку для PM2. Поэтому команды `pm2` должны быть префиксированы с указанием местоположения (для конфигурации PM2), иначе для каждого пользователя будет запущен новый процесс PM2 с конфигурацией в его домашнем каталоге, что приведет к несогласованности.

```
PM2_HOME=/etc/.pm2 pm2 start app.js
```

Развертывание с использованием PM2

PM2 — это менеджер процессов для Node.js-приложений в продакшене, который позволяет поддерживать их работу постоянно и перезагружать их без простоев. PM2 также позволяет управлять логированием, мониторингом и кластеризацией приложений.

Установите PM2 глобально:

```
npm install -g pm2
```

Затем запустите Node.js-приложение с использованием PM2:

```
pm2 start server.js --name "my-app"
```

```
$ pm2 start app.js --name my-app
[PM2] restartProcessId process id 0
┌──────────┬────┬──────┬───────┬────────┬─────────┬────────┬──────────┬──────────┐
│ App name │ id │ mode │ pid   │ status │ restart │ uptime │ memory   │ watching │
├──────────┼────┼──────┼───────┼────────┼─────────┼────────┼──────────┼──────────┤
│ my-app   │ 0  │ fork │ 64029 │ online │ 1       │ 0s     │ 17.816 MB│ disabled │
└──────────┴────┴──────┴───────┴────────┴─────────┴────────┴──────────┴──────────┘
 Use the `pm2 show <id|name>` command to get more details about an app.
```

Полезные команды при работе с PM2

Список всех запущенных процессов

```
pm2 list
```

Остановить приложение:

```
pm2 stop my-app
```

Перезапустить приложение:

```
pm2 restart my-app
```

Посмотреть подробную информацию о приложении:

```
pm2 show my-app
```

Удалить приложение из реестра PM2:

```
pm2 delete my-app
```

Forever

`forever` — это инструмент командной строки, обеспечивающий непрерывную работу заданного скрипта. Простота интерфейса `forever` делает его идеальным для развертывания небольших приложений и скриптов на Node.js.

`forever` отслеживает ваш процесс и перезапускает его в случае сбоя.

Установка `forever` глобально:

```
npm install -g forever
```

Запуск приложения:

```
forever start server.js
```

Это запустит сервер и присвоит ему идентификатор (начиная с 0).

Перезапуск приложения:

```
forever restart 0
```

Здесь 0 — это идентификатор сервера.

Остановка приложения:

```
forever stop 0
```

Аналогично перезапуску 0 — это идентификатор сервера. Вы также можете указать идентификатор процесса или имя скрипта вместо идентификатора, присвоенного `forever`.

Для получения дополнительных команд перейдите по ссылке или QR-коду ниже: https://www.npmjs.com/package/forever.

Использование разных свойств/конфигураций для разных окружений, таких как dev, qa, staging и т. д.

Крупномасштабные приложения часто требуют разных свойств при работе в разных окружениях. Мы можем достичь этого, передавая аргументы приложению Node.js и используя те же аргументы в процессе Node для загрузки конкретного файла свойств окружения.

Предположим, у нас есть два файла свойств для разных окружений:

`dev.json`

```
{
    "PORT": 3000,
    "DB": {
        "host": "localhost",
        "user": "bob",
        "password": "12345"
    }
}
```

`qa.json`

```
{
    "PORT": 3001,
    "DB": {
        "host": "where_db_is_hosted",
```

```
        "user": "bob",
        "password": "54321"
    }
}
```

Следующий код в приложении экспортирует соответствующий файл свойств, который мы хотим использовать:

```
process.argv.forEach(function (val) {
    var arg = val.split("=");
    if (arg.length > 0) {
        if (arg[0] === 'env') {
            var env = require('./' + arg[1] + '.json');
            exports.prop = env;
        }
    }
});
```

Мы передаем аргументы приложению следующим образом:

```
node app.js env=dev
```

Если мы используем менеджер процессов, такой как `forever`, то это делается так же просто:

```
forever start app.js env=dev
```

Использование преимуществ кластеров

Один экземпляр Node.js работает в одном потоке. Чтобы воспользоваться преимуществами многоядерных систем, пользователь иногда захочет запустить кластер процессов Node.js для обработки нагрузки.

```
var cluster = require('cluster');
var numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
    // В реальном применении, вероятно, вы бы использовали больше чем
два рабочих процесса,
    // и, возможно, не размещали бы мастер и рабочие процессы в одном
файле.
    //
    // Вы также, конечно, можете усложнить логирование
    // и реализовать любую необходимую логику для предотвращения
    // DoS-атак и других вредоносных действий.
    //
```

```
    // Смотрите опции в документации по кластеру.
    //
    // Важно то, что мастер выполняет очень мало,
    // увеличивая устойчивость приложения к неожиданным ошибкам.
    console.log('your server is working on ' + numCPUs + ' cores');

    for (var i = 0; i < numCPUs; i++) {
        cluster.fork();
    }
    cluster.on('disconnect', function(worker) {
        console.error('disconnect!');
        //clearTimeout(timeout);
        cluster.fork();
    });

} else {
    require('./app.js');
}
```

Глава 24.
ECMAScript 2015 (ES6) с Node.js

Примеры

Объявления const/let

В отличие от `var`, `const/let` привязаны к лексической области видимости, а не к области видимости функции.

```
{
    var x = 1 // выйдет за пределы области видимости
    let y = 2 // привязан к лексической области видимости
    const z = 3 // привязан к лексической области видимости, являет-
ся константой
}

console.log(x) // 1
console.log(y) // ReferenceError: y is not defined
console.log(z) // ReferenceError: z is not defined
```

Стрелочные функции

Стрелочные функции автоматически привязываются к лексической области видимости `this` окружающего кода.

```
performSomething(result => {
    this.someVariable = result
})
```

вместо:

```
performSomething(function(result) {
    this.someVariable = result
}.bind(this))
```

Пример стрелочной функции

Рассмотрим пример, который выводит квадраты чисел 3, 5 и 7:

```
let nums = [3, 5, 7]

let squares = nums.map(function (n) {
    return n * n
})

console.log(squares)
```

Функцию, переданную в `.map`, также можно записать как стрелочную функцию, убрав ключевое слово `function` и добавив стрелку `=>`:

```
let nums = [3, 5, 7]

let squares = nums.map((n) => {
    return n * n
})

console.log(squares)
```

Однако это можно записать еще более кратко. Если тело функции состоит только из одного выражения, и это выражение вычисляет возвращаемое значение, фигурные скобки, оборачивающие тело функции, можно убрать, как и ключевое слово `return`.

```
let nums = [3, 5, 7]

let squares = nums.map(n => n * n)

console.log(squares)
```

Деструктуризация

```
let [x,y, ...nums] = [0, 1, 2, 3, 4, 5, 6];
console.log(x, y, nums);

let {a, b, ...props} = {a:1, b:2, c:3, d:{e:4}}
console.log(a, b, props);

let dog = {name: 'fido', age: 3};
let {name:n, age} = dog;
console.log(n, age);
```

Flow

```
/* @flow */

function product(a: number, b: number){
    return a * b;
}

const b = 3;
let c = [1,2,3,,{}];
let d = 3;

import request from 'request';

request('http://dev.markitondemand.com/MODApis/Api/v2/Quote/
json?symbol=AAPL', (err, res,
payload)=>{
    payload = JSON.parse(payload);
    let {LastPrice} = payload;
    console.log(LastPrice);
});
```

Класс ES6

```
class Mammal {
    constructor(legs){
        this.legs = legs;
    }

    eat(){
        console.log('eating...');
    }
    static count(){
        console.log('static count...');
    }
}

class Dog extends Mammal{
    constructor(name, legs){
        super(legs);
        this.name = name;
    }

    sleep(){
        super.eat();
```

```
        console.log('sleeping');
    }
}


let d = new Dog('fido', 4);
d.sleep();
d.eat();
console.log('d', d);
```

Глава 25.
Окружение

Примеры

Доступ к переменным окружения

Свойство `process.env` возвращает объект, содержащий пользовательское окружение. Этот объект выглядит следующим образом:

```
{
    TERM: 'xterm-256color',
    SHELL: '/usr/local/bin/bash',
    USER: 'maciej',
    PATH: '~/.bin/:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin',
    PWD: '/Users/maciej',
    EDITOR: 'vim',
    SHLVL: '1',
    HOME: '/Users/maciej',
    LOGNAME: 'maciej',
    _: '/usr/local/bin/node'
}
process.env.HOME // '/Users/maciej'
```

Если установить переменную окружения `FOO` в значение `foobar`, то она будет доступна следующим образом:

```
process.env.FOO // 'foobar'
```

Аргументы командной строки process.argv

`process.argv` — это массив, содержащий аргументы командной строки. Первый элемент будет `node`, второй элемент будет с именем JavaScript-файла. Следующие элементы будут любыми дополнительными аргументами командной строки.

Пример кода

Вывод суммы всех аргументов командной строки:

```
index.js
var sum = 0;
for (i = 2; i < process.argv.length; i++) {
    sum += Number(process.argv[i]);
}
console.log(sum);
```

Пример использования

```
node index.js 2 5 6 7
```

Вывод будет равен 20

Краткое объяснение кода

Здесь в цикле `for (i = 2; i < process.argv.length; i++)` цикл начинается с 2, потому что первые два элемента в массиве `process.argv` всегда такие: `['path/to/node.exe', 'path/to/js/file', ...]`.

Преобразование в число `Number(process.argv[i])`, потому что элементы в массиве `process.argv` всегда являются строками.

Использование различных свойств/конфигураций для разных окружений, таких как dev, qa, staging и т. д.

Крупномасштабные приложения часто требуют различных свойств при запуске в разных окружениях. Этого можно достичь путем передачи аргумента в приложение Node.js и использования этого аргумента для загрузки определенного файла свойств окружения.

Предположим, у нас есть два файла свойств для разных окружений:

- `dev.json`

```
{
    PORT : 3000,
    DB : {
        host : "localhost",
        user : "bob",
        password : "12345"
    }
}
```

- `qa.json`

```
{
    PORT : 3001,
```

```
    DB : {
        host : "where_db_is_hosted",
        user : "bob",
        password : "54321"
    }
}
```

Следующий код в приложении экспортирует соответствующий файл свойств, который мы хотим использовать.

Предположим, что код находится в `environment.js`:

```
process.argv.forEach(function (val, index, array) {
    var arg = val.split("=");

    if (arg.length > 0) {
        if (arg[0] === 'env') {
            var env = require('./' + arg[1] + '.json');
            module.exports = env;
        }
    }
});
```

Мы передаем аргументы в приложение следующим образом:

```
node app.js env=dev
```

Если мы используем менеджер процессов, такой как `forever`, то это так же просто, как:

```
forever start app.js env=dev
```

Как использовать конфигурационный файл

```
var env = require("environment.js");
```

Загрузка свойств окружения из «файла свойств»

- Установите `properties-reader`:

```
npm install properties-reader -save
```

- Создайте директорию `env` для хранения ваших файлов свойств

```
mkdir env
```

- Создайте `environments.js`:

```
process.argv.forEach(function (val, index, array) {
    var arg = val.split("=");
    if (arg.length > 0) {
```

```
        if (arg[0] === 'env') {
            var env = require('./env/' + arg[1] + '.properties');
            module.exports = env;
        }
    }
});
```

- Пример файла свойств `development.properties`:

```
# Dev properties
[main]
# Порт приложения для запуска сервера node
app.port=8080
[database]
# Подключение к базе данных mysql
mysql.host=localhost
mysql.port=2500
```

- Пример использования загруженных свойств

```
var environment = require('./environments');
var PropertiesReader = require('properties-reader');
var properties = new PropertiesReader(environment);
var someVal = properties.get('main.app.port');
```

- Запуск сервера Express:

```
npm start env=development
```

или

```
npm start env=production
```

Глава 26.
Источники событий

Примечания

Когда событие «срабатывает» (что означает то же самое, что и «публикация события» или «генерация события»), каждый слушатель будет вызван синхронно вместе с любыми данными, переданными в emit(), независимо от того, сколько аргументов вы передадите:

```
myDog.on('bark', (howLoud, howLong, howIntense) => {
    // обработка события
})
myDog.emit('bark', 'loudly', '5 seconds long', 'fiercely')
```

На источник можно перейти по ссылке: https://nodejs.org/dist/latest-v6.x/docs/api/events.html#events_asynchronous_vs_synchronous или по QR-коду ниже.

Слушатели будут вызваны в том порядке, в котором они были зарегистрированы:

```
myDog.on('urinate', () => console.log('My first thought was "Oh-no"'))
myDog.on('urinate', () => console.log('My second thought was "Not my
lawn :)"'))
myDog.emit('urinate')
```

```
// Эти console.log будут выполнены в правильном порядке, так как они были зарегистрированы в этом порядке.
```

Но если вам нужно, чтобы слушатель сработал первым, перед всеми остальными слушателями, которые уже были добавлены, вы можете использовать prependListener() следующим образом:

```
myDog.prependListener('urinate', () => console.log('This happens before my first and second thoughts, even though it was registered after them'))
```

Если вам нужно прослушивать событие, но вы хотите, чтобы оно срабатывало только один раз, вы можете использовать once вместо on, или prependOnceListener вместо prependListener. После того как событие сработает и слушатель будет вызван, слушатель автоматически удалится и не будет вызываться снова при следующем срабатывании события.

Наконец, если вы хотите удалить всех слушателей и начать с чистого листа, вы можете сделать это так:

```
myDog.removeAllListeners()
```

Примеры

Аналитика HTTP через источник событий

В коде HTTP-сервера (например, server.js):

```
const EventEmitter = require('events')
const serverEvents = new EventEmitter()

// Настройка HTTP сервера
const http = require('http')
const httpServer = http.createServer((request, response) => {
    // Обработка запроса...
    // Затем генерируем событие о том, что произошло
    serverEvents.emit('request', request.method, request.url)
});

// Экспортируем генератор событий
module.exports = serverEvents
```

В коде надзора (например supervisor.js):

```
const server = require('./server.js')

// Поскольку сервер экспортировал генератор событий, мы можем слушать его изменения:
```

```
server.on('request', (method, url) => {
    console.log(`Got a request: ${method} ${url}`)
})
```

Каждый раз, когда сервер получает запрос, он генерирует событие, называемое request, которое будет прослушиваться надзором, и затем надзор может отреагировать на событие.

Основы

Источники событий встроены в Node и используются для реализации паттерна pub-sub, при котором издатель генерирует события, на которые подписчики могут реагировать. В терминологии Node издатели называются источниками событий (Event Emitters), и они генерируют события, тогда как подписчики называются слушателями (listeners), и они реагируют на события.

```
// Подключаем модуль событий для начала работы с ними
const EventEmitter = require('events').EventEmitter;
// У собак есть события для публикации или генерации
class Dog extends EventEmitter {};
class Food {};

let myDog = new Dog();

// Когда myDog жует, выполните следующую функцию
myDog.on('chew', (item) => {
    if (item instanceof Food) {
        console.log('Good dog');
    } else {
        console.log(`Time to buy another ${item}`);
    }
});

myDog.emit('chew', 'shoe'); // Выведет в консоль: 'Time to buy another
shoe'
const bacon = new Food();
myDog.emit('chew', bacon); // Выведет в консоль: 'Good dog'
```

В приведенном выше примере собака является издателем/источником событий, в то время как функция, проверяющая предмет, является подписчиком/слушателем. Вы также можете создать больше слушателей:

```
myDog.on('bark', () => {
    console.log('WHO\'S AT THE DOOR?');
    // Паника
});
```

Также может быть несколько слушателей для одного события, и можно даже удалить слушателей:

```
myDog.on('chew', takeADeepBreathe);
myDog.on('chew', calmDown);
// Отменяем предыдущую строку следующей:
myDog.removeListener('chew', calmDown);
```

Если вы хотите прослушивать событие только один раз, вы можете использовать:

```
myDog.once('chew', pet);
```

...которое автоматически удалит слушателя без условий гонки (race condition).

Получение имен событий, на которые подписаны

Функция `EventEmitter.eventNames()` вернет массив, содержащий имена событий, на которые в данный момент подписаны.

```
const EventEmitter = require("events");
class MyEmitter extends EventEmitter{}
var emitter = new MyEmitter();
emitter
.on("message", function(){ // слушаем событие message
    console.log("a message was emitted!");
})
.on("message", function(){ // слушаем событие message
    console.log("this is not the right message");
})
.on("data", function(){ // слушаем событие data
    console.log("a data just occured!!");
});

console.log(emitter.eventNames()); //=> ["message","data"]
emitter.removeAllListeners("data"); //=> удаляем всех слушателей события data
console.log(emitter.eventNames()); //=> ["message"]
```

Получение количества слушателей, зарегистрированных для определенного события

Функция `Emitter.listenerCount(eventName)` вернет количество слушателей, которые в данный момент прослушивают событие, переданное в качестве аргумента.

```
const EventEmitter = require("events");
class MyEmitter extends EventEmitter{}
var emitter = new MyEmitter();
emitter
.on("data", () => { // добавляем слушателя для события data
    console.log("data event emitter");
});

console.log(emitter.listenerCount("data")) // => 1
console.log(emitter.listenerCount("message")) // => 0

emitter.on("message", function mListener(){ // добавляем слушателя
для события message
    console.log("message event emitted");
});

console.log(emitter.listenerCount("data")) // => 1
console.log(emitter.listenerCount("message")) // => 1

emitter.once("data", (stuff) => { // добавляем еще одного слушателя
для события data
    console.log(`Tell me my ${stuff}`);
});

console.log(emitter.listenerCount("data")) // => 2
console.log(emitter.listenerCount("message")) // => 1
```

Глава 27.
Event loop (цикл событий)

Введение

В этой главе рассматривается, как возникла концепция цикла событий и как она может использоваться для высокопроизводительных серверов и событийно-ориентированных приложений, таких как GUI.

Примеры

Рассмотрим, как развивалась концепция цикла событий.

Event loop на псевдокоде

Event loop — это цикл, который ожидает событий, а затем реагирует на них:

```
while True:
    ожидать, пока не произойдет что-то
    реагировать на произошедшее
```

Пример однопоточного HTTP-сервера без event loop

```
while True:
    сокет = ожидать следующего TCP-соединения
    прочитать HTTP-заголовки запроса из (сокета)
    содержимое_файла = получить запрашиваемый файл с диска
    записать HTTP-заголовки ответа в (сокет)
    записать (содержимое_файла) в (сокет)
    закрыть (сокет)
```

Вот простой пример HTTP-сервера, который является однопоточным, но без event loop. Проблема здесь заключается в том, что сервер ждет, пока каж-

дый запрос не будет завершен, прежде чем начать обработку следующего. Если требуется некоторое время для чтения заголовков HTTP-запроса или для получения файла с диска, мы должны иметь возможность начать обработку следующего запроса, пока ожидаем завершения предыдущего.

Наиболее распространенное решение — сделать программу многопоточной.

Пример многопоточного HTTP-сервера без event loop

```
функция обработать_соединение(сокет):
    прочитать HTTP-заголовки запроса из (сокета)
    содержимое_файла = получить запрашиваемый файл с диска
    записать HTTP-заголовки ответа в (сокет)
    записать (содержимое_файла) в (сокет)
    закрыть (сокет)
while True:
    сокет = ожидать следующего TCP-соединения
    запустить новый поток, выполняющий обработку соединения(soket)
```

Теперь мы сделали наш небольшой HTTP-сервер многопоточным. Таким образом, мы можем сразу же приступить к следующему запросу, так как текущий запрос выполняется в фоновом потоке. Многие серверы, включая Apache, используют этот подход.

Но он не идеален. Одним из ограничений является то, что вы можете создать только определенное количество потоков. Для рабочих нагрузок, где имеется огромное количество соединений, но каждое соединение требует внимания лишь изредка, многопоточная модель не будет работать эффективно. Решением в таких случаях является использование event loop.

Пример HTTP-сервера с event loop

```
while True:
    событие = ожидать следующего события
    if (событие.тип == НОВОЕ_TCP_СОЕДИНЕНИЕ):
        соединение = новое соединение
        соединение.сокет = событие.сокет
         начать чтение HTTP-заголовков запроса из (соединение.сокет)
с пользовательскими данными = (соединение)
    else if (событие.тип == ЗАВЕРШЕНО_ЧТЕНИЕ_ИЗ_СОКЕТА):
        соединение = событие.пользовательские_данные
        начать получение запрашиваемого файла с диска с пользователь-
скими данными = (соединение)
    else if (событие.тип == ЗАВЕРШЕНО_ЧТЕНИЕ_С_ДИСКА):
        соединение = событие.пользовательские_данные
```

```
        соединение.содержимое_файла = данные, которые мы получили с диска
        соединение.текущее_состояние = "запись заголовков"
        начать запись HTTP-заголовков ответа в (соединение.сокет)
с пользовательскими данными = (соединение)
    else if (событие.тип == ЗАВЕРШЕНО_ЗАПИСЬ_В_СОКЕТ):
        соединение = событие.пользовательские_данные
        if (соединение.текущее_состояние == "запись заголовков"):
            соединение.текущее_состояние = "запись содержимого файла"
            начать запись (соединение.содержимое_файла) в (соедине-
ние.сокет) с пользовательскими данными = (соединение)
        else if (соединение.текущее_состояние == "запись содержимого
файла"):
            закрыть (соединение.сокет)
```

Думаем, этот псевдокод понятен. Вот что происходит: мы ждем, пока что-то случится. Всякий раз, когда создается новое соединение или существующее соединение требует нашего внимания, мы обрабатываем его, затем возвращаемся к ожиданию. Таким образом, мы эффективно работаем, когда имеется много соединений, и каждое из них требует внимания лишь изредка.

В реальном приложении (не в псевдокоде), работающем на Linux, часть «ожидать следующего события» была бы реализована с помощью системного вызова `poll()` или `epoll()`. Части «начать чтение HTTP-заголовков запроса из (соединение.сокет)» были бы реализованы с помощью системных вызовов `recv()` или `send()` в неблокирующем режиме.

Ссылка:

Статья «Как работает цикл событий?» доступна по ссылке https://www.quora.com/How-does-an-event-loop-work или по QR-коду ниже.

Глава 28.
Обработка исключений

Примеры

Обработка исключений в Node.js

В Node.js существует три основных способа обработки исключений/ошибок:

1. блок `try-catch`
2. передача ошибки в качестве первого аргумента в `callback`
3. генерация события ошибки с использованием `eventEmitter`

`try-catch` используется для перехвата исключений, возникающих в синхронном коде. Если вызывающий код (или код, который его вызывает) использует `try-catch`, то они могут поймать ошибку. Если ни один из вызовов не использует `try-catch`, программа завершится с ошибкой.

Если использовать `try-catch` для асинхронной операции и исключение возникнет в `callback` асинхронного метода, то оно не будет перехвачено `try-catch`. Чтобы перехватить исключение из `callback` асинхронной операции, предпочтительно использовать `promise`.

```
Пример для лучшего понимания
// ** Пример - 1 **
function doSomeSynchronousOperation(req, res) {
    if(req.body.username === ''){
        throw new Error('User Name cannot be empty');
    }
    return true;
}

// вызов метода выше
try {
    // синхронный код
    doSomeSynchronousOperation(req, res)
```

```
} catch(e) {
    // исключение обрабатывается здесь
    console.log(e.message);
}

// ** Пример - 2 **
function doSomeAsynchronousOperation(req, res, cb) {
    // имитация асинхронной операции
    return setTimeout(function(){
        cb(null, []);
    }, 1000);
}
try {
    // асинхронный код
    doSomeAsynchronousOperation(req, res, function(err, rs){
        throw new Error("async operation exception");
    })
} catch(e) {
    // исключение не будет обработано здесь
    console.log(e.message);
}
//Исключение не обработано, и поэтому приложение завершится с ошибкой.
```

В основном в Node.js используются обратные вызовы, так как они передают событие асинхронно. Пользователь передает вам функцию (callback), и вы вызываете ее позже, когда асинхронная операция завершится.

Обычно callback вызывается как callback(err, result), где только одно из значений err и result не равно null, в зависимости от того, завершилась операция успешно или нет.

```
function doSomeAsynchronousOperation(req, res, callback) {
    setTimeout(function(){
        return callback(new Error('User Name cannot be empty'));
    }, 1000);
    return true;
}

doSomeAsynchronousOperation(req, res, function(err, result) {
    if (err) {
        // исключение обрабатывается здесь
        console.log(err.message);
    }
    // выполнение дальнейших действий с действительными данными
});
```

Генерация событий

В более сложных случаях вместо использования callback функция сама может возвращать объект EventEmitter, и от вызывающего кода ожидается, что он будет отслеживать события ошибок на эмиттере.

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

function doSomeAsynchronousOperation(req, res) {
    let myEvent = new EventEmitter();
    // выполняется асинхронно
    setTimeout(function(){
        myEvent.emit('error', new Error('User Name cannot be empty'));
    }, 1000);
    return myEvent;
}
// Вызов функции
let event = doSomeAsynchronousOperation(req, res);
event.on('error', function(err) {
    console.log(err);
});
event.on('done', function(result) {
    console.log(result); // true
});
```

Управление необработанными исключениями

Так как Node.js работает в одном потоке, необработанные исключения могут стать проблемой, о которой стоит помнить при разработке приложений.

Тихая обработка исключений

Многие разработчики позволяют серверу Node.js незаметно «проглатывать» ошибки.

```
// Тихая обработка исключений
process.on('uncaughtException', function (err) {
    console.log(err);
});
```

Это работает, но:

- Первопричина останется неизвестной, а значит, данный способ не поможет в решении проблемы, вызвавшей исключение (ошибку).

- В случае если по какой-либо причине соединение с базой данных (pool) закрыто, это приведет к постоянной передаче ошибок, а это означает, что сервер будет работать, но не переподключится к базе данных.

Возврат к начальному состоянию

В случае `uncaughtException` желательно перезапустить сервер и вернуть его в начальное состояние, при котором он точно будет работать. Исключение записывается в лог, приложение завершает работу, но поскольку оно будет работать в контейнере, который следит за тем, чтобы сервер продолжал работать, мы добьемся перезапуска сервера (возврата к начальному рабочему состоянию).

- Установка утилиты `forever` (или другого CLI-инструмента, чтобы сервер Node.js работал непрерывно):

```
npm install forever -g
```

- Запуск сервера с `forever`:

```
forever start app.js
```

Причина, по которой сервер запускается с `forever`, заключается в том, что после завершения работы сервера процесс `forever` снова запустит сервер.

- Перезапуск сервера:

```
process.on('uncaughtException', function (err) {
    console.log(err);
    // механизм логирования
    // ....
    process.exit(1); // завершает процесс
});
```

Заметим, что ранее также существовал способ обработки исключений с помощью кластеров и доменов. Доменный модуль был признан устаревшим, подробнее об этом можно узнать здесь: https://nodejs.org/api/domain.html или по QR-коду ниже.

Ошибки и Promises (промисы)

Promises обрабатывают ошибки иначе, чем синхронный код или код на основе callback.

```
const p = new Promise(function (resolve, reject) {
    reject(new Error('Oops'));
});

// все, что отклонено внутри promise, будет доступно через catch
// когда promise отклонен, .then не будет вызван
p
    .then(() => {
        console.log("won't be called");
    })
    .catch(e => {
        console.log(e.message); // вывод: Oops
    })
    // после перехвата ошибки выполнение продолжается
    .then(() => {
        console.log('hello!'); // вывод: hello!
    });
```

В настоящее время ошибки, вызванные в promise и не перехваченные, приводят к тому, что ошибка «проглатывается», а это может затруднить отладку. Данную проблему можно решить, используя инструменты линтинга, такие как eslint, или путем обеспечения того, чтобы у вас всегда был блок catch.

Такое поведение устарело в Node.js 8 в пользу завершения процесса Node.js.

Глава 29.
Выполнение файлов или команд с помощью дочерних процессов

Синтаксис

- `child_process.exec(command[, options][, callback])`
- `child_process.execFile(file[, args][, options][, callback])`
- `child_process.fork(modulePath[, args][, options])`
- `child_process.spawn(command[, args][, options])`
- `child_process.execFileSync(file[, args][, options])`
- `child_process.execSync(command[, options])`
- `child_process.spawnSync(command[, args][, options])`

Замечания

При работе с дочерними процессами все асинхронные методы будут возвращать экземпляр `ChildProcess`, в то время как все синхронные версии будут возвращать результат выполнения. Как и другие синхронные операции в Node.js, если возникает ошибка, она будет выброшена.

Примеры

Создание нового процесса для выполнения команды

Чтобы создать новый процесс, если вам нужен небуферизированный вывод (например, для долго работающих процессов, которые могут выводить данные на протяжении определенного времени, а не сразу после завершения), используйте `child_process.spawn()`.

Этот метод создает новый процесс с помощью заданной команды и массива аргументов. Возвращаемое значение — это экземпляр `ChildProcess`, который, в свою очередь, предоставляет свойства `stdout` и `stderr`. Оба этих потока являются экземплярами `stream.Readable`.

Следующий код эквивалентен команде `ls -lh /usr`.

```
const spawn = require('child_process').spawn;
const ls = spawn('ls', ['-lh', '/usr']);
ls.stdout.on('data', (data) => {
    console.log(`stdout: ${data}`);
});
ls.stderr.on('data', (data) => {
    console.log(`stderr: ${data}`);
});

ls.on('close', (code) => {
    console.log(`child process exited with code ${code}`);
});
```

Другой пример команды:

```
zip -0vr "archive" ./image.png
```

...может быть записан как:

```
spawn('zip', ['-0vr', '"archive"', './image.png']);
```

Создание оболочки для выполнения команды

Для выполнения команды в оболочке, где требуется буферизированный вывод (то есть он не является потоком), используйте `child_process.exec`. Например, если вам нужно выполнить команду `cat *.js file | wc -l` без дополнительных опций, это будет выглядеть так:

```
const exec = require('child_process').exec;

exec('cat *.js file | wc -l', (err, stdout, stderr) => {
    if (err) {
        console.error(`exec error: ${err}`);
        return;
    }
    console.log(`stdout: ${stdout}`);
    console.log(`stderr: ${stderr}`);
});
```

Функция принимает до трех параметров

```
child_process.exec(command[, options][, callback]);
```

Параметр `command` является строкой и обязателен, а объект `options` и `callback` — необязательными. Если объект `options` не указан, то `exec` будет использовать следующие значения по умолчанию:

```
{
    encoding: 'utf8',
    timeout: 0,
    maxBuffer: 200*1024,
    killSignal: 'SIGTERM',
    cwd: null,
    env: null
}
```

Объект `options` также поддерживает параметр `shell`, который по умолчанию равен `/bin/sh` на UNIX и `cmd.exe` на Windows, параметр `uid` для задания идентификатора пользователя процесса и параметр `gid` для задания группы.

`Callback`, который вызывается после завершения выполнения команды, принимает три аргумента (`err`, `stdout`, `stderr`). Если команда выполнена успешно, `err` будет равен `null`, иначе это будет экземпляр `Error`, где `err.code` — это код завершения процесса, а `err.signal` — сигнал, который был отправлен для его завершения.

Аргументы `stdout` и `stderr` представляют собой вывод команды. Он декодируется с помощью кодировки, указанной в объекте `options` (по умолчанию: строка), но также может быть возвращен как объект типа `Buffer`.

Существует также синхронная версия `exec`, которая называется `execSync`. Синхронная версия не принимает `callback` и возвращает `stdout` вместо экземпляра `ChildProcess`. Если синхронная версия сталкивается с ошибкой, она выбросит исключение и остановит выполнение программы. Пример использования:

```
const execSync = require('child_process').execSync;
const stdout = execSync('cat *.js file | wc -l');
console.log(`stdout: ${stdout}`);
```

Создание процесса для запуска исполняемого файла

Если вы хотите запустить файл, такой как исполняемый файл, используйте `child_process.execFile`. Вместо создания оболочки, как это делает `child_process.exec`, он напрямую создает новый процесс, что немного эффективнее для выполнения команды. Функция может быть использована следующим образом:

```
const execFile = require('child_process').execFile;
const child = execFile('node', ['--version'], (err, stdout, stderr)
=> {
```

```
    if (err) {
        throw err;
    }
    console.log(stdout);
});
```

В отличие от child_process.exec эта функция принимает до четырех параметров, где второй параметр — массив аргументов, которые вы хотите передать исполняемому файлу:

```
child_process.execFile(file[, args][, options][, callback]);
```

Формат options и callback в остальном идентичен child_process.exec. То же самое относится и к синхронной версии функции:

```
const execFileSync = require('child_process').execFileSync;
const stdout = execFileSync('node', ['--version']);
console.log(stdout);
```

Глава 30.
Экспорт и использование модулей

Примечания

Хотя почти все в Node.js выполняется асинхронно, функция `require()` является исключением. Поскольку модули на практике нужно загружать только один раз, это является блокирующей операцией, и ее следует использовать правильно.

Модули кешируются после первой загрузки. Если вы редактируете модуль в процессе разработки, вам нужно удалить его из кеша модулей, чтобы использовать новые изменения. Тем не менее, даже если модуль очищен из кеша, сам модуль не будет освобожден сборщиком мусора, поэтому в производственных средах нужно быть осторожным.

Примеры

Загрузка и использование модуля

Модуль можно «импортировать» или «подключить» с помощью функции `require()`. Например, чтобы загрузить модуль `http`, который поставляется с Node.js, можно использовать следующий код:

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

Помимо модулей, поставляемых вместе с Node.js, вы также можете подключать модули, установленные через `npm`, например. Если вы уже установили Express на свою систему через `npm install express`, можно просто написать:

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

Вы также можете включить модули, написанные вами, как часть вашего приложения. В этом случае, чтобы подключить файл с именем `lib.js`, находящийся в той же директории, что и текущий файл, сделайте следующее:

```
const mylib = require('./lib');
```

Обратите внимание, что расширение файла можно опустить, и будет подразумеваться .js. Как только модуль загрузится, переменная будет содержать объект, который включает методы и свойства, опубликованные из подключаемого файла. Полный пример:

```
const http = require('http');
// Модуль `http` содержит свойство `STATUS_CODES`
console.log(http.STATUS_CODES[404]); // выводит 'Not Found'
// Также содержит метод `createServer()`
http.createServer(function(req, res) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write('<html><body>Module Test</body></html>');
    res.end();
}).listen(80);
```

Создание модуля hello-world.js

Node предоставляет интерфейс `module.exports` для экспорта функций и переменных в другие файлы. Самый простой способ сделать это — экспортировать только один объект (функцию или переменную), как показано в первом примере.

```
hello-world.js
module.exports = function(subject) {
    console.log('Hello ' + subject);
};
```

Если вы не хотите, чтобы весь экспорт представлял собой один объект, можно экспортировать функции и переменные как свойства объекта `exports`. Три следующих примера демонстрируют это немного по-разному:

- `hello-venus.js`: определение функции выполнено отдельно, а затем добавлено как свойство объекта `module.exports`.
- `hello-jupiter.js`: определения функций напрямую помещены в значения свойств объекта `module.exports`.
- `hello-mars.js`: определение функции напрямую указано как свойство объекта `exports`, что является короткой версией использования `module.exports`.

hello-venus.js

```
function hello(subject) {
    console.log('Venus says Hello ' + subject);
}
```

```
module.exports = {
    hello: hello
};
```

hello-jupiter.js

```
module.exports = {
    hello: function(subject) {
        console.log('Jupiter says hello ' + subject);
    },
    bye: function(subject) {
        console.log('Jupiter says goodbye ' + subject);
    }
};
```

hello-mars.js

```
exports.hello = function(subject) {
    console.log('Mars says Hello ' + subject);
};
```

Загрузка модуля по имени директории

У нас есть директория с именем hello, которая содержит следующие файлы:

index.js

```
// hello/index.js
module.exports = function(){
    console.log('Hej');
};
```

main.js

```
// hello/main.js
// Мы можем включить другие файлы, которые мы определили, с помощью
метода `require()`
var hw = require('./hello-world.js'),
    hm = require('./hello-mars.js'),
    hv = require('./hello-venus.js'),
    hj = require('./hello-jupiter.js'),
    hu = require('./index.js');
// Поскольку мы назначили нашу функцию всему объекту `module.exports`,
// мы можем использовать ее напрямую
hw('World!'); // выводит "Hello World!"
// В этом случае мы назначили нашу функцию свойству `hello` объекта
`exports`,
// поэтому нужно использовать это свойство здесь тоже
```

```
hm.hello('Solar System!'); // выводит "Mars says Hello Solar System!"
// Результат назначения module.exports в hello-world.js такой же,
как и в hello-world.js
hv.hello('Milky Way!'); // выводит "Venus says Hello Milky Way!"
hj.hello('Universe!'); // выводит "Jupiter says hello Universe!"
hj.bye('Universe!'); // выводит "Jupiter says goodbye Universe!"
hu(); // выводит 'hej'
```

Признание недействительным кеша модуля

В процессе разработки вы можете заметить, что использование `require()` для одного и того же модуля несколько раз всегда возвращает один и тот же модуль, даже если вы внесли изменения в данный файл. Это происходит потому, что модули кешируются при первой загрузке, и последующие загрузки модуля будут происходить из кеша.

Чтобы обойти эту проблему, нужно удалить запись в кеше. Например, если вы загрузили модуль:

```
var a = require('./a');
```

...затем можно удалить запись в кеше:

```
var rpath = require.resolve('./a.js');
delete require.cache[rpath];
```

...после чего снова загрузить модуль:

```
var a = require('./a');
```

Учтите, что это не рекомендуется в производственной среде, поскольку `delete` удаляет только ссылку на загруженный модуль, но не сами загруженные данные. Модуль не будет собран сборщиком мусора, поэтому неправильное использование этой функции может привести к утечке памяти.

Создание собственных модулей

Вы также можете использовать объект для публичного экспорта и последовательно добавлять методы к этому объекту:

```
const auth = module.exports = {}
const config = require('../config')
const request = require('request')
auth.email = function (data, callback) {
    // Аутентификация с использованием email
}
auth.facebook = function (data, callback) {
```

```
    // Аутентификация с использованием аккаунта Facebook
}
auth.twitter = function (data, callback) {
    // Аутентификация с использованием аккаунта Twitter
}
auth.slack = function (data, callback) {
    // Аутентификация с использованием аккаунта Slack
}
auth.stack_overflow = function (data, callback) {
    // Аутентификация с использованием аккаунта Stack Overflow
}
```

Чтобы использовать любой из этих методов, просто подключите модуль, как обычно:

```
const auth = require('./auth')
module.exports = function (req, res, next) {
    auth.facebook(req.body, function (err, user) {
        if (err) return next(err)
        req.user = user
        next()
    })
}
```

Каждый модуль загружается только один раз

Node.js выполняет модуль только при первом подключении. Любые последующие вызовы `require` будут использовать тот же объект, таким образом, код в модуле не будет выполняться повторно. Также Node кеширует модули при первой загрузке с использованием `require`. Это снижает количество операций чтения файлов и помогает ускорить работу приложения.

myModule.js

```
console.log(123);
exports.var1 = 4;
```

index.js

```
var a = require('./myModule'); // Вывод: 123
var b = require('./myModule'); // Вывода нет
console.log(a.var1); // Вывод: 4
console.log(b.var1); // Вывод: 4
a.var2 = 5;
console.log(b.var2); // Вывод: 5
```

Загрузка модуля из node_modules

Модули могут быть подключены без использования относительных путей, если они находятся в специальной директории node_modules.

Например, чтобы подключить модуль с именем foo из файла index.js, можно использовать следующую структуру директорий:

```
index.js
\- node_modules
    \- foo
        |- foo.js
        \- package.json
```

Модули должны быть помещены в отдельную директорию вместе с файлом package.json. Поле main в файле package.json должно указывать на точку входа для вашего модуля — это файл, который будет импортирован, когда пользователи используют require('your-module'). Если main не указано, по умолчанию будет использоваться index.js. Кроме того, вы можете ссылаться на файлы относительно вашего модуля, просто добавив относительный путь к вызову require: require('your-module/path/to/file').

Модули также могут быть загружены из директорий node_modules, расположенных выше в иерархии файловой системы. Если у нас есть следующая структура директорий:

```
my-project
\- node_modules
    |- foo // модуль foo
    \- ...
    \- baz // модуль baz
        \- node_modules
            \- bar // модуль bar
```

...мы сможем подключить модуль foo из любого файла в bar с использованием require('foo').

Обратите внимание, что Node будет загружать только тот модуль, который находится ближе всего к файлу в иерархии файловой системы, начиная с директории (текущая директория файла/node_modules). Node будет искать директории таким образом вплоть до корня файловой системы.

Вы можете либо установить новые модули из npm-репозитория или других npm-репозиториев, либо создать свои собственные.

Папка (директория) как модуль

Модули могут быть разделены на несколько .js-файлов в одной папке. Пример в папке my_module:

function_one.js

```
module.exports = function() {
    return 1;
}
```

function_two.js

```
module.exports = function() {
    return 2;
}
```

index.js

```
exports.f_one = require('./function_one.js');
exports.f_two = require('./function_two.js');
```

Такой модуль используется со ссылкой на него по имени папки:

```
var split_module = require('./my_module');
```

Учтите, что если вы подключаете модуль, опуская ./ или любое указание на путь к папке в аргументе функции `require`, Node попытается загрузить модуль из папки `node_modules`.

В качестве альтернативы вы можете создать в той же папке файл `package.json` со следующим содержимым:

```
{
    "name": "my_module",
    "main": "./your_main_entry_point.js"
}
```

Таким образом, вам не обязательно называть основной файл модуля `index`.

Глава 31.
Экспорт и импорт модулей в Node.js

Примеры

Использование простого модуля в Node.js

Что такое модуль в Node.js?

> Модуль инкапсулирует связанный код в единое целое. При создании модуля это можно интерпретировать как перенос всех связанных функций в один файл.

Ссылка на статью: https://www.sitepoint.com/understanding-module-exports-exports-node-js/.

Теперь давайте рассмотрим пример. Представьте, что все файлы находятся в одной директории:

Файл: printer.js

```
"use strict";
exports.printHelloWorld = function (){
    console.log("Hello World!!!");
}
```

Другой способ использования модулей:

Файл: animals.js

```
"use strict";
module.exports = {
    lion: function() {
        console.log("ROAARR!!!");
    }
};
```

Файл: app.js

Запустите этот файл, перейдя в свою директорию и набрав `node app.js`

```
"use strict";

//require('./path/to/module.js') указывает node, какой модуль загрузить
var printer = require('./printer');
var animals = require('./animals');
printer.printHelloWorld(); // выводит "Hello World!!!"
animals.lion(); // выводит "ROAARR!!!"
```

Использование Imports в ES6

Node.js построен на современных версиях V8. Обновляясь до последних версий этого движка, мы обеспечиваем своевременное внедрение новых функций из спецификации JavaScript ECMA-262 для разработчиков Node.js, а также продолжаем улучшать производительность и стабильность.

Все функции ECMAScript 2015 (ES6) разделены на три группы: отправляемые, подготовленные и находящиеся в разработке.

Все отправляемые функции, которые V8 считает стабильными, включены по умолчанию в Node.js и НЕ требуют никаких флагов времени выполнения. Подготовленные функции, которые почти завершены, но не считаются стабильными командой V8, требуют флага времени выполнения: `--harmony`. Функции, находящиеся в разработке, могут быть активированы индивидуально с помощью соответствующих флагов `harmony`, хотя это крайне не рекомендуется, если только для тестирования. Обратите внимание: эти флаги предоставляются V8 и могут изменяться без уведомления.

В настоящее время ES6 поддерживает операторы импорта нативно.

Итак, если у нас есть файл под названием `fun.js`...

```
export default function say(what){
    console.log(what);
}
export function sayLoud(whoot) {
```

```
    say(whoot.toUpperCase());
}
```

...и если существует другой файл с именем app.js, в который мы хотим импортировать ранее определенные функции, есть три способа сделать это.

Импорт по умолчанию

```
import say from './fun';
say('Hello Stack Overflow!!'); // Вывод: Hello Stack Overflow!!
```

Импортирует функцию say(), потому что она помечена как экспорт по умолчанию в исходном файле (export default ...).

Именованный импорт

```
import { sayLoud } from './fun';
sayLoud('JS modules are awesome.'); // Вывод: JS MODULES ARE AWESOME.
```

Именованный импорт позволяет нам импортировать только те части модуля, которые нам действительно нужны. Мы делаем это, явно называя их, в нашем случае указывая sayLoud в фигурных скобках в операторе импорта.

Групповой импорт

```
import * as i from './fun';
i.say('What?'); // Вывод: What?
i.sayLoud('Whoot!'); // Вывод: WHOOT!
```

Если мы хотим иметь все экспорты, это лучший способ. Используя синтаксис * as i, оператор импорта предоставляет нам объект i, который содержит все экспорты нашего модуля fun как свойства с соответствующими именами.

Пути

Имейте в виду, что вам нужно явно указывать пути импорта как относительные, даже если файл для импорта находится в той же директории, что и файл, в который вы импортируете с помощью ./. Например, импорты из путей без префиксов

```
import express from 'express';
```

...будут искаться в локальных и глобальных папках (директориях) node_modules и вызовут ошибку, если соответствующие модули не будут найдены.

Экспорт с использованием синтаксиса ES6

Рассмотрим другой эквивалентный пример, но с использованием ES6:

```
export function printHelloWorld() {
    console.log("Hello World!!!");
}
```

Глава 32.
Загрузка файлов

Примеры

Загрузка одного файла с использованием multer

Не забудьте:

- создать папку (директорию) для загрузки (в примере – `uploads`);
- установить `multer`: `npm i -S multer`.

server.js:

```
var express = require("express");
var multer = require('multer');
var app = express();
var fs = require('fs');
app.get('/', function(req, res) {
    res.sendFile(__dirname + "/index.html");
});
var storage = multer.diskStorage({
    destination: function (req, file, callback) {
        fs.mkdir('./uploads', function(err) {
            if(err) {
                console.log(err.stack);
            } else {
                callback(null, './uploads');
            }
        });
    },
    filename: function (req, file, callback) {
        callback(null, file.fieldname + '-' + Date.now());
    }
});
```

```
app.post('/api/file', function(req, res) {
    var upload = multer({ storage: storage }).single('userFile');
    upload(req, res, function(err) {
        if(err) {
            return res.end("Error uploading file.");
        }
        res.end("File is uploaded");
    });
});
app.listen(3000, function() {
    console.log("Working on port 3000");
});
```

index.html:

```
<form id="uploadForm" enctype="multipart/form-data" action="/api/file"
method="post">
    <input type="file" name="userFile" />
    <input type="submit" value="Upload File" name="submit">
</form>
```

Примечание: чтобы загрузить файл с сохранением расширения, можно использовать встроенную библиотеку Node.js – `path`.

Для этого подключите `path` в файл `server.js`:

```
var path = require('path');
```

и измените:

```
callback(null, file.fieldname + '-' + Date.now());
```

...добавив расширение файла следующим образом:

```
callback(null, file.fieldname + '-' + Date.now() + path.extname(file.
originalname));
```

Как фильтровать загрузку по расширению

В этом примере показано, как загружать файлы, разрешая только определенные расширения.

Например, только изображения. Просто добавьте к `var upload = multer({ storage: storage }).single('userFile');` условие `fileFilter`.

```
var upload = multer({
    storage: storage,
    fileFilter: function (req, file, callback) {
        var ext = path.extname(file.originalname);
```

```
        if(ext !== '.png' && ext !== '.jpg' && ext !== '.gif' && ext
!== '.jpeg') {
            return callback(new Error('Only images are allowed'));
        }
        callback(null, true);
    }
}).single('userFile');
```

Теперь можно загружать только изображения с расширениями png, jpg, gif или jpeg.

Использование модуля formidable

Установите модуль и прочтите документацию:

```
npm i formidable@latest
```

Ссылка на документацию: https://github.com/node-formidable/formidable.

Пример сервера на порту 8080:

```
var formidable = require('formidable'),
    http = require('http'),
    util = require('util');

http.createServer(function(req, res) {
    if (req.url == '/upload' && req.method.toLowerCase() == 'post') {
        // парсинг загрузки файла
        var form = new formidable.IncomingForm();
        form.parse(req, function(err, fields, files) {
            if (err)
                do-smth; // обработка ошибки
            // Копирование файла из временного места
            // var fs = require('fs');
```

```
            // fs.rename(file.path, <targetPath>, function (err) {...});
            // Отправка результата клиенту
            res.writeHead(200, {'content-type': 'text/plain'});
            res.write('received upload:\n\n');
            res.end(util.inspect({fields: fields, files: files}));
        });
        return;
    }
    // отображение формы для загрузки файла
    res.writeHead(200, {'content-type': 'text/html'});
    res.end(
            '<form action="/upload" enctype="multipart/form-data"
method="post">' +
        '<input type="text" name="title"><br>' +
        '<input type="file" name="upload" multiple="multiple"><br>' +
        '<input type="submit" value="Upload">' +
        '</form>'
    );
}).listen(8080);
```

Глава 33. Файловый ввод/вывод (Filesystem I/O)

Примечания

В Node.js ресурсоемкие операции, такие как ввод/вывод, выполняются *асинхронно*, но имеют *синхронный* аналог (например, существует `fs.readFile` и его аналог — `fs.readFileSync`). Поскольку Node.js однопоточен, следует быть осторожным при использовании синхронных операций, так как они блокируют весь процесс.

Если процесс блокируется синхронной операцией, весь цикл выполнения (включая цикл событий) останавливается. Это означает, что другой асинхронный код, включая события и обработчики событий, не будет выполняться, и ваша программа продолжит ждать завершения единственной блокирующей операции.

Существуют подходящие случаи для использования как синхронных, так и асинхронных операций, но нужно внимательно следить за тем, чтобы они использовались правильно.

Примеры

Запись в файл с использованием writeFile или writeFileSync

```
var fs = require('fs');
// Сохраните строку "Hello world!" в файле "hello.txt"
// в каталоге "/tmp" с использованием кодировки по умолчанию (utf8).
// Эта операция будет выполнена в фоновом режиме, и
// обратный вызов будет вызван, когда она будет завершена или завершится ошибкой.
fs.writeFile('/tmp/hello.txt', 'Hello world!', function(err) {
    // Если произошла ошибка, покажите ее и завершите выполнение
    if(err) return console.error(err);
    // Успешно записано в файл!
```

```
});
// Сохраните бинарные данные в файле с именем "binary.txt"
// в текущем каталоге. Эта операция также будет выполнена в фоновом
режиме.
var buffer = new Buffer([ 0x48, 0x65, 0x6c, 0x6c, 0x6f ]);
fs.writeFile('binary.txt', buffer, function(err) {
    // Если произошла ошибка, покажите ее и завершите выполнение
    if(err) return console.error(err);
    // Бинарное содержимое успешно записано в файл!
});
```

`fs.writeFileSync` ведет себя аналогично `fs.writeFile`, но не принимает обратный вызов, так как выполняется синхронно и поэтому блокирует основной поток. Большинство разработчиков Node.js предпочитают асинхронные варианты, которые практически не вызывают задержек в выполнении программы.

Примечание: блокировка основного потока считается плохой практикой в Node.js. Синхронные функции должны использоваться только при отладке или когда нет других вариантов.

```
// Записать строку в другой файл и установить режим доступа к файлу
на 0755
try {
    fs.writeFileSync('sync.txt', 'anni', { mode: 0o755 });
} catch(err) {
    // Произошла ошибка
    console.error(err);
}
```

Асинхронное чтение из файлов

Для всех файловых операций используйте модуль файловой системы:

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

С использованием кодировки

В этом примере читается файл `hello.txt` из каталога `/tmp`. Эта операция будет выполнена в фоновом режиме, а обратный вызов будет вызван по завершении или при ошибке:

```
fs.readFile('/tmp/hello.txt', { encoding: 'utf8' }, (err, content) =>
{
    // Если произошла ошибка, выведите ее и завершите выполнение
    if(err) return console.error(err);
    // Ошибки не произошло, content — это строка
    console.log(content);
});
```

Без использования кодировки

Считайте бинарный файл `binary.txt` из текущего каталога асинхронно в фоновом режиме. Обратите внимание, что мы не указываем опцию 'encoding' — это предотвращает декодирование содержимого в строку в Node.js:

```
fs.readFile('binary', (err, binaryContent) => {
    // Если произошла ошибка, выведите ее и завершите выполнение
    if(err) return console.error(err);
    // Ошибки не произошло, content - это Buffer, выводим его
    // в шестнадцатеричном формате.
    console.log(content.toString('hex'));
});
```

Относительные пути

Имейте в виду, что ваш скрипт может быть запущен с произвольным текущим рабочим каталогом. Чтобы обратиться к файлу относительно текущего скрипта, используйте `__dirname` или `__filename`:

```
fs.readFile(path.resolve(__dirname, 'someFile'), (err, binaryContent)
=> {
    // Остальная часть функции
});
```

Список содержимого каталога с использованием readdir или readdirSync

```
const fs = require('fs');
// Чтение содержимого каталога /usr/local/bin асинхронно.
// Обратный вызов будет вызван, когда операция завершится
// либо успешно, либо с ошибкой.
fs.readdir('/usr/local/bin', (err, files) => {
    // В случае ошибки покажите ее и завершите выполнение
    if(err) return console.error(err);
    // files - это массив, содержащий имена всех записей
    // в каталоге, за исключением '.' (сам каталог)
    // и '..' (родительский каталог).
    // Отображение содержимого каталога
    console.log(files.join(' '));
});
```

Синхронный вариант доступен как `readdirSync`, который блокирует основной поток и, следовательно, предотвращает выполнение асинхронного кода в то же время. Большинство разработчиков избегают синхронных функций ввода/вывода, чтобы улучшить производительность.

```
let files;
try {
    files = fs.readdirSync('/var/tmp');
} catch(err) {
    // Произошла ошибка
    console.error(err);
}
```

Использование генератора

```
const fs = require('fs');
// Итерация по всем элементам, полученным через
// выражения 'yield'
// Обратный вызов передается функции-генератору, потому что
// это требуется методом 'readdir'
function run(gen) {
    var iter = gen((err, data) => {
        if (err) { iter.throw(err); }
        return iter.next(data);
    });
    iter.next();
}
const dirPath = '/usr/local/bin';
// Выполнение функции-генератора
run(function* (resume) {
    // Эмитирует список файлов в каталоге из генератора
    var contents = yield fs.readdir(dirPath, resume);
    console.log(contents);
});
```

Синхронное чтение из файла

Для любых файловых операций вам понадобится модуль файловой системы:

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

Чтение строки

`fs.readFileSync` ведет себя аналогично `fs.readFile`, но не принимает функцию обратного вызова, так как выполняется синхронно и, следовательно, блокирует основной поток. Большинство разработчиков Node.js предпочитают асинхронные варианты, которые практически не вызывают задержек в выполнении программы.

Если указан параметр кодировки, будет возвращена строка, в противном случае будет возвращен `Buffer`.

```
// Чтение строки из другого файла синхронно
let content;
try {
    content = fs.readFileSync('sync.txt', { encoding: 'utf8' });
} catch(err) {
    // Произошла ошибка
    console.error(err);
}
```

Удаление файла с использованием unlink или unlinkSync

Удаление файла асинхронно:

```
var fs = require('fs');
fs.unlink('/path/to/file.txt', function(err) {
    if (err) throw err;
    console.log('file deleted');
});
```

Можно также удалить файл синхронно[1]:

```
var fs = require('fs');
fs.unlinkSync('/path/to/file.txt');
console.log('file deleted');
```

Чтение файла в Buffer с использованием потоков

Хотя чтение содержимого из файла уже является асинхронным при использовании метода `fs.readFile()`, иногда мы хотим получать данные в виде потока, а не просто в функции обратного вызова. Это позволяет передавать данные в другие места или обрабатывать их по мере поступления, а не все сразу в конце:

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

// Массив для хранения частей данных файла
let chunks = [];
// Эта переменная может использоваться для хранения окончательных данных
let fileBuffer;
// Чтение файла в stream.Readable
let fileStream = fs.createReadStream('text.txt');

// Произошла ошибка с потоком
```

[1] Избегайте синхронных методов, так как они блокируют весь процесс до завершения выполнения программы.

```
fileStream.once('error', (err) => {
    // Обязательно обработайте это правильно!
    console.error(err);
});

// Файл завершен чтением
fileStream.once('end', () => {
    // создание окончательных данных Buffer из частей данных;
    fileBuffer = Buffer.concat(chunks);
     // Конечно, вы можете сделать здесь все что вам нужно, например,
сгенерировать событие!
});
// Данные передаются из fileStream частями,
// эта функция обратного вызова будет выполняться для каждой части
fileStream.on('data', (chunk) => {
    chunks.push(chunk); // добавление части данных в массив
    // Мы можем выполнять действия с частичными данными, которые у нас
есть на данный момент!
});
```

Проверка прав доступа к файлу или директории

`fs.access()` определяет, существует ли путь и какие разрешения у пользователя есть для файла или директории по указанному пути. `fs.access` не возвращает результат; если ошибка не возвращается, путь существует и у пользователя есть необходимые права.

Режимы разрешений доступны как свойства объекта `fs`, `fs.constants`:

- `fs.constants.F_OK` — Имеет права на чтение/запись/выполнение (если режим не указан, это значение используется по умолчанию).
- `fs.constants.R_OK` — Имеет права на чтение.
- `fs.constants.W_OK` — Имеет права на запись.
- `fs.constants.X_OK` — Имеет права на выполнение (работает так же, как `fs.constants.F_OK` на Windows).

Асинхронно

```
var fs = require('fs');
var path = '/path/to/check';
// проверка прав на выполнение
fs.access(path, fs.constants.X_OK, (err) => {
    if (err) {
```

```
        console.log("%s doesn't exist", path);
    } else {
        console.log('can execute %s', path);
    }
});
// Проверка прав на чтение/запись
// При указании нескольких режимов разрешений
// каждый режим разделяется с помощью символа `|`
fs.access(path, fs.constants.R_OK | fs.constants.W_OK, (err) => {
    if (err) {
        console.log("%s doesn't exist", path);
    } else {
        console.log('can read/write %s', path);
    }
});
```

Синхронно

`fs.access` также имеет синхронную версию `fs.accessSync`. При использовании `fs.accessSync` необходимо заключить вызов функции в блок `try/catch`.

```
// Проверка права на запись
try {
    fs.accessSync(path, fs.constants.W_OK);
    console.log('can write %s', path);
} catch (err) {
    console.log("%s doesn't exist", path);
}
```

Избегание состояния гонки (race condition) при создании или использовании существующего каталога

Из-за асинхронной природы Node.js создание или использование директории, сначала:

- проверяя ее существование с помощью `fs.stat()`, затем
- создавая или используя ее в зависимости от результатов проверки существования,

может привести к состоянию гонки, если папка (директория) создается между моментом проверки и моментом создания. Метод ниже оборачивает `fs.mkdir()` и `fs.mkdirSync()` в обработчики ошибок, которые позволяют исключению проходить, если его код — EEXIST (уже существует). Если ошибка другая, например, EPERM (отказано в доступе), выбросите или передайте ошибку, как это делают нативные функции.

Асинхронная версия с `fs.mkdir()`

```
var fs = require('fs');
function mkdir (dirPath, callback) {
    fs.mkdir(dirPath, (err) => {
        callback(err && err.code !== 'EEXIST' ? err : null);
    });
}
mkdir('./existingDir', (err) => {
    if (err)
        return console.error(err.code);
    // Здесь можно сделать что-то с `./existingDir`
});
```

Синхронная версия с `fs.mkdirSync()`

```
function mkdirSync (dirPath) {
    try {
        fs.mkdirSync(dirPath);
    } catch(e) {
        if (e.code !== 'EEXIST') throw e;
    }
}
mkdirSync('./existing-dir');
// Здесь можно что-то сделать с `./existing-dir`
```

Проверка, существует ли файл или директория

Асинхронно

```
var fs = require('fs');
fs.stat('path/to/file', function(err) {
    if (!err) {
        console.log('file or directory exists');
    } else if (err.code === 'ENOENT') {
        console.log('file or directory does not exist');
    }
});
```

Синхронно

Здесь следует обернуть вызов функции в блок `try/catch` для обработки ошибок:

```
var fs = require('fs');
try {
    fs.statSync('path/to/file');
```

```
    console.log('file or directory exists');
} catch (err) {
    if (err.code === 'ENOENT') {
        console.log('file or directory does not exist');
    }
}
```

Клонирование файла с использованием потоков

Эта программа иллюстрирует, как можно скопировать файл с использованием потоков чтения и записи с помощью функций `createReadStream()` и `createWriteStream()`, предоставленных модулем файловой системы.

```
// Подключаем модуль файловой системы
var fs = require('fs');
/*
 Создаем поток чтения для файла в текущей директории (__dirname) с именем 'node.txt'
 Используем кодировку utf8
 Читаем данные кусками по 16 килобайт
*/
var readable = fs.createReadStream(__dirname + '/node.txt', { encoding: 'utf8', highWaterMark: 16 * 1024 });
// создаем поток записи
var writable = fs.createWriteStream(__dirname + '/nodeCopy.txt');

// Записываем каждый кусок данных в поток записи
readable.on('data', function(chunk) {
    writable.write(chunk);
});
```

Копирование файлов с использованием соединения потоков

Эта программа копирует файл с использованием потока чтения и записи с помощью функции `pipe()`, предоставленной классом потока.

```
// Подключаем модуль файловой системы
var fs = require('fs');
/*
 Создаем поток чтения для файла в текущей директории с именем 'node.txt'
```

```
 Используем кодировку utf8
 Читаем данные кусками по 16 килобайт
*/
var readable = fs.createReadStream(__dirname + '/node.txt', { encoding:
'utf8', highWaterMark: 16 * 1024 });
// создаем поток записи
var writable = fs.createWriteStream(__dirname + '/nodePipe.txt');
// используем pipe для копирования из потока чтения в поток записи
readable.pipe(writable);
```

Изменение содержимого текстового файла

Mongoose заменяем слово email на name в текстовом файле index.txt с использованием простого регулярного выражения replace(/email/gim, 'name'):

```
var fs = require('fs');
fs.readFile('index.txt', 'utf-8', function(err, data) {
    if (err) throw err;
    var newValue = data.replace(/email/gim, 'name');
    fs.writeFile('index.txt', newValue, 'utf-8', function(err, data) {
        if (err) throw err;
        console.log('Done!');
    });
});
```

Определение количества строк в текстовом файле

app.js

```
const readline = require('readline');
const fs = require('fs');
var file = 'path.to.file';
var linesCount = 0;
var rl = readline.createInterface({
    input: fs.createReadStream(file),
    output: process.stdout,
    terminal: false
});

rl.on('line', function (line) {
  linesCount++; // при каждом разрыве строки добавляем +1 к 'linesCount'
});
```

```
rl.on('close', function () {
    console.log(linesCount); // выводим результат при вызове события
'close'
});
```

Использование:

```
node app
```

Чтение файла построчно

app.js

```
const readline = require('readline');
const fs = require('fs');
var file = 'path.to.file';
var rl = readline.createInterface({
    input: fs.createReadStream(file),
    output: process.stdout,
    terminal: false
});
rl.on('line', function (line) {
    console.log(line); // вывод содержимого строки при каждом разры-
ве строки
});
```

Использование:

```
node app
```

Глава 34.
Начало работы с профилированием Node.js

Введение

Цель этой главы — познакомиться с профилированием приложения на Node.js и научиться анализировать результаты для обнаружения ошибок или утечек памяти. Запущенное приложение на Node.js — это процесс движка V8, который во многих аспектах схож с веб-сайтом, работающим в браузере. Мы можем получить все метрики, связанные с процессом веб-сайта, и для приложения Node.js.

Инструмент — Chrome DevTools или Chrome Inspector в сочетании с `node-inspector`.

Замечания

Иногда `node-inspector` не удается прикрепиться к процессу отладки Node.js, в результате чего вы не сможете установить точку останова в DevTools. Попробуйте несколько раз обновить вкладку DevTools и подождите несколько секунд, чтобы убедиться, что она находится в режиме отладки. Если это не поможет, перезапустите `node-inspector` из командной строки.

Примеры

Профилирование простого приложения на Node.js

Шаг 1. Установите пакет `node-inspector` глобально на вашем компьютере с помощью `npm`:

```
$ npm install -g node-inspector
```

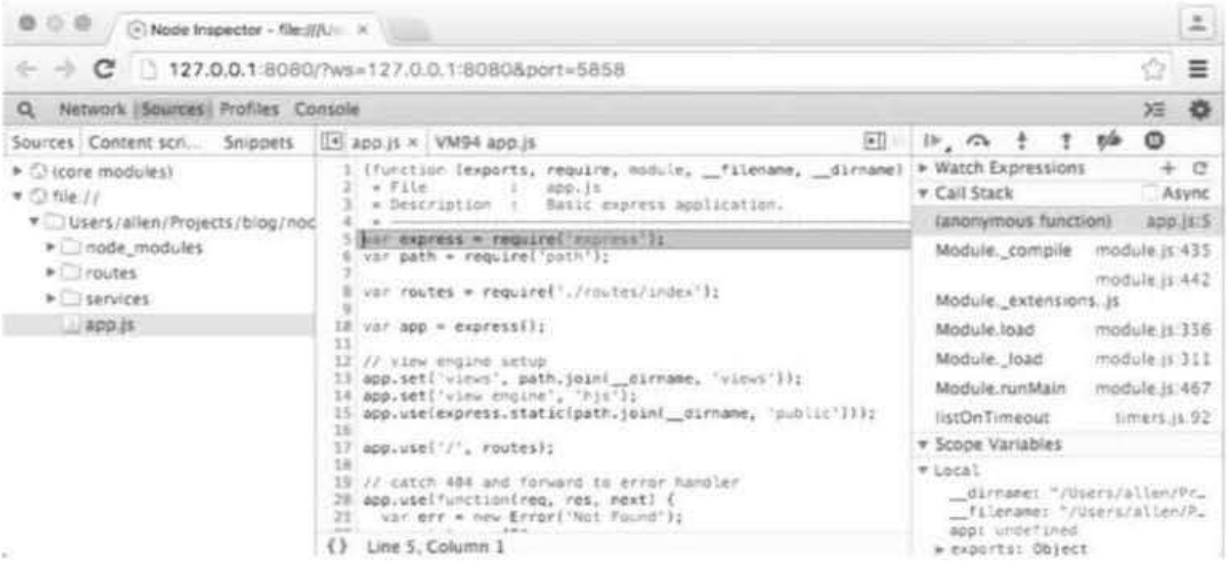
Шаг 2. Запустите сервер `node-inspector`:

```
$ node-inspector
```

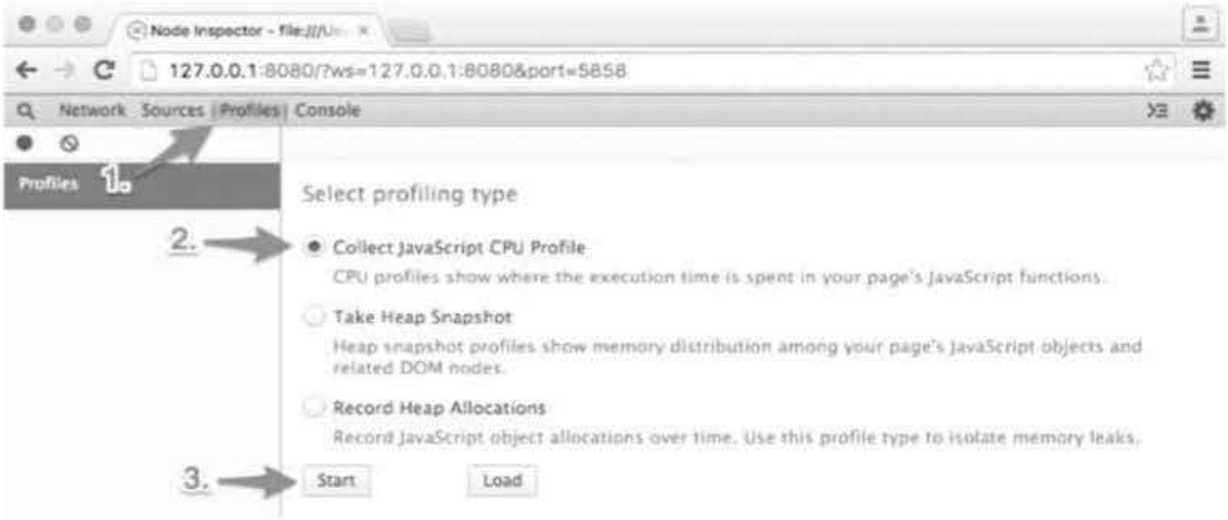
Шаг 3. Начните отладку вашего Node.js-приложения:

```
$ node --debug-brk your/short/node/script.js
```

Шаг 4. Откройте `http://127.0.0.1:8080/?port=5858` в браузере Chrome. Вы увидите интерфейс Chrome DevTools с исходным кодом вашего приложения на Node.js в левой панели. Так как мы использовали опцию `debug break` при отладке приложения, выполнение кода остановится на первой строке.



Шаг 5. Это самая простая часть — переключитесь на вкладку профилирования и начните профилирование приложения. Если вы хотите получить профиль для конкретного метода или потока, убедитесь, что выполнение кода приостановлено непосредственно перед выполнением этого участка кода.



Шаг 6. После того как вы записали ваш профиль CPU или снимок/снимок кучи, или выделение памяти кучи, вы можете просмотреть результаты в том же окне или сохранить их на локальный диск для дальнейшего анализа или сравнения с другими профилями.

Вы можете воспользоваться следующими статьями, чтобы узнать, как читать профили:

Reading CPU Profiles: https://commandlinefanatic.com/cgi-bin/showarticle.cgi?article=art037.

Chrome CPU profiler and Heap profiler: https://developer.chrome.com/docs/devtools?hl=ru.

Глава 35.
Хороший стиль кодирования

Примечания

Мы рекомендуем начинающим разработчикам использовать хороший стиль кодирования. И если кто-то может предложить лучший способ, будем рады любым предложениям. Заранее спасибо.

Примеры

Базовая программа для регистрации

На этом примере мы рассмотрим, как разделить код Node.js на разные модули/папки для лучшего понимания. Следование этой технике облегчает другим разработчикам понимание кода, так как они могут сразу обратиться к соответствующему файлу вместо того, чтобы просматривать весь код. Основное преимущество данного подхода проявляется, когда вы работаете в команде, и новый разработчик присоединяется на более позднем этапе — ему будет проще разобраться с кодом.

index.js — этот файл управляет подключением сервера.

```
// Импорт библиотек
var express = require('express'),
    session = require('express-session'),
    mongoose = require('mongoose'),
    request = require('request');

// Импорт пользовательских модулей
var userRoutes = require('./app/routes/userRoutes');
var config = require('./app/config/config');

// Подключение к MongoDB
mongoose.connect(config.getDBString());
```

```
// Создание нового приложения Express и его настройка
var app = express();

// Настройка маршрутов
app.use(config.API_PATH, userRoutes());

// Запуск сервера
app.listen(config.PORT);
console.log('Server started at - ' + config.URL + ":" + config.PORT);
```

config.js — этот файл управляет всеми параметрами конфигурации, которые останутся неизменными на протяжении всего проекта.

```
var config = {
    VERSION: 1,
    BUILD: 1,
    URL: 'http://127.0.0.1',
    API_PATH : '/api',
    PORT : process.env.PORT || 8080,
    DB : {
        // Конфигурация MongoDB
        HOST : 'localhost',
        PORT : '27017',
        DATABASE : 'db'
    },
    /*
    * Получить строку подключения к базе данных MongoDB
    */
    getDBString : function(){
        return 'mongodb://' + this.DB.HOST + ':' + this.DB.PORT + '/'
+ this.DB.DATABASE;
    },
    /*
    * Получить http URL
    */
    getHTTPUrl : function(){
        return 'http://' + this.URL + ":" + this.PORT;
    }
}

module.exports = config;
```

user.js — модельный файл, в котором определена схема.

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
```

```
// Схема для пользователя
var UserSchema = new Schema({
    name: {
        type: String,
        // required: true
    },
    email: {
        type: String
    },
    password: {
        type: String,
        // required: true
    },
    dob: {
        type: Date,
        // required: true
    },
    gender: {
        type: String, // Male/Female
        // required: true
    }
});

// Определение модели для пользователя
var User;
if(mongoose.models.User)
    User = mongoose.model('User');
else
    User = mongoose.model('User', UserSchema);

// Экспорт модели User
module.exports = User;
```

userController — этот файл содержит функцию для регистрации пользователя.

```
var User = require('../models/user');
var crypto = require('crypto');

// Контроллер для пользователя
var UserController = {
    // Создание пользователя
    create: function(req, res){
        var repassword = req.body.repassword;
        var password = req.body.password;
```

```
var userEmail = req.body.email;

// Проверка, существует ли уже адрес электронной почты
User.find({"email": userEmail}, function(err, usr){
    if(usr.length > 0){
        // Электронная почта уже существует
        res.json('Email already exists');
        return;
    }
    else
    {
        // Новый Email
        // Проверка на совпадение паролей
        if(password != repassword){
            res.json('Passwords do not match');
            return;
        }

        // Генерация хеша пароля на основе sha1
        var shasum = crypto.createHash('sha1');
        shasum.update(req.body.password);
        var passwordHash = shasum.digest('hex');

        // Создание пользователя
        var user = new User();
        user.name = req.body.name;
        user.email = req.body.email;
        user.password = passwordHash;
        user.dob = Date.parse(req.body.dob) || "";
        user.gender = req.body.gender;

        // Валидация пользователя
        user.validate(function(err){
            if(err){
                res.json(err);
                return;
            }else{
                // Наконец, сохранение пользователя
                user.save(function(err){
                    if(err)
                    {
                        res.json(err);
                        return;
                    }
```

```
                                // Удаление пароля перед отправкой данных
                                    пользователя
                                user.password = undefined;
                                res.json(user);
                                return;
                            });
                        }
                    });
                }
            });
    }
}

module.exports = UserController;
```

userRoutes.js — это маршрут для userController.

```
var express = require('express');
var UserController = require('../controllers/userController');

// Маршруты для пользователя
var UserRoutes = function(app)
{
    var router = express.Router();
    router.route('/users')
        .post(UserController.create);
    return router;
}

module.exports = UserRoutes;
```

Приведенный выше пример может показаться большим, но если начинающий в Node.js с небольшими знаниями Express подробно разберется в нем, он найдет его простым и действительно полезным.

Глава 36.
Корректное завершение

Примеры

Корректное завершение — SIGTERM

Используя `server.close()` и `process.exit()`, мы можем перехватить исключение сервера и выполнить его корректное завершение.

```
var http = require('http');
var server = http.createServer(function (req, res) {
    setTimeout(function () { // симуляция долгого запроса
        res.writeHead(200, {'Content-Type': 'text/plain'});
        res.end('Hello World\n');
    }, 4000);
}).listen(9090, function (err) {
    console.log('listening http://localhost:9090/');
    console.log('pid is ' + process.pid);
});

process.on('SIGTERM', function () {
    server.close(function () {
        process.exit(0);
    });
});
```

Глава 37.
Grunt

Примечания

Дополнительные ресурсы:

Руководство Installing Grunt (https://gruntjs.com/installing-grunt) содержит подробную информацию об установке конкретных версий Grunt и grunt-cli для использования в продакшене или в процессе разработки.

Руководство Configuring Tasks (https://gruntjs.com/configuring-tasks) предоставляет детальное объяснение того, как настраивать задачи, цели, опции и файлы в Gruntfile, а также объясняет шаблоны, шаблоны с глобальными паттернами и импорт внешних данных.

Руководство Creating Tasks (https://gruntjs.com/creating-tasks) описывает различия между типами задач в Grunt и показывает примеры задач и конфигураций.

Примеры

Введение в Grunt.js

Grunt — это JavaScript Task Runner, используемый для автоматизации повторяющихся задач, таких как минификация, компиляция, модульное тестирование, проверка кода и т. д.

Для начала работы вам нужно установить интерфейс командной строки (CLI) Grunt глобально.

```
npm install -g grunt-cli
```

Подготовка нового проекта Grunt: типичная настройка включает добавление двух файлов в ваш проект: `package.json` и `Gruntfile`.

`package.json`: этот файл используется `npm` для хранения метаданных о проектах, опубликованных как модули `npm`. В этом файле вы перечислите Grunt и необходимые для вашего проекта плагины Grunt как `devDependencies`.

`Gruntfile`: этот файл называется `Gruntfile.js` и используется для настройки или определения задач и загрузки плагинов Grunt.

Пример package.json:

```
{
    "name": "my-project-name",
    "version": "0.1.0",
    "devDependencies": {
        "grunt": "~0.4.5",
        "grunt-contrib-jshint": "~0.10.0",
        "grunt-contrib-nodeunit": "~0.4.1",
        "grunt-contrib-uglify": "~0.5.0"
    }
}
```

Пример Gruntfile:

```
module.exports = function(grunt) {
    // Настройка проекта.
    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json'),
        uglify: {
            options: {
                    banner: '/*! <%= pkg.name %> <%= grunt.template.
today("yyyy-mm-dd") %> */\n'
            },
            build: {
                src: 'src/<%= pkg.name %>.js',
                dest: 'build/<%= pkg.name %>.min.js'
            }
        }
    });
    // Загрузка плагина, который предоставляет задачу "uglify".
    grunt.loadNpmTasks('grunt-contrib-uglify');
    // Задача по умолчанию.
    grunt.registerTask('default', ['uglify']);
};
```

Установка плагинов Grunt

Добавление зависимости

Чтобы использовать плагин Grunt, сначала нужно добавить его как зависимость к вашему проекту. Возьмем в качестве примера плагин `jshint`.

```
npm install grunt-contrib-jshint --save-dev
```

Опция --save-dev используется для добавления плагина в `package.json`, чтобы он всегда устанавливался после команды `npm install`.

Загрузка плагина

Вы можете загрузить ваш плагин в файле `Gruntfile` с помощью `loadNpmTasks`.

```
grunt.loadNpmTasks('grunt-contrib-jshint');
```

Настройка задачи

Вы настраиваете задачу в `Gruntfile`, добавляя свойство `jshint` в объект, передаваемый `grunt.initConfig`.

```
grunt.initConfig({
```

```
    jshint: {
        all: ['Gruntfile.js', 'lib/**/*.js', 'test/**/*.js']
    }
});
```

Не забывайте, что вы можете добавлять другие свойства для других плагинов, которые используете.

Запуск задачи

Чтобы запустить задачу с плагином, вы можете использовать командную строку:

```
grunt jshint
```

...или вы можете добавить `jshint` к другой задаче:

```
grunt.registerTask('default', ['jshint']);
```

Задача по умолчанию запускается с помощью команды `grunt` в терминале без каких-либо дополнительных опций.

Глава 38.
Совет по упрощению работы

Добавление новых расширений в require()

Вы можете добавить новые расширения в `require()`, расширяя `require.extensions`.

Пример для XML:

```
// Добавляем .xml для require()
require.extensions['.xml'] = (module, filename) => {
    const fs = require('fs')
    const xml2js = require('xml2js')

    module.exports = (callback) => {
        // Читаем требуемый файл.
        fs.readFile(filename, 'utf8', (err, data) => {
            if (err) {
                callback(err)
                return
            }
            // Парсим его.
            xml2js.parseString(data, (err, result) => {
                callback(null, result)
            })
        })
    }
}
```

Если содержимое файла `hello.xml` следующее:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
    <bar>baz</bar>
    <qux />
</foo>
```

...вы можете прочитать и распарсить его через `require()`:

```
require('./hello')((err, xml) => {
    if (err)
        throw err;
    console.log(err);
})
```

Это выведет `{ foo: { bar: [ 'baz' ], qux: [ '' ] } }`.

Глава 39.
Обработка POST-запросов в Node.js

Замечания

Node.js использует потоки для обработки входящих данных.
Цитата из документации:

> Поток (stream) — это абстрактный интерфейс для работы с потоковыми данными в Node.js. Модуль stream предоставляет базовый API, который облегчает создание объектов, реализующих интерфейс потока.

Источник: https://nodejs.org/api/stream.html#stream_stream.

Для обработки тела запроса POST используйте объект запроса, который является читаемым потоком. Потоки данных передаются в виде событий `data` на объекте запроса.

```
request.on('data', chunk => {
    buffer += chunk;
});
request.on('end', () => {
    // Тело POST-запроса теперь доступно как `buffer`
});
```

Просто создайте пустую строку буфера и добавляйте в нее данные буфера по мере их получения через события `data`.

ПРИМЕЧАНИЕ

1. Данные буфера, полученные при событиях `data`, имеют тип `Buffer`.

2. Создавайте новую строку буфера для сбора буферизованных данных из событий `data` для каждого запроса, то есть создавайте строку буфера внутри обработчика запроса.

Примеры

Пример сервера на Node.js, который обрабатывает только POST-запросы:

```
'use strict';

const http = require('http');

const PORT = 8080;
const server = http.createServer((request, response) => {
    let buffer = '';
    request.on('data', chunk => {
        buffer += chunk;
    });
    request.on('end', () => {
        const responseString = `Received string ${buffer}`;
        console.log(`Responding with: ${responseString}`);
        response.writeHead(200, "Content-Type: text/plain");
        response.end(responseString);
    });
}).listen(PORT, () => {
    console.log(`Listening on ${PORT}`);
});
```

Глава 40.
Как загружаются модули

Примеры

Режим глобальной установки

Если вы установили Node с использованием стандартного каталога, при работе в глобальном режиме NPM устанавливает пакеты в `/usr/local/lib/node_modules`. Если вы введете в оболочке следующую команду, NPM будет искать, загружать и устанавливать последнюю версию пакета с именем `sax` в каталоге `/usr/local/lib/node_modules/express`:

```
$ npm install -g express
```

Убедитесь, что у вас есть права доступа к этой папке. Данные модули будут доступны для всех процессов Node, которые будут выполняться на конкретной машине.

При установке в локальном режиме NPM загрузит и установит модули в текущем рабочем каталоге, создав новую папку с именем `node_modules`. Например, если вы находитесь в каталоге `/home/user/apps/my_app`, то будет создана новая папка `/home/user/apps/my_app/node_modules`, если она еще не существует.

Загрузка модулей

Когда мы ссылаемся на модуль в коде, Node сначала ищет папку `node_module` внутри папки, указанной в инструкции `require`. Если имя модуля не является относительным и не относится к основным модулям, Node попробует найти его внутри папки `node_modules` в текущем каталоге. Например, если вы выполните следующее, Node попытается найти файл `./node_modules/myModule.js`:

```
var myModule = require('myModule.js');
```

Если Node не удается найти файл, он будет искать его в родительской папке `../node_modules/myModule.js`. Если поиск снова не удастся, он продолжит поиск в родительских папках, пока не достигнет корня или не найдет требуемый модуль.

Вы также можете опустить расширение .js, в этом случае Node добавит расширение .js и будет искать файл.

Загрузка модуля из папки (директории)

Вы можете использовать путь к папке для загрузки модуля следующим образом:

```
var myModule = require('./myModuleDir');
```

Если вы это сделаете, Node будет искать внутри этой папки. Node будет считать, что эта папка является пакетом, и попытается найти определение пакета. Это определение пакета должно быть файлом с именем package.json. Если в этой папке нет файла определения пакета с именем package.json, точка входа пакета будет считаться по умолчанию index.js, и Node будет искать файл по пути ./myModuleDir/index.js.

Последним вариантом, если модуль не найден ни в одной из папок, будет глобальная папка установки модулей.

Глава 41.
HTTP

Примеры

HTTP-сервер

Приведем простой пример HTTP-сервера. Напишите следующий код в файл `http_server.js`:

```
var http = require('http');
var httpPort = 80;
http.createServer(handler).listen(httpPort, start_callback);

function handler(req, res) {
    var clientIP = req.connection.remoteAddress;
    var connectUsing = req.connection.encrypted ? 'SSL' : 'HTTP';
    console.log('Request received: ' + connectUsing + ' ' + req.method
+ ' ' + req.url);
    console.log('Client IP: ' + clientIP);
    res.writeHead(200, "OK", {'Content-Type': 'text/plain'});
    res.write("OK");
    res.end();
    return;
}

function start_callback(){
    console.log('Start HTTP on port ' + httpPort)
}
```

Затем из расположения вашего файла `http_server.js` выполните команду:

```
node http_server.js
```

Вы должны увидеть следующий результат:

```
> Start HTTP on port 80
```

Теперь вам нужно протестировать сервер, открыв интернет-браузер и перейдя по следующему URL:

```
http://127.0.0.1:80
```

Если ваша машина работает под управлением Linux-сервера, вы можете протестировать его следующим образом:

```
curl 127.0.0.1:80
```

Вы должны увидеть следующий результат:

```
ok
```

В вашей консоли, где запущено приложение, вы увидите такие результаты:

```
> Request received: HTTP GET /
> Client IP: ::ffff:127.0.0.1
```

HTTP-клиент

Простой пример HTTP-клиента:

Напишите следующий код в файл `http_client.js`:

```
var http = require('http');
var options = {
    hostname: '127.0.0.1',
    port: 80,
    path: '/',
    method: 'GET'
};
var req = http.request(options, function(res) {
    console.log('STATUS: ' + res.statusCode);
    console.log('HEADERS: ' + JSON.stringify(res.headers));
    res.setEncoding('utf8');
    res.on('data', function (chunk) {
        console.log('Response: ' + chunk);
    });
    res.on('end', function (chunk) {
        console.log('Response ENDED');
    });
});
req.on('error', function(e) {
    console.log('problem with request: ' + e.message);
});
req.end();
```

Затем из расположения вашего файла `http_client.js` выполните команду:

```
node http_client.js
```

Вы должны увидеть следующий результат:

```
> STATUS: 200
> HEADERS: {"content-type":"text/plain","date":"Thu, 21 Jul 2016
11:27:17 GMT","connection":"close","transfer-encoding":"chunked"}
> Response: OK
> Response ENDED
```

Примечание: этот пример зависит от HTTP-сервера.

Глава 42.
Установка Node.js

Примеры

Установка Node.js на Ubuntu

Использование пакетного менеджера apt

```
sudo apt-get update
sudo apt-get install nodejs
sudo apt-get install npm
sudo ln -s /usr/bin/nodejs /usr/bin/node

# версии node и npm в apt устарели. Вот как можно их обновить:
sudo npm install -g npm
sudo npm install -g n
sudo n stable # (или lts, или конкретную версию)
```

Использование последней или конкретной версии (например LTS 6.x) непосредственно из nodesource

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -

apt-get install -y nodejs
```

Также для правильной установки глобальных модулей npm настройте личный каталог для них (это устраняет необходимость в sudo и исключает ошибки EACCES):

```
mkdir ~/.npm-global
echo "export PATH=~/.npm-global/bin:$PATH" >> ~/.profile
source ~/.profile
npm config set prefix '~/.npm-global'
```

Установка Node.js на Windows

Стандартная установка

Все бинарные файлы, установщики и исходные файлы Node.js можно скачать здесь: https://nodejs.org/en/download/package-manager.

Вы можете загрузить только исполняемый файл `node.exe` или использовать установщик для Windows (`.msi`), который также установит `npm`, рекомендуемый пакетный менеджер для Node.js, и настроит пути.

Установка с помощью пакетного менеджера

Вы также можете установить Node.js с помощью пакетного менеджера `Chocolatey` (система автоматизации управления ПО).

```
# choco install nodejs.install
```

Более подробную информацию о текущей версии можно найти в репозитории choco здесь: https://community.chocolatey.org/packages/nodejs.install.

Использование Node Version Manager (nvm)

Node Version Manager, или `nvm`, — это bash-скрипт, который упрощает управление несколькими версиями Node.js.

Для установки nvm используйте предоставленный установочный скрипт:

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/
install.sh | bash
```

Для Windows существует пакет nvm-windows с установщиком. На этой странице GitHub (https://github.com/coreybutler/nvm-windows) вы найдете подробности об установке и использовании пакета nvm-windows.

После установки nvm выполните команду «nvm on» из командной строки. Это позволит nvm управлять версиями Node.

Примечание: возможно, вам потребуется перезапустить терминал, чтобы он распознал только что установленную команду nvm.

Затем установите последнюю версию Node:

```
$ nvm install node
```

Вы также можете установить конкретную версию Node, указав основные, второстепенные и/или патч-версии:

```
$ nvm install 6
$ nvm install 4.2
```

Чтобы просмотреть список доступных для установки версий:

```
$ nvm ls-remote
```

Вы можете переключаться между версиями, указывая версию так же, как и при установке:

```
$ nvm use 5
```

Вы можете установить конкретную версию Node, которая будет использоваться по умолчанию, введя:

```
$ nvm alias default 4.2
```

Чтобы отобразить список версий Node, установленных на вашей машине, введите:

```
$ nvm ls
```

Чтобы использовать версии Node, специфичные для проекта, вы можете сохранить версию в файле .nvmrc. Таким образом, после начала работы над другим проектом будет меньше ошибок после его получения из репозитория.

```
$ echo "4.2" > .nvmrc
$ nvm use
```

Вывод:

```
Found '/path/to/project/.nvmrc' with version <4.2>
Now using node v4.2 (npm v3.7.3)
```

Когда Node установлен через nvm, вам не нужно использовать sudo для установки глобальных пакетов, так как они устанавливаются в домашнюю папку. Таким образом, `npm i -g http-server` работает без каких-либо ошибок разрешения.

Установка Node.js из исходных кодов с помощью пакетного менеджера APT

Необходимые компоненты:

```
sudo apt-get install build-essential
sudo apt-get install python
[optional]
sudo apt-get install git
```

Получение исходного кода и сборка:

```
cd ~
git clone https://github.com/nodejs/node.git
```

ИЛИ для последней версии LTS Node.js 6.10.2:

```
cd ~
wget https://nodejs.org/dist/v6.3.0/node-v6.10.2.tar.gz
tar -xzvf node-v6.10.2.tar.gz
```

Перейдите в каталог с исходными файлами, например:

```
cd ~/node-v6.10.2
```

Выполните:

```
./configure

make

sudo make install
```

Установка Node.js на Mac с помощью пакетного менеджера

Homebrew

Вы можете установить Node.js с помощью пакетного менеджера `Homebrew`.
Начните с обновления `brew`:

```
brew update
```

Возможно, вам потребуется изменить разрешения или пути. Лучше выполнить это перед продолжением:

```
brew doctor
```

Далее вы можете установить Node.js, запустив

```
brew install node
```

После установки Node.js вы можете проверить установленную версию, запустив

```
node -v
```

Macports

Вы также можете установить node.js через `Macports`.
Сначала обновите `Macports`, чтобы убедиться, что используются последние версии пакета:

```
sudo port selfupdate
```

Затем установите `nodejs` и `npm`:

```
sudo port install nodejs npm
```

Теперь вы можете запускать `node` через командную строку, просто вводя `node`. Также можно проверить текущую версию `node` с помощью:

```
node -v
```

Установка с помощью MacOS X Installer

Вы можете найти установочные пакеты на странице загрузки Node.js (https://nodejs.org/en/download/package-manager).

Обычно Node.js рекомендует две версии — LTS (долгосрочная поддержка) и текущую версию (последний релиз). Если вы новичок в Node, выберите LTS и нажмите кнопку «Macintosh Installer», чтобы скачать пакет.

Если вы хотите найти другие версии Node.js, перейдите по этой ссылке: https://nodejs.org/en/about/previous-releases.

Выберите нужный релиз и нажмите «Скачать».

На странице загрузки найдите файл с расширением .pkg.

После того как вы скачали файл (с расширением .pkg, конечно), дважды щелкните по нему, чтобы установить. Установщик включает в себя Node.js и `npm`, по умолчанию пакет установит оба, но вы можете настроить, что именно установить, нажав кнопку «Customize» на шаге «Installation Type». В остальном следуйте инструкциям по установке, они довольно просты.

Проверка установки Node

Откройте терминал (если вы не знаете, как его открыть, посмотрите эту статью на wikihow: https://www.wikihow.com/Get-to-the-Command-Line-on-a-Mac).

В терминале введите `node --version` и нажмите Enter. Если Node установлен, ваш терминал будет выглядеть следующим образом:

```
$ node --version
v7.2.1
```

v7.2.1 — это версия вашего Node.js. Если вместо этого вы получите сообщение «command not found: node», значит, возникла проблема с установкой.

Установка Node.js на Raspberry PI

Чтобы установить версию v6.x, обновите пакеты:

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
```

Использование пакетного менеджера apt:

```
sudo apt-get install -y nodejs
```

Установка с помощью Node Version Manager в Fish Shell с Oh My Fish

Node Version Manager (`nvm`) значительно упрощает управление версиями Node.js, их установку и устраняет необходимость в использовании sudo при работе с пакетами (например, `npm install ...`).

Fish Shell (`fish`) — это «умная и удобная командная оболочка для OS X, Linux и других систем», которая популярна среди программистов как альтернатива обычным оболочкам, таким как bash.

Наконец, Oh My Fish (`omf`) позволяет настраивать и устанавливать пакеты в оболочке Fish.

Этот раздел предполагает, что вы уже используете Fish в качестве оболочки.

Установите nvm:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.4/
install.sh | bash
```

Установите Oh My Fish:

```
curl -L https://github.com/oh-my-fish/oh-my-fish/raw/master/bin/
install | fish
```

(**Примечание:** вам будет предложено перезапустить терминал на данном этапе. Сделайте это).

Установите плагин nvm для Oh My Fish

Мы установим плагин `nvm` через Oh My Fish, чтобы использовать возможности `nvm` в оболочке Fish:

```
omf install nvm
```

Установка Node.js с помощью Node Version Manager

Теперь вы готовы использовать `nvm`. Вы можете установить и использовать любую версию Node.js по вашему выбору. Примеры:

- Установить последнюю версию Node:

```
nvm install node
```

- Установить конкретную версию, например 6.3.1:

```
nvm install 6.3.1
```

- Показать список установленных версий:

```
nvm ls
```

- Переключиться на ранее установленную версию, например 4.3.1:

```
nvm use 4.3.1
```

Заключительные примечания

Помните, что при использовании этого метода больше не нужно задействовать sudo при работе с Node.js! Версии Node, пакеты и прочее устанавливаются в вашем домашнем каталоге.

Установка Node.js из исходников на CentOS, RHEL и Fedora

Необходимые компоненты:

- git
- clang и clang++ 3.4^ или gcc и g++ 4.8^

- Python 2.6 или 2.7
- GNU Make 3.81^

Получение исходного кода:

Node.js v6.x LTS:

```
git clone -b v6.x https://github.com/nodejs/node.git
```

Node.js v7.x:

```
git clone -b v7.x https://github.com/nodejs/node.git
```

Сборка:

```
cd node
./configure
make -jX
su -c make install
```

X — количество ядер процессора, что значительно ускоряет сборку.

Очистка [по желанию]:

```
cd
rm -rf node
```

Установка Node.js с помощью n

Для начала есть удобный скрипт для настройки n на вашей системе. Просто выполните:

```
curl -L https://git.io/n-install | bash
```

...чтобы установить n. Затем установите бинарные файлы различными способами:

Последняя версия:

```
n latest
```

Стабильная версия:

```
n stable
```

LTS-версия:

```
n lts
```

Любая другая версия:

```
n <version>
```

Например:

```
n 4.4.7
```

Если эта версия уже установлена, данная команда активирует эту версию.

Переключение версий

n сам по себе отобразит список установленных бинарных файлов. Используйте клавиши «вверх» и «вниз», чтобы найти нужный, и нажмите Enter, чтобы активировать его.

Глава 43.
Взаимодействие с консолью

Синтаксис

- `console.log([data][, ...])`
- `console.error([data][, ...])`
- `console.time(label)`
- `console.timeEnd(label)`

Примеры

Логирование

Модуль Console

Подобно среде браузера JavaScript, Node.js предоставляет модуль `console`, который обеспечивает простые возможности для логирования и отладки.

Самыми важными методами, предоставляемыми модулем `console`, являются `console.log`, `console.error` и `console.time`. Но также существуют и другие, такие как `console.info`.

console.log

Параметры будут выведены на стандартный вывод (stdout) с новой строки.

```
console.log('Hello World');
```

```
> console.log('Hello World')
Hello World
```

console.error

Параметры будут выведены на стандартный поток ошибок (`stderr`) с новой строки.

```
console.error('Oh, sorry, there is an error.');
```

```
> console.error("Oh, sorry, error");
Oh, sorry, error
```

console.time, console.timeEnd

`console.time` запускает таймер с уникальной меткой, который можно использовать для вычисления продолжительности операции.

Когда вы вызываете `console.timeEnd` с той же меткой, таймер останавливается и выводит время выполнения в миллисекундах в `stdout`.

```
console.time('label');
// Код, который нужно измерить
console.timeEnd('label');
```

```
> console.time("label");
undefined
> console.timeEnd("label");
label: 9297.320ms
```

Модуль Process

Можно использовать модуль `process` для прямой записи в стандартный вывод консоли. Для этого существует метод `process.stdout.write`.

В отличие от `console.log`, этот метод не добавляет новую строку перед вашим выводом.

Таким образом, в следующем примере метод вызывается два раза, но новая строка не добавляется между их выводами.

```
> process.stdout.write("123");process.stdout.write("456");
123456true
```

```
process.stdout.write('First part of the message');
process.stdout.write(' continues without a new line.');
```

Форматирование

Можно использовать управляющие коды терминала (`control codes`) для выполнения определенных команд, таких как изменение цвета или позиционирование курсора.

```
> console.log("\033[31mThis will be red");
This will be red
```

Общее

Эффект	Код
Сброс	\033[0m
Высокая яркость	\033[1m
Подчеркивание	\033[4m
Инверсия	\033[7m

Цвет шрифта

Эффект	Код
Черный	\033[30m
Красный	\033[31m
Зеленый	\033[32m
Желтый	\033[33m
Синий	\033[34m
Magenta (Фиолетовый)	\033[35m
Cyan (Голубой)	\033[36m
Белый	\033[37m

Цвет фона

Эффект	Код
Черный	\033[40m
Красный	\033[41m
Зеленый	\033[42m
Желтый	\033[43m
Синий	\033[44m
Magenta (Фиолетовый)	\033[45m
Cyan (Голубой)	\033[46m
Белый	\033[47m

Глава 44.
Постоянное выполнение Node.js-приложения

Примеры

Использование PM2 в качестве менеджера процессов

PM2 позволяет запускать ваши скрипты Node.js бесконечно. В случае сбоя вашего приложения PM2 автоматически перезапустит его.

Установите PM2 глобально для управления вашими экземплярами Node.js:

```
npm install pm2 -g
```

Перейдите в директорию, в которой находится ваш скрипт Node.js, и выполните следующую команду каждый раз, когда вы хотите запустить экземпляр Node.js, который будет отслеживаться PM2:

```
pm2 start server.js --name "app1"
```

Полезные команды для мониторинга процессов

1. Список всех экземпляров Node.js, управляемых PM2:

```
pm2 list
```

2. Остановка конкретного экземпляра Node.js:

```
pm2 stop <instance name>
```

3. Удаление конкретного экземпляра Node.js:

```
pm2 delete <instance name>
```

4. Перезапуск конкретного экземпляра Node.js:

```
pm2 restart <instance name>
```

5. Мониторинг всех экземпляров Node.js:

```
pm2 monit
```

6. Остановка PM2:

```
pm2 kill
```

7. В отличие от `restart`, который останавливает и перезапускает процесс, `reload` обеспечивает перезагрузку без простоя:

```
pm2 reload <instance name>
```

8. Просмотр логов

```
pm2 logs <instance_name>
```

Запуск и остановка daemon Forever

Чтобы запустить процесс:

```
forever start index.js

warn: --minUptime not set. Defaulting to: 1000ms
warn: --spinSleepTime not set. Your script will exit if it does not
stay up for at least 1000ms
info: Forever processing file: index.js
```

Список запущенных экземпляров Forever:

```
$ forever list

info: Forever processes running
|data: | index | uid | command | script |forever pid|id | logfile |
uptime |
|------|-------|-----|-----------------|-------------|-----------|--
---|--------------------|-------------|
|data: | [0] |f4Kt |/usr/bin/nodejs | src/index.js|2131 |2146|/root/.
forever/f4Kt.log | 0:0:0:11.485 |
```

Остановка первого процесса:

```
forever stop 0

forever stop 2146

forever stop --uid f4Kt

forever stop --pidFile 2131
```

Непрерывная работа с помощью nohup

Альтернативой Forever на Linux является nohup.

Чтобы запустить экземпляр nohup:

1. Перейдите в папку, где находится app.js, или в папку www.
2. Выполните команду:

```
nohup nodejs app.js &
```

Чтобы завершить процесс:

1. Выполните:

```
ps -ef | grep nodejs
```

2. Завершите процесс:

```
kill -9 <the process number>
```

Управление процессами с помощью Forever

Установка

```
npm install forever -g
cd /node/project/directory
```

Использование

```
forever start app.js
```

Глава 45.
Фреймворк Koa v2

Примеры

Пример Hello World

```
const Koa = require('koa')

const app = new Koa()

app.use(async ctx => {
    ctx.body = 'Hello World'
})

app.listen(8080)
```

Обработка ошибок с использованием middleware (промежуточного ПО)

```
app.use(async (ctx, next) => {
    try {
        await next() // попытка вызвать следующий middleware
    } catch (err) {
         handleError(err, ctx) // определите свою собственную функцию
обработки ошибок
    }
})
```

Глава 46.
Lodash

Введение

Lodash — это удобная JavaScript-библиотека утилит.

Примеры

Фильтрация коллекции

Ниже приведен пример различных способов фильтрации массива объектов с использованием lodash.

```
let lodash = require('lodash');
var countries = [
    {"key": "DE", "name": "Deutschland", "active": false},
    {"key": "ZA", "name": "South Africa", "active": true}
];

var filteredByFunction = lodash.filter(countries, function (country) {
    return country.key === "DE";
});
// => [{"key": "DE", "name": "Deutschland"}];

var filteredByObjectProperties = lodash.filter(countries, { "key":
"DE" });
// => [{"key": "DE", "name": "Deutschland"}];

var filteredByProperties = lodash.filter(countries, ["key", "ZA"]);
// => [{"key": "ZA", "name": "South Africa"}];

var filteredByProperty = lodash.filter(countries, "active");
// => [{"key": "ZA", "name": "South Africa"}];
```

Глава 47.
Loopback — REST-соединитель

Введение

В этой главе мы рассмотрим, что такое REST-соединители и как с ними работать. Мы знаем, что Loopback не обеспечивает комфорта при работе с REST-соединениями.

Примеры

Добавление веб-соединителя

```
//Этот пример получает ответ от iTunes

{
    "rest": {
        "name": "rest",
        "connector": "rest",
        "debug": true,
        "options": {
            "useQuerystring": true,
            "timeout": 10000,
            "headers": {
                "accepts": "application/json",
                "content-type": "application/json"
            }
        },
        "operations": [
            {
                "template": {
                    "method": "GET",
                    "url": "https://itunes.apple.com/search",
                    "query": {
                        "term": "{keyword}",
```

```
                        "country": "{country=IN}",
                        "media": "{itemType=music}",
                        "limit": "{limit=10}",
                        "explicit": "false"
                    }
                },
                "functions": {
                    "search": [
                        "keyword",
                        "country",
                        "itemType",
                        "limit"
                    ]
                }
            },
            {
                "template": {
                    "method": "GET",
                    "url": "https://itunes.apple.com/lookup",
                    "query": {
                        "id": "{id}"
                    }
                },
                "functions": {
                    "findById": [
                        "id"
                    ]
                }
            }
        ]
    }
}
```

Глава 48. metalsmith

Примеры

Создание простого блога

Предполагая, что у вас установлены node и npm, создайте папку (директорию) проекта с действительным файлом `package.json`. Установите необходимые зависимости:

```
npm install --save-dev metalsmith metalsmith-in-place handlebars
```

Создайте файл под названием `build.js` в корне папки вашего проекта, содержащий следующий код:

```
var metalsmith = require('metalsmith');
var handlebars = require('handlebars');
var inPlace = require('metalsmith-in-place');

Metalsmith(__dirname)
    .use(inPlace('handlebars'))
    .build(function(err) {

        if (err) throw err;
        console.log('Build finished!');
    });
```

Создайте папку (директорию) `src` в корне папки вашего проекта. Создайте `index.html` в папке `src`, содержащий следующий код:

```
---
title: My awesome blog
```

```
---
<h1>{{ title }}</h1>
```

Запуск `node build.js` теперь создаст все файлы в папке `src`.

После выполнения этой команды у вас будет `index.html` в вашей папке `build` со следующим содержимым:

```
<h1>My awesome blog</h1>
```

Глава 49.
Интеграция с MongoDB

Синтаксис

- `db.collection.insertOne(document, options(w, wtimeout, j, serializeFunctions, forceServerObjectId, bypassDocumentValidation), callback)`
- `db.collection.insertMany([documents], options(w, wtimeout, j, serializeFunctions, forceServerObjectId, bypassDocumentValidation), callback)`
- `db.collection.find(query)`
- `db.collection.updateOne(filter, update, options(upsert, w, wtimeout, j, bypassDocumentValidation), callback)`
- `db.collection.updateMany(filter, update, options(upsert, w, wtimeout, j), callback)`
- `db.collection.deleteOne(filter, options(upsert, w, wtimeout, j), callback)`
- `db.collection.deleteMany(filter, options(upsert, w, wtimeout, j), callback)`

Параметры

Параметр	Описание
document	JavaScript-объект, представляющий документ
documents	Массив документов
query	Объект, определяющий поисковый запрос

filter	Объект, определяющий поисковый запрос
callback	Функция, вызываемая по завершении операции
options	(опционально) Опциональные настройки (по умолчанию: null)
w	(опционально) Write Concern
wtimeout	(опционально) Тайм-аут для Write Concern (по умолчанию: null)
j	(опционально) Определяет требования журнала (по умолчанию: false)
upsert	(опционально) Обновление или вставка, если запись не найдена (по умолчанию: false)
multi	(опционально) Обновление одного/всех документов (по умолчанию: false)
serializeFunctions	(опционально) Сериализация функций на любом объекте (по умолчанию: false)
forceServerObjectId	(опционально) Принуждение сервера назначать значения _id вместо драйвера (по умолчанию: false)
bypassDocumentValidation	(опционально) Позволяет драйверу обходить валидацию схемы в MongoDB 3.2 или выше (по умолчанию: false)

Примеры

Подключение к MongoDB

Подключитесь к MongoDB, выведите «Connected!» и закройте соединение.

```
const MongoClient = require('mongodb').MongoClient;
var url = 'mongodb://localhost:27017/test';
MongoClient.connect(url, function(err, db) { // метод MongoClient
'connect'
    if (err) throw new Error(err);
    console.log("Connected!");
    db.close(); // Не забудьте закрыть соединение, когда закончите
});
```

Метод Connect() для MongoClient

```
MongoClient.connect(url, options, callback)
```

Аргумент	Тип	Описание
url	строка	Строка, определяющая IP/hostname сервера, порт и базу данных
options	объект	Опциональные настройки (по умолчанию: null)
callback	функция	Функция, вызываемая по завершении попытки подключения

Функция обратного вызова принимает два аргумента:

- `err` : ошибка — если возникает ошибка, аргумент err будет определен
- `db` : объект — экземпляр MongoDB

Добавление документа

Добавьте документ под названием myFirstDocument и установите два свойства, `greetings` и `farewell`:

```
const MongoClient = require('mongodb').MongoClient;
const url = 'mongodb://localhost:27017/test';
MongoClient.connect(url, function (err, db) {
    if (err) throw new Error(err);
        db.collection('myCollection').insertOne({  //  Метод  вставки
'insertOne'
        "myFirstDocument": {
            "greetings": "Hellu",
            "farewell": "Bye"
        }
    }, function (err, result) {
        if (err) throw new Error(err);
            console.log("Inserted  a  document  into  the  myCollection
collection!");
        db.close(); // Не забудьте закрыть соединение, когда закончите
    });
});
```

Метод коллекции insertOne()

```
db.collection(collection).insertOne(document, options, callback)
```

Аргумент	Тип	Описание
collection	строка	Строка, указывающая коллекцию
document	объект	Документ, который будет вставлен в коллекцию
options	объект	(опционально) Опциональные настройки (по умолчанию: null)
callback	функция	Функция, вызываемая по завершении операции вставки

Функция обратного вызова принимает два аргумента:

- `err` : ошибка — если возникает ошибка, аргумент err будет определен
- `result` : объект — Объект, содержащий детали о выполненной операции вставки

Чтение коллекции

Получите все документы в коллекции 'myCollection' и выведите их в консоль.

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

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
    if (err) throw new Error(err);
     var cursor = db.collection('myCollection').find(); // Метод чте-
ния 'find'
    cursor.each(function (err, doc) {
        if (err) throw new Error(err);
        if (doc != null) {
            console.log(doc); // Вывод всех документов
        } else {
             db.close(); // Не забудьте закрыть соединение, когда за-
кончите
        }
    });
});
```

Метод коллекции find()

```
db.collection(collection).find()
```

Аргумент	Тип	Описание
collection	строка	Строка, указывающая коллекцию

Обновление документа

Найдите документ со свойством `{ greetings: 'Hellu' }` и измените его на `{ greetings: 'Whut?' }`.

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

const url = 'mongodb://localhost:27017/test';
MongoClient.connect(url, function (err, db) {
    if (err) throw new Error(err);
       db.collection('myCollection').updateOne({ // Метод обновления
'updateOne'
        greetings: "Hellu" },
        { $set: { greetings: "Whut?" }},
        function (err, result) {
            if (err) throw new Error(err);
             db.close(); // Не забудьте закрыть соединение, когда за-
кончите
        });
});
```

Метод коллекции updateOne()

```
db.collection(collection).updateOne(filter, update, options, callback)
```

Параметр	Тип	Описание
filter	объект	Определяет критерии выбора
update	объект	Определяет модификации, которые будут применены
options	объект	(опционально) Опциональные настройки (по умолчанию: null)
callback	функция	Функция, вызываемая по завершении операции

Функция обратного вызова принимает два аргумента:

- `err` : ошибка — если возникает ошибка, аргумент err будет определен
- `db` : объект — экземпляр MongoDB

Удаление документа

Удаление документа со свойством { greetings: 'Whut?' }

```
const MongoClient = require('mongodb').MongoClient;
const url = 'mongodb://localhost:27017/test';
MongoClient.connect(url, function (err, db) {
    if (err) throw new Error(err);
    db.collection('myCollection').deleteOne(// Метод удаления
'deleteOne'
        { greetings: "Whut?" },
        function (err, result) {
            if (err) throw new Error(err);
            db.close(); // Не забудьте закрыть соединение, когда за-
кончите
        }
    );
});
```

Метод коллекции deleteOne()

```
db.collection(collection).deleteOne(filter, options, callback)
```

Параметр	Тип	Описание
filter	объект	Документ, определяющий критерии выбора.
options	объект	(Необязательно) Опциональные настройки (по умолчанию: null).
callback	функция	Функция, вызываемая по завершении операции.

Функция обратного вызова принимает два аргумента:

- err : ошибка — Если произошла ошибка, аргумент err будет определен.
- db : объект — Экземпляр MongoDB.

Удаление нескольких документов

Удаление всех документов со свойством 'farewell', равным 'okay'.

```
const MongoClient = require('mongodb').MongoClient;
const url = 'mongodb://localhost:27017/test';
MongoClient.connect(url, function (err, db) {
    if (err) throw new Error(err);
```

```
    db.collection('myCollection').deleteMany(// Метод удаления
'deleteMany'
        { farewell: "okay" }, // Удалить все документы со свойством
'farewell: okay'
        function (err, result) {
            if (err) throw new Error(err);
            db.close(); // Не забудьте закрыть соединение, когда за-
кончите
        }
    );
});
```

Метод коллекции deleteMany()

```
db.collection(collection).deleteMany(filter, options, callback)
```

Параметр	Тип	Описание
filter	документ	Документ, определяющий критерии выбора.
options	объект	(Необязательно) Опциональные настройки (по умолчанию: null).
callback	функция	Функция, вызываемая по завершении операции.

Функция обратного вызова (callback) принимает два аргумента:

- err : ошибка — Если произошла ошибка, аргумент err будет определен.
- db : объект — Экземпляр MongoDB.

Простое подключение

```
MongoDB.connect('mongodb://localhost:27017/databaseName',
function(error, database) {
    if(error) return console.log(error);
    const collection = database.collection('collectionName');
    collection.insert({key: 'value'}, function(error, result) {
        console.log(error, result);
    });
});
```

Простое подключение с использованием промисов

```
const MongoDB = require('mongodb');
MongoDB.connect('mongodb://localhost:27017/databaseName')
    .then(function(database) {
        const collection = database.collection('collectionName');
        return collection.insert({key: 'value'});
    })
    .then(function(result) {
        console.log(result);
    });
```

Глава 50.
Интеграция MongoDB для Node.js/Express.js

Введение

MongoDB является одной из самых популярных баз данных NoSQL благодаря поддержке стека MEAN. Взаимодействие с базой данных Mongo через приложение на Express является быстрым и простым, как только вы поймете несколько непривычный синтаксис запросов. Мы будем использовать Mongoose для помощи в этом.

Примечания

Больше информации можно найти здесь: https://Mongoosejs.com/docs/guide.html.

Примеры

Установка MongoDB

```
npm install --save mongodb
npm install --save mongoose //Простой wrapper для облегчения разработки
```

В вашем серверном файле (обычно называется index.js или server.js)

```
const express = require('express');
const mongodb = require('mongodb');
const mongoose = require('mongoose');
const mongoConnectString = 'http://localhost/database name';
mongoose.connect(mongoConnectString, (err) => {
    if (err) {
        console.log('Could not connect to the database'); //Не удалось
подключиться к базе данных
    }
});
```

Создание модели Mongoose

```
const Schema = mongoose.Schema;
const ObjectId = Schema.Types.ObjectId;
const Article = new Schema({
    title: {
        type: String,
        unique: true,
        required: [true, 'Article must have title'] //У статьи должен
быть заголовок
    },
    author: {
        type: ObjectId,
        ref: 'User'
    }
});
module.exports = mongoose.model('Article, Article');
```

Давайте разберем это. MongoDB и Mongoose используют JSON (на самом деле BSON, но это здесь не имеет значения) как формат данных. Вверху мы установили несколько переменных для уменьшения объема кода.

Создадим новую схему и присвоим ее константе. Это простой JSON, и каждый атрибут является еще одним объектом со свойствами, которые помогают обеспечить более согласованную схему. Unique заставляет новые экземпляры, вставляемые в базу данных, быть уникальными. Это отлично подходит для предотвращения создания пользователем нескольких аккаунтов в сервисе.

Required — еще один атрибут, объявленный как массив. Первый элемент — это булево значение, а второй — сообщение об ошибке, если значение, которое вставляется или обновляется, не существует.

ObjectId используется для отношений между моделями. Примерами могут быть 'Users have many Comments'. Другие атрибуты могут быть использованы вместо ObjectId. Строки, такие как имя пользователя, являются одним из примеров.

Наконец, экспортирование модели для использования с вашими API-маршрутами предоставляет доступ к вашей схеме.

Запрос к вашей базе данных Mongo

Рассмотрим простой GET-запрос. Предположим, что модель из приведенного выше примера находится в файле `./db/models/Article.js`.

```
const express = require('express');
const Articles = require('./db/models/Article');
module.exports = function (app) {
    const routes = express.Router();
    routes.get('/articles', (req, res) => {
        Articles.find().limit(5).lean().exec((err, doc) => {
            if (doc.length > 0) {
                res.send({ data: doc });
            } else {
                res.send({ success: false, message: 'No documents
retrieved' }); //Документы не найдены
            }
        });
    });
    app.use('/api', routes);
};
```

Теперь мы можем получить данные из нашей базы данных, отправив HTTP-запрос на эту конечную точку. Однако есть несколько ключевых моментов

1. `Limit` делает именно то, что кажется. Мы получаем только пять документов.

2. `Lean` удаляет некоторые данные из исходного BSON, уменьшая сложность и нагрузку. Это не обязательно, но полезно.

3. При использовании `find` вместо `findOne`, убедитесь, что длина `doc.length` больше 0. Это связано с тем, что `find` всегда возвращает массив, поэтому пустой массив не обработает вашу ошибку, если не проверить длину.

4. Измените сообщение об ошибке в соответствии с вашими потребностями. То же самое касается возвращаемого документа.

5. Код в этом примере написан с учетом того, что вы поместили его в другой файл, а не непосредственно на сервер `Express`. Чтобы вызвать это в сервере, включите следующие строки в свой серверный код:

```
const app = express();
require('./path/to/this/file')(app);
```

Глава 51.
Библиотека Mongoose

Примеры

Подключение к MongoDB с использованием Mongoose

Сначала установите Mongoose с помощью:

```
npm install mongoose
```

Затем добавьте его в файл `server.js` как зависимость:

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
```

Далее создайте схему базы данных и имя коллекции:

```
var schemaName = new Schema({
    request: String,
    time: Number
}, {
    collection: 'collectionName'
});
```

Создайте модель и подключитесь к базе данных:

```
var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');
```

Затем запустите MongoDB и выполните `server.js`, используя команду `node server.js`.

Чтобы проверить, удалось ли подключиться к базе данных, мы можем использовать события `open` и `error` из объекта `Mongoose.connection`:

```
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:')); //
ошибка подключения
```

```
db.once('open', function() {
    // мы подключены!
});
```

Сохранение данных в MongoDB с использованием Mongoose и маршрутов Express.js

Настройка

Сначала установите необходимые пакеты с помощью:

```
npm install express cors mongoose
```

Код

Затем добавьте зависимости в ваш файл `server.js`, создайте схему базы данных и имя коллекции, создайте сервер `Express.js` и подключитесь к MongoDB:

```
var express = require('express');
var cors = require('cors'); // Мы будем использовать CORS для разрешения запросов с других доменов.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var app = express();
var schemaName = new Schema({
    request: String,
    time: Number
}, {
    collection: 'collectionName'
});
var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');
var port = process.env.PORT || 8080;
app.listen(port, function() {
    console.log('Node.js listening on port ' + port); // Node.js слушает на порту
});
```

Теперь добавьте маршруты `Express.js`, которые мы будем использовать для записи данных:

```
app.get('/save/:query', cors(), function(req, res) {
    var query = req.params.query;
    var savedata = new Model({
        'request': query,
```

```
        'time': Math.floor(Date.now() / 1000) // Время сохранения
данных в формате unix timestamp
    }).save(function(err, result) {
        if (err) throw err;

        if(result) {
            res.json(result);
        }
    });
});
```

Здесь переменная query будет параметром <query> из входящего HTTP-запроса, который будет сохранен в MongoDB:

```
var savedata = new Model({
    'request': query,
    //...
```

Если возникнет ошибка при попытке записи в MongoDB, вы получите сообщение об ошибке в консоли. Если все пройдет успешно, вы увидите сохраненные данные в формате JSON на странице.

```
//...
}).save(function(err, result) {
    if (err) throw err;

    if(result) {
        res.json(result);
    }
});
//...
```

Теперь вам нужно запустить MongoDB и выполнить ваш файл server.js с помощью команды node server.js.

Использование

Чтобы использовать этот код для сохранения данных, перейдите по следующему URL в вашем браузере:

```
http://localhost:8080/save/<query>
```

...где <query> — это новый запрос, который вы хотите сохранить.

Пример:

```
http://localhost:8080/save/JavaScript%20is%20Awesome
```

Вывод в формате JSON:

```
{
    __v: 0,
    request: "JavaScript is Awesome",
    time: 1469411348,
    _id: "57957014b93bc8640f2c78c4"
}
```

Поиск данных в MongoDB с использованием Mongoose и маршрутов Express.js

Настройка

Сначала установите необходимые пакеты с помощью:

```
npm install express cors mongoose
```

Код

Затем добавьте зависимости в файл `server.js`, создайте схему базы данных и имя коллекции, создайте сервер Express.js и подключитесь к MongoDB:

```
var express = require('express');
var cors = require('cors'); // Мы будем использовать CORS для разреше-
ния запросов с других доменов.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var app = express();
var schemaName = new Schema({
    request: String,
    time: Number
}, {
    collection: 'collectionName'
});
var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');
var port = process.env.PORT || 8080;
app.listen(port, function() {
    console.log('Node.js listening on port ' + port); // Node.js слу-
шает на порту
});
```

Теперь добавьте маршруты Express.js, которые мы будем использовать для запроса данных:

```
app.get('/find/:query', cors(), function(req, res) {
    var query = req.params.query;
    Model.find({
        'request': query
    }, function(err, result) {
        if (err) throw err;

        if (result) {
            res.json(result);
        } else {
            res.send(JSON.stringify({
                error : 'Error' // Ошибка
            }));
        }
    });
});
```

Предположим, что следующие документы находятся в коллекции, связанной с моделью:

```
{
    "_id" : ObjectId("578abe97522ad414b8eeb55a"),
    "request" : "JavaScript is Awesome",
    "time" : 1468710551
}
{
    "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
    "request" : "JavaScript is Awesome",
    "time" : 1468710555
}
{
    "_id" : ObjectId("578abea0522ad414b8eeb55c"),
    "request" : "JavaScript is Awesome",
    "time" : 1468710560
}
```

Цель заключается в том, чтобы найти и отобразить все документы, содержащие «JavaScript is Awesome» под ключом «`request`».

Для этого запустите MongoDB и выполните `server.js` с помощью команды `node server.js`.

Использование

Чтобы использовать этот код для поиска данных, перейдите по следующему URL в браузере:

```
http://localhost:8080/find/<query>
```

...где <query> — это поисковый запрос.

Пример:

```
http://localhost:8080/find/JavaScript%20is%20Awesome
```

Вывод:

```
[{
    _id: "578abe97522ad414b8eeb55a",
    request: "JavaScript is Awesome",
    time: 1468710551,
    __v: 0
},
{
    _id: "578abe9b522ad414b8eeb55b",
    request: "JavaScript is Awesome",
    time: 1468710555,
    __v: 0
},
{
    _id: "578abea0522ad414b8eeb55c",
    request: "JavaScript is Awesome",
    time: 1468710560,
    __v: 0
}]
```

Поиск данных в MongoDB с использованием Mongoose, маршрутов Express.js и оператора $text

Настройка

Сначала установите необходимый пакет с помощью:

```
npm install express cors mongoose
```

Код

Затем добавьте зависимости в `server.js`, создайте схему базы данных и имя коллекции, создайте сервер `Express.js` и подключитесь к MongoDB:

```
var express = require('express');
var cors = require('cors'); // Мы будем использовать CORS для разреше-
ния запросов с других доменов.
var mongoose = require('mongoose');
```

```
var Schema = mongoose.Schema;
var app = express();
var schemaName = new Schema({
    request: String,
    time: Number
}, {
    collection: 'collectionName'
});
var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');
var port = process.env.PORT || 8080;
app.listen(port, function() {
    console.log('Node.js listening on port ' + port); // Node.js слу-
шает на порту
});
```

Теперь добавьте маршруты Express.js, которые мы будем использовать для запроса данных:

```
app.get('/find/:query', cors(), function(req, res) {
    var query = req.params.query;
    Model.find({
        'request': query
    }, function(err, result) {
        if (err) throw err;
        if (result) {
            res.json(result);
        } else {
            res.send(JSON.stringify({
                error : 'Error' // Ошибка
            }));
        }
    });
});
```

Предположим, что следующие документы находятся в коллекции, связанной с моделью:

```
{
    "_id" : ObjectId("578abe97522ad414b8eeb55a"),
    "request" : "JavaScript is Awesome",
    "time" : 1468710551
}
{
    "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
    "request" : "JavaScript is Awesome",
```

```
    "time" : 1468710555
}
{
    "_id" : ObjectId("578abea0522ad414b8eeb55c"),
    "request" : "JavaScript is Awesome",
    "time" : 1468710560
}
```

Наша цель — найти и отобразить все документы, содержащие только слово «JavaScript» под ключом «request».

Чтобы сделать это, сначала создайте текстовый индекс для «request» в коллекции. Для этого добавьте следующий код в server.js:

```
schemaName.index({ request: 'text' });
```

...и замените:

```
Model.find({
    'request': query
}, function(err, result) {
```

на:

```
Model.find({
    $text: {
        $search: query
    }
}, function(err, result) {
```

Здесь мы используем операторы MongoDB $text и $search, чтобы найти все документы в коллекции collectionName, которые содержат хотя бы одно слово из указанного поискового запроса.

Использование

Чтобы использовать этот код для поиска данных, перейдите по следующему URL в браузере:

```
http://localhost:8080/find/<query>
```

...и <query> — это поисковый запрос.

Mongoose

```
http://localhost:8080/find/JavaScript
```

Вывод:

```
[{
    _id: "578abe97522ad414b8eeb55a",
```

```
    request: "JavaScript is Awesome",
    time: 1468710551,
    __v: 0
},
{
    _id: "578abe9b522ad414b8eeb55b",
    request: "JavaScript is Awesome",
    time: 1468710555,
    __v: 0
},
{
    _id: "578abea0522ad414b8eeb55c",
    request: "JavaScript is Awesome",
    time: 1468710560,
    __v: 0
}]
```

Индексы в моделях

MongoDB поддерживает вторичные индексы. В Mongoose мы определяем эти индексы внутри нашей схемы. Определение индексов на уровне схемы необходимо, когда нужно создать составные индексы.

Подключение Mongoose

```
var strConnection = 'mongodb://localhost:27017/dbName';
var db = mongoose.createConnection(strConnection);
```

Создание базовой схемы

```
var Schema = require('mongoose').Schema;
var usersSchema = new Schema({
    username: {
        type: String,
        required: true,
        unique: true
    },
    email: {
        type: String,
        required: true
    },
    password: {
        type: String,
        required: true
    },
```

```
    created: {
        type: Date,
        default: Date.now
    }
});
var usersModel = db.model('users', usersSchema);
module.exports = usersModel;
```

По умолчанию Mongoose добавляет два новых поля в нашу модель, даже если они не определены в модели. Эти поля:

_id

Mongoose присваивает каждой из ваших схем поле `_id` по умолчанию, если оно не передано в конструктор Schema. Присвоенный тип — это `ObjectId`, чтобы совпадать с поведением MongoDB по умолчанию. Если вы не хотите, чтобы `_id` добавлялся в вашу схему вообще, вы можете отключить его с помощью этой опции:

```
var usersSchema = new Schema({
    username: {
        type: String,
        required: true,
        unique: true
    }, {
        _id: false
    }
});
```

__v или versionKey

`versionKey` — это свойство, установленное на каждом документе при его первом создании с помощью Mongoose. Значение этого ключа содержит внутреннюю ревизию документа. Имя этого свойства документа настраивается. Вы можете легко отключить это поле в конфигурации модели:

```
var usersSchema = new Schema({
    username: {
        type: String,
        required: true,
        unique: true
    }, {
        versionKey: false
    }
});
```

Составные индексы

Мы можем создавать другие индексы, кроме тех, которые создает Mongoose:

```
usersSchema.index({username: 1 });
usersSchema.index({email: 1 });
```

В этом случае наша модель будет иметь два дополнительных индекса: один для поля username и другой для поля email. Но мы можем создавать составные индексы:

```
usersSchema.index({username: 1, email: 1 });
```

Влияние индексов на производительность

По умолчанию Mongoose всегда вызывает ensureIndex для каждого индекса последовательно и генерирует событие index для модели, когда все вызовы ensureIndex завершились успешно или произошла ошибка.

В MongoDB ensureIndex устарел с версии 3.0.0 и теперь является алиасом для createIndex.

Рекомендуется отключить это поведение, установив для опции autoIndex вашей схемы значение false, или глобально для подключения, установив опцию config.autoIndex в false.

```
usersSchema.set('autoIndex', false);
```

Полезные функции Mongoose

Mongoose содержит некоторые встроенные функции, которые расширяют стандартный find().

```
doc.find({'some.value': 5}, function(err, docs) {
    // возвращает массив docs
});
javascript
Копировать код
doc.findOne({'some.value': 5}, function(err, doc) {
    // возвращает документ doc
});
javascript
Копировать код
doc.findById(obj._id, function(err, doc) {
    // возвращает документ doc
});
```

Поиск данных в MongoDB с использованием промисов

Настройка

Сначала установите необходимый пакет с помощью:

```
npm install express cors mongoose
```

Код

Затем добавьте зависимости в `server.js`, создайте схему базы данных и имя коллекции, создайте сервер `Express.js` и подключитесь к MongoDB:

```
var express = require('express');
var cors = require('cors'); // Мы будем использовать CORS для разреше-
ния запросов с других доменов.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var app = express();
var schemaName = new Schema({
    request: String,
    time: Number
}, {
    collection: 'collectionName'
});
var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');
var port = process.env.PORT || 8080;
app.listen(port, function() {
    console.log('Node.js listening on port ' + port); // Node.js слу-
шает на порту
});

app.use(function(err, req, res, next) {
    console.error(err.stack);
    res.status(500).send('Something broke!'); // Что-то пошло не так!
});

app.use(function(req, res, next) {
   res.status(404).send('Sorry cant find that!'); // Извините, не могу
это найти!
});
```

Теперь добавьте маршруты `Express.js`, которые мы будем использовать для запроса данных:

```
app.get('/find/:query', cors(), function(req, res, next) {
    var query = req.params.query;

    Model.find({
        'request': query
    })
    .exec() // не забудьте добавить exec, запросы имеют атрибут .then,
но не являются промисами
```

```
    .then(function(result) {
        if (result) {
            res.json(result);
        } else {
            next(); // переход к обработчику 404
        }
    })
    .catch(next); // переход к обработчику ошибок
});
```

Предположим, что следующие документы находятся в коллекции, связанной с моделью:

```
{
    "_id" : ObjectId("578abe97522ad414b8eeb55a"),
    "request" : "JavaScript is Awesome",
    "time" : 1468710551
}
{
    "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
    "request" : "JavaScript is Awesome",
    "time" : 1468710555
}
{
    "_id" : ObjectId("578abea0522ad414b8eeb55c"),
    "request" : "JavaScript is Awesome",
    "time" : 1468710560
}
```

Наша цель — найти и отобразить все документы, содержащие «`JavaScript is Awesome`» под ключом «`request`».

Для этого запустите MongoDB и выполните `server.js` с помощью `node server.js`:

Использование

Чтобы использовать этот код для поиска данных, перейдите по следующему URL в браузере:

```
http://localhost:8080/find/<query>
```

...где `<query>` — это поисковый запрос.

Пример:

```
http://localhost:8080/find/JavaScript%20is%20Awesome
```

Вывод:

```
[{
    _id: "578abe97522ad414b8eeb55a",
    request: "JavaScript is Awesome",
    time: 1468710551,
    __v: 0
},
{
    _id: "578abe9b522ad414b8eeb55b",
    request: "JavaScript is Awesome",
    time: 1468710555,
    __v: 0
},
{
    _id: "578abea0522ad414b8eeb55c",
    request: "JavaScript is Awesome",
    time: 1468710560,
    __v: 0
}]
```

Глава 52.
Интеграция MSSQL

Введение

Для интеграции любой базы данных с Node.js требуется пакет-драйвер, или, как его еще можно назвать, модуль `npm`, который предоставит вам базовый API для подключения к базе данных и выполнения операций. То же самое справедливо и для базы данных MSSQL. Здесь мы интегрируем MSSQL с Node.js и выполним несколько базовых запросов к таблицам SQL.

Примечания

Мы предполагаем, что у вас будет запущен локальный экземпляр сервера базы данных MSSQL на локальном компьютере. Вы можете обратиться к этому документу (https://learn.microsoft.com/en-us/sql/linux/quickstart-install-connect-ubuntu?view=sql-server-ver16&tabs=ubuntu2004) для получения дополнительной информации.

Также убедитесь, что у вас создан соответствующий пользователь с добавленными привилегиями.

Примеры

Подключение к SQL через модуль npm mssql

Мы начнем с создания простого Node.js-приложения с базовой структурой, затем подключимся к локальному серверу базы данных SQL и выполним несколько запросов к этой базе данных.

Шаг 1. Создайте каталог/папку с именем проекта, который вы собираетесь создать. Инициализируйте приложение Node.js, используя команду `npm init`, которая создаст файл `package.json` в текущем каталоге.

```
mkdir mySqlApp
//папка (директория) создана
cd mySqlApp
//переход в новосозданный каталог
npm init
//ответьте на все вопросы ..
npm install
//Это завершится быстро, так как мы еще не добавили никакие пакеты
в наше приложение.
```

Шаг 2. Теперь мы создадим файл `App.js` в этом каталоге и установим некоторые пакеты, которые нам понадобятся для подключения к базе данных SQL.

```
sudo gedit App.js
//Это создаст файл App.js, вы можете использовать свой любимый тексто-
вый редактор :)
npm install --save mssql
//Это установит пакет mssql в ваше приложение
```

Шаг 3. Теперь мы добавим базовую конфигурационную переменную в наше приложение, которая будет использоваться модулем `mssql` для установления соединения.

```
console.log("Hello world, This is an app to connect to sql server.");

var config = {
    "user": "myusername", //по умолчанию sa
    "password": "yourStrong(!)Password",
    "server": "localhost", // для локальной машины
    "database": "staging", // имя базы данных
    "options": {
```

```
        "encrypt": true
    }
}


sql.connect(config, err => {
    if(err){
        throw err ;
    }

    console.log("Connection Successful !");
    new sql.Request().query('select 1 as number', (err, result) => {
        //обработка ошибки
        console.dir(result)
        // Этот пример использует стратегию обратных вызовов для полу-
чения результатов.
    })

});

sql.on('error', err => {
    // ... обработка ошибок
    console.log("Sql database connection error ", err);
})
```

Шаг 4. Это самый простой шаг, где мы запускаем приложение, и оно подключается к серверу SQL и выводит некоторые простые результаты.

```
node App.js

// Вывод:
// Hello world, This is an app to connect to sql server.
// Connection Successful !
// 1
```

Чтобы использовать промисы или async для выполнения запросов, обратитесь к официальной документации пакета mssql.

- Promises: https://www.npmjs.com/package/mssql#promises.

- Async/Await: https://www.npmjs.com/package/mssql#async-await.

Глава 53. Многопоточность

Введение

Node.js был разработан однопоточным. Таким образом, для всех практических целей приложения, запускаемые с Node, будут работать в одном потоке.

Тем не менее сам Node.js работает многопоточно. Операции ввода-вывода и подобные задачи выполняются в пуле потоков. Кроме того, любой экземпляр приложения Node будет запускаться в отдельном потоке, поэтому для запуска многопоточных приложений нужно запускать несколько экземпляров.

Замечания

Понимание цикла событий (`Event loop`) важно для понимания того, как и зачем использовать несколько потоков.

Примеры

Cluster

Модуль `cluster` позволяет запустить одно и то же приложение несколько раз.

Кластеризация желательна, когда разные экземпляры имеют одинаковый поток выполнения и не зависят друг от друга. В этом сценарии у вас есть один мастер, который может запускать форки (или дочерние процессы). Дочерние процессы работают независимо и имеют свое собственное пространство оперативной памяти и цикл событий.

Настройка кластеров может быть полезна для веб-сайтов / API. Любой поток может обслуживать любого клиента, так как он не зависит от других потоков. База данных (например Redis) может быть задействована для совместного использования Cookies, так как переменные не могут быть разделены между потоками:

```
// runs in each instance
var cluster = require('cluster');
var numCPUs = require('os').cpus().length;
```

```
console.log('I am always called');
if (cluster.isMaster) {
    // runs only once (within the master);
    console.log('I am the master, launching workers!');
    for(var i = 0; i < numCPUs; i++) cluster.fork();
} else {
    // runs in each fork
    console.log('I am a fork!');
    // here one could start, as an example, a web server
}
console.log('I am always called as well');
```

Child Process

Дочерние процессы (Child Processes) используются, когда необходимо запустить процессы независимо друг от друга с различными инициализацией и задачами. Как и форки в кластерах, `child_process` выполняется в своем потоке, но, в отличие от форков, он может общаться со своим родительским процессом.

Связь происходит в обоих направлениях, поэтому родитель и потомок могут прослушивать сообщения и отправлять сообщения.

Родительский процесс (../parent.js):

```
var child_process = require('child_process');
console.log('[Parent]', 'initalize');

var child1 = child_process.fork(__dirname + '/child');
child1.on('message', function(msg) {
    console.log('[Parent]', 'Answer from child: ', msg);
});

// one can send as many messages as one want
child1.send('Hello'); // Hello to you too :)
child1.send('Hello'); // Hello to you too :)
// one can also have multiple children

var child2 = child_process.fork(__dirname + '/child');
```

Дочерний процесс (../child.js):

```
// here would one initialize this child
// this will be executed only once
console.log('[Child]', 'initalize');
// here one listens for new tasks from the parent
```

```
process.on('message', function(messageFromParent) {
    // do some intense work here
    console.log('[Child]', 'Child doing some intense work');
     if(messageFromParent == 'Hello') process.send('Hello to you too
:)');
    else process.send('what?');
});
```

Кроме сообщения можно прослушивать множество событий, таких как 'error', 'connected' или 'disconnect'.

Запуск дочернего процесса связан с определенными затратами. Желательно создавать как можно меньше таких процессов.

Глава 54.
Пул соединений MySQL

Примеры

Использование пула соединений без указания базы данных

Рассмотрим достижение многопользовательского режима (multitenancy) на сервере базы данных с несколькими размещенными на нем базами данных.

Многопользовательский режим является обычным требованием корпоративных приложений в настоящее время, и создание пула соединений для каждой базы данных на сервере базы данных не рекомендуется. Вместо этого можно создать пул соединений с сервером базы данных, а затем переключаться между размещенными на нем базами данных по запросу.

Предположим, что в нашем приложении есть разные базы данных для каждой компании, размещенные на сервере базы данных. Мы будем подключаться к соответствующей базе данных компании, когда пользователь обращается к приложению. Вот пример, как это сделать:

```
var pool = mysql.createPool({
    connectionLimit: 10,
    host: 'example.org',
    user: 'bobby',
    password: 'pass'
});
pool.getConnection(function(err, connection) {
    if (err) {
        return cb(err);
    }
    connection.changeUser({database: "firm1"});
    connection.query("SELECT * from history", function(err, data) {
        connection.release();
        cb(err, data);
    });
});
```

Разберем пример.

При определении конфигурации пула мы не указывали имя базы данных, а указали только сервер базы данных, то есть:

```
{
    connectionLimit: 10,
    host: 'example.org',
    user: 'bobby',
    password: 'pass'
}
```

Таким образом, когда нам нужно использовать конкретную базу данных на сервере, мы просим соединение обратиться к базе данных, используя:

```
connection.changeUser({database: "firm1"});
```

Вы можете ознакомиться с официальной документацией здесь: https://github.com/mysqljs/mysql#switching-users-and-altering-connection-state.

Глава 55.
Интеграция MySQL

Введение

В этой главе вы узнаете, как интегрироваться с Node.js, используя инструмент управления базами данных MySQL. Вы научитесь различным способам подключения и взаимодействия с данными, хранящимися в MySQL, с помощью программы и скрипта на Node.js.

Примеры

Запрос объекта подключения с параметрами

Когда вы хотите использовать в SQL контент, сгенерированный пользователем, это делается с помощью параметров. Например, для поиска пользователя с именем aminadav следует сделать так:

```
var username = 'aminadav';

var querystring = 'SELECT name, email from users where name = ?';

connection.query(querystring, [username], function(err, rows, fields)
{
    if (err) throw err;
    if (rows.length) {
        rows.forEach(function(row) {
            console.log(row.name, 'email address is', row.email);
        });

    } else {
        console.log('There were no results.');
    }
});
```

Использование пула соединений

Выполнение нескольких запросов одновременно

Все запросы в MySQL-соединении выполняются один за другим. Это означает, что если вы хотите выполнить 10 запросов, и каждый запрос занимает 2 секунды, то выполнение всего процесса займет 20 секунд. Решением является создание 10 соединений и выполнение каждого запроса в разном соединении. Это можно сделать автоматически с помощью пула соединений.

```
var pool = mysql.createPool({

    connectionLimit: 10,
    host: 'example.org',
    user: 'bobby',
    password: 'pass',
    database: 'schema'
});
for(var i=0; i<10; i++){
    pool.query('SELECT ` as example', function(err, rows, fields) {

        if (err) throw err;

        console.log(rows[0].example); //Показать 1
    });
}
```

Это выполнит все 10 запросов параллельно.

Когда вы используете пул, вам больше не нужно создавать соединение. Вы можете выполнять запросы напрямую из пула. Модуль MySQL будет искать следующее свободное соединение для выполнения вашего запроса.

Подключение к MySQL

Одним из самых простых способов подключения к MySQL является использование модуля `mysql`. Этот модуль обрабатывает соединение между приложением Node.js и сервером MySQL. Вы можете установить его, как и любой другой модуль:

```
npm install --save mysql
```

Теперь вам нужно создать MySQL-соединение, которое вы сможете использовать для выполнения запросов в дальнейшем.

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

const connection = mysql.createConnection({
```

```
    host: 'localhost',
    user: 'me',
    password: 'secret',
    database: 'database_schema'
});

connection.connect();

// Выполнение некоторых запросов
// Например, SELECT * FROM FOO

connection.end();
```

В следующем примере вы узнаете, как выполнить запрос с использованием объекта соединения.

Запрос объекта соединения без параметров

Вы отправляете запрос в виде строки, и в ответной функции обратного вызова получаете результат. Функция обратного вызова возвращает вам ошибку, массив строк и поля. Каждая строка содержит все столбцы возвращаемой таблицы. Вот пример для пояснения:

```
connection.query('SELECT name, email from users', function(err, rows,
fields) {

    if (err) throw err;
    console.log('There are:', rows.length, 'users');
    console.log('First user name is:', rows[0].name);
});
```

Выполнение нескольких запросов с одним соединением из пула

Могут возникнуть ситуации, когда у вас настроен пул соединений MySQL, но вам нужно выполнить несколько запросов последовательно:

```
SELECT 1;
SELECT 2;
```

Вы могли бы просто выполнить их с помощью `pool.query`, как показано ранее, однако если у вас есть только одно свободное соединение в пуле, вы будете вынуждены ждать, пока оно станет доступным, чтобы выполнить второй запрос.

Однако вы можете сохранить активное соединение из пула и выполнить столько запросов, сколько вам нужно, используя одно соединение, с помощью `pool.getConnection`:

```
pool.getConnection(function (err, conn) {

    if (err) return callback(err);
    conn.query('SELECT 1 AS seq', function (err, rows) {
        if (err) throw err;
        conn.query('SELECT 2 AS seq', function (err, rows) {
            if (err) throw err;
            conn.release();
            callback();
        });
    });
});
```

Примечание: вы должны помнить о необходимости освободить соединение, иначе останется одно соединение MySQL, недоступное для остальной части пула!

Для получения дополнительной информации о пуле подключений MySQL ознакомьтесь с документацией по MySQL: https://www.npmjs.com/package/mysql#pooling-connections.

Возврат запроса при возникновении ошибки

Вы можете прикрепить выполненный запрос к объекту `err`, когда возникает ошибка:

```
var q = mysql.query('SELECT `name` FROM `pokedex` WHERE `id` = ?', [
25 ], function (err, result) {
    if (err) {
        // Таблица 'test.pokedex' не существует
        err.query = q.sql; // SELECT `name` FROM `pokedex` WHERE `id`
= 25
        callback(err);
    }

    else {
```

```
        callback(null, result);
    }
});
```

Экспорт пула соединений

```
// db.js
const mysql = require('mysql');
const pool = mysql.createPool({
    connectionLimit: 10,
    host: 'example.org',
    user: 'bob',
    password: 'secret',
    database: 'my_db'
});
module.export = {
    getConnection: (callback) => {
        return pool.getConnection(callback);
    }
}
// app.js
const db = require('./db');
db.getConnection((err, conn) => {
     conn.query('SELECT something from sometable', (error, results,
fields) => {
        // получить результаты
        conn.release();
    });
});
```

Глава 56.
N-API

Введение

N-API — это новый улучшенный способ создания нативных модулей для Node.js. N-API находится на ранней стадии разработки, поэтому документация по нему может быть неполной или противоречивой.

Примеры

Приветствие в N-API

Этот модуль регистрирует функцию `hello` в модуле `hello`. Функция `hello` выводит в консоль сообщение «`Hello world`» с помощью `printf` и возвращает 1373 из нативной функции в JavaScript-вызов:

```
#include <node_api.h>
#include <stdio.h>

napi_value say_hello(napi_env env, napi_callback_info info)
{
    napi_value retval;
    printf("Hello world\n");
    napi_create_number(env, 1373, &retval);
    return retval;
}

void init(napi_env env, napi_value exports, napi_value module, void*
priv)
{
    napi_status status;
    napi_property_descriptor desc = {
        /*
       * Строка, описывающая ключ для свойства, закодированная в UTF8.
```

```
        */
        .utf8name = "hello",
        /*
         * Установите это значение, чтобы свойство value объекта дес-
криптора свойства
        * стало JavaScript-функцией, представленной методом.
       * Если этот параметр передан, установите значения value, getter
и setter в NULL
        * (так как эти члены не будут использоваться).
        */
        .method = say_hello,
        /*
       * Функция, которая вызывается при получении доступа к свойству.
      * Если этот параметр передан, установите значения value и method
в NULL
        * (так как эти члены не будут использоваться).
         * Данная функция вызывается неявно средой выполнения при до-
ступе к свойству
          * из JavaScript-кода (или если выполнено получение значения
свойства с помощью
        * вызова N-API).
        */
        .getter = NULL,
        /*
       * Функция, которая вызывается при установке значения свойства.
      * Если этот параметр передан, установите значения value и method
в NULL
        * (так как эти члены не будут использоваться).
        * Данная функция вызывается неявно средой выполнения при уста-
новке значения свойства
          * из JavaScript-кода (или если выполнена установка значения
свойства с помощью вызова
        * N-API).
        */
        .setter = NULL,
        /*
       * Значение, которое возвращается при получении доступа к свой-
ству, если
        * это свойство является свойством данных.
           * Если этот параметр передан, установите значения getter,
setter, method и data в NULL
        * (так как эти члены не будут использоваться).
        */
        .value = NULL,
```

```
        /*
       * Атрибуты, связанные с конкретным свойством. См. napi_property_
attributes.
        */
        .attributes = napi_default,
        /*
           * Данные обратного вызова, передаваемые в method, getter
и setter, если эта функция
        * вызывается.
        */
        .data = NULL
    };
    /*
     * Этот метод позволяет эффективно определить несколько свойств
для заданного объекта.
    */
    status = napi_define_properties(env, exports, 1, &desc);
    if (status != napi_ok)
        return;
}

NAPI_MODULE(hello, init)
```

Глава 57.
Локализация Node.js

Введение

Рассмотрим, как поддерживать локализацию в Node.js с использованием `express`.

Примеры

Использование модуля i18n для поддержания локализации в приложении на Node.js

Рассмотрим легковесный простой модуль для перевода с динамическим хранением в формате JSON. Поддерживает обычные приложения на чистом Node.js и должен работать с любым фреймворком (такими как `express`, `restify` и, вероятно, другими), который предоставляет метод `app.use()`, передавая в него объекты `res` и `req`. Использует общий синтаксис __('...') в приложениях и шаблонах. Сохраняет языковые файлы в формате JSON, совместимом с форматом webtranslateit JSON. Добавляет новые строки «на лету», когда они впервые используются в вашем приложении. Не требуется дополнительный парсинг:

```
// обычные требования
var express = require('express'),
    i18n = require('i18n'),
    app = module.exports = express();

i18n.configure({
    // настройка некоторых локалей - другие локали по умолчанию переключаются на en без предупреждения
    locales: ['en', 'ru', 'de'],
    // задает имя cookie для анализа настроек локали
    cookie: 'yourcookiename',
    // где хранить json-файлы - по умолчанию './locales'
```

```
    directory: __dirname + '/locales'
});

app.configure(function () {
    // вам нужно будет использовать cookieParser, чтобы сделать cookies
доступными через req.cookies
    app.use(express.cookieParser());
    // инициализация i18n, анализирует req на наличие заголовков язы-
ка, cookies и т.д.
    app.use(i18n.init);
});

// обслуживание главной страницы
app.get('/', function (req, res) {
    res.send(res.__('Hello World'));
});

// запуск сервера
if (!module.parent) {
    app.listen(3000);
}
```

Глава 58.
Сервер Node без фреймворка

Примечания

Node имеет множество фреймворков, которые помогают быстро запустить сервер, среди них:

Express: самый используемый фреймворк.

Total: УНИВЕРСАЛЬНЫЙ фреймворк, который имеет все необходимое и не зависит от других фреймворков или модулей.

Но, как известно, универсального решения не существует, поэтому разработчику может понадобиться создать свой собственный сервер без каких-либо зависимостей.

Если приложение доступно через внешний сервер, CORS может стать проблемой. Рассмотрим код, который поможет избежать этого.

Примеры

Сервер на Node без фреймворка

```
var http = require('http');
var fs = require('fs');
var path = require('path');

http.createServer(function (request, response) {
    console.log('request ', request.url);
    var filePath = '.' + request.url;
    if (filePath == './')
        filePath = './index.html';

    var extname = String(path.extname(filePath)).toLowerCase();
    var contentType = 'text/html';
    var mimeTypes = {
        '.html': 'text/html',
```

```
        '.js': 'text/javascript',
        '.css': 'text/css',
        '.json': 'application/json',
        '.png': 'image/png',
        '.jpg': 'image/jpg',
        '.gif': 'image/gif',
        '.wav': 'audio/wav',
        '.mp4': 'video/mp4',
        '.woff': 'application/font-woff',
        '.ttf': 'applilcation/font-ttf',
        '.eot': 'application/vnd.ms-fontobject',
        '.otf': 'application/font-otf',
        '.svg': 'application/image/svg+xml'
    };

    contentType = mimeTypes[extname] || 'application/octect-stream';
    fs.readFile(filePath, function(error, content) {
        if (error) {
            if(error.code == 'ENOENT'){
                fs.readFile('./404.html', function(error, content) {
                 response.writeHead(200, { 'Content-Type': contentType
});
                    response.end(content, 'utf-8');
                });
            }
            else {
                response.writeHead(500);
                  response.end('Sorry, check with the site admin for
error: ' + error.code + ' ..\n');
                response.end();
            }
        }
        else {
            response.writeHead(200, { 'Content-Type': contentType });
            response.end(content, 'utf-8');
        }
    });
}).listen(8125);
console.log('Server running at http://127.0.0.1:8125/');
```

Решение проблем с CORS

```
// Сайт, который вы хотите разрешить для подключения
response.setHeader('Access-Control-Allow-Origin', '*');
```

```
// Методы запросов, которые вы хотите разрешить
response.setHeader('Access-Control-Allow-Methods', 'GET, POST,
OPTIONS, PUT, PATCH, DELETE');

// Заголовки запросов, которые вы хотите разрешить
response.setHeader('Access-Control-Allow-Headers', 'X-Requested-
With,content-type');

// Установите в true, если вам нужно, чтобы сайт включал куки в запро-
сы, отправляемые на API
// (например, в случае использования сессий)
response.setHeader('Access-Control-Allow-Credentials', true);
```

Глава 59.
Node.js (express.js) с Angular.js

Введение

В этой главе показано, как создать базовое приложение на Express и затем использовать AngularJS.

Примеры

Создание проекта

Мы готовы начать, поэтому выполняем в консоли:

```
mkdir our_project
cd our_project
```

Теперь мы находимся в папке, где будет размещен наш код. Чтобы создать основной файл нашего проекта, можно запустить команду.

Как создать каркас проекта на express?

Это просто:

```
npm install -g express express-generator
```

В дистрибутивах Linux и Mac для установки этой команды необходимо использовать sudo, так как они устанавливаются в каталог nodejs, доступ к которому имеет только пользователь `root`. Если все прошло успешно, мы наконец можем создать каркас Express-приложения, просто запустив

```
express
```

Эта команда создаст пример приложения `express` внутри нашей папки. Структура будет следующей:

```
bin/
public/
routes/
views/
app.js
package.json
```

Теперь, если мы запустим npm start и перейдем на http://localhost:3000, мы увидим, что приложение Express работает. Это хорошо, но как мы можем объединить это с AngularJS?

Как работает Express?

Express — это фреймворк, построенный поверх Node.js. Вы можете ознакомиться с официальной документацией на сайте Express (https://expressjs.com/).

Для наших целей нужно знать, что Express отвечает за отображение страницы, когда мы вводим, например, http://localhost:3000/home. В недавно созданном приложении можно проверить:

Файл: routes/index.js

```
var express = require('express');
var router = express.Router();
/* GET home page. */
router.get('/', function(req, res, next) {
    res.render('index', { title: 'Express' });
});
module.exports = router;
```

Этот код указывает, что при переходе пользователя на http://localhost:3000 должно быть отображено представление index и передан JSON с свойством

title и значением Express. Но если мы проверим каталог views и откроем файл index.jade, то увидим следующее:

```
extends layout

block content
    h1= title
    p Welcome to #{title}
```

Это еще одна мощная функция Express — шаблонизаторы (template engines). Они позволяют отображать содержимое на странице, передавая в него переменные, или наследовать другой шаблон, чтобы ваши страницы были более компактными и понятными для других. Расширение файла .jade. По некоторым сведениям, Jade сменил название на Pug. Это по сути тот же шаблонизатор, но с некоторыми обновлениями и изменениями ядра.

Установка Pug и обновление шаблонизатора Express

Итак, чтобы начать использовать Pug в качестве шаблонизатора в нашем проекте, нужно запустить:

```
npm install --save pug
```

Это установит Pug как зависимость нашего проекта и сохранит его в package.json. Чтобы использовать его, нужно изменить файл app.js:

```
var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
```

Замените строку view engine на pug, и на этом все. Мы можем снова запустить наш проект с помощью npm start и убедиться, что все работает правильно.

Как AngularJS вписывается во все это?

AngularJS — это JavaScript-фреймворк MVW (Model-View-Whatever), в основном используемый для создания SPA (Single Page Application). Установка довольно проста: вы можете зайти на сайт AngularJS (https://angularjs.org/) и загрузить последнюю версию, на данный момент это v1.6.4.

После того как мы загрузили AngularJS, нужно скопировать файл в нашу папку public/javascripts внутри проекта. Поясним: это папка,

которая обслуживает статические ресурсы нашего сайта, такие как изображения, CSS, JavaScript-файлы и так далее. Конечно, это можно настроить через файл `app.js`, но мы будем придерживаться простого подхода. Теперь создаем файл с именем `ng-app.js`, в котором будет находиться наше приложение, внутри папки `javascripts` в `public`, где также находится AngularJS. Чтобы запустить AngularJS, нужно изменить содержимое файла `views/layout.pug` следующим образом:

```
doctype html
html(ng-app='first-app')
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body(ng-controller='indexController')
    block content
    script(type='text-javascript', src='javascripts/angular.min.js')
    script(type='text-javascript', src='javascripts/ng-app.js')
```

Что мы делаем здесь? Мы включаем ядро AngularJS и наш недавно созданный файл `ng-app.js`, чтобы при отображении шаблона AngularJS был активирован. Обратите внимание на использование директивы `ng-app` — она указывает AngularJS, что это имя нашего приложения, и он должен привязываться к нему.

Итак, содержимое нашего файла `ng-app.js` будет следующим:

```
angular.module('first-app', [])
    .controller('indexController', ['$scope', indexController]);

function indexController($scope) {
    $scope.name = 'sigfried';
}
```

Здесь мы используем базовую функцию AngularJS — двустороннее связывание данных. Это позволяет нам обновлять содержимое представления и контроллера мгновенно. Это очень простое объяснение, но вы можете поискать информацию в Google или на StackOverflow, чтобы узнать, как это работает на самом деле.

Итак, у нас есть базовые блоки нашего приложения на AngularJS, но есть еще кое-что, что нужно сделать: обновить нашу страницу `index.pug`, чтобы увидеть изменения в нашем Angular-приложении. Давайте сделаем это:

```
extends layout

block content
  div(ng-controller='indexController')
    h1= title
    p Welcome {{name}}
    input(type='text' ng-model='name')
```

Здесь мы просто связываем ввод с нашим определенным свойством name в области видимости AngularJS внутри нашего контроллера:

```
$scope.name = 'sigfried';
```

Цель этого заключается в том, что всякий раз, когда мы изменяем текст в поле ввода, абзац выше будет обновляться с содержимым внутри {{name}}. Это называется интерполяцией и является еще одной функцией AngularJS для отображения содержимого в шаблоне.

Итак, все настроено, теперь мы можем запустить `npm start`, перейти на `http://localhost:3000` и увидеть, как наше Express-приложение обслуживает страницу, а AngularJS управляет фронтендом приложения.

Глава 60.
Node.js и MongoDB

Примечания

Это основные операции CRUD для работы с MongoDB в Node.js.
Вопрос: Можно ли сделать то, что сделано здесь, другими способами?
Ответ: Да, существует множество способов сделать это.
Вопрос: Обязательно ли использовать Mongoose?
Ответ: Нет. Существуют другие пакеты, которые могут помочь.
Вопрос: Где я могу найти полную документацию по Mongoose?
Ответ: Перейдите по ссылке https://Mongoosejs.com/docs/api/Mongoose.html.

Примеры

Подключение к базе данных

Для подключения к базе данных Mongo из приложения на Node.js требуется Mongoose.

Установка Mongoose

Перейдите в корень вашего приложения и установите Mongoose с помощью команды:

```
npm install mongoose
```

Далее подключаемся к базе данных.

```
var mongoose = require('mongoose');
// Подключаемся к тестовой базе данных, запущенной на стандартном порту mongod на localhost
mongoose.connect('mongodb://localhost/test');

// Подключение с использованием пользовательских учетных данных
mongoose.connect('mongodb://USER:PASSWORD@HOST:PORT/DATABASE');

// Использование Pool Size для определения количества открытых соединений
// Также можно использовать функцию обратного вызова для обработки ошибок
mongoose.connect(
    'mongodb://localhost:27017/consumers',
    {server: { poolSize: 50 }},
    function(err) {
        if(err) {
            console.log('error in this')
            console.log(err);
            // Выполнить действия для обработки ошибки
        } else {
            console.log('Connected to the database');
        }
    }
);
```

Создание новой коллекции

В Mongoose все основывается на схеме (Schema). Давайте создадим схему:

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var AutoSchema = new Schema({
    name : String,
    countOf: Number,
});
// определение структуры документа
// по умолчанию коллекция, созданная в базе данных, будет названа в соответствии с первым параметром (или его множественным числом)
module.exports = mongoose.model('Auto', AutoSchema);
```

```
// мы можем переопределить это и задать имя коллекции, указав его
в третьем параметре.
module.exports = mongoose.model('Auto', AutoSchema, 'collectionName');

// Также мы можем определить методы в моделях.
AutoSchema.methods.speak = function () {
    var greeting = this.name
        ? "Hello this is " + this.name + " and I have counts of " +
this.countOf
        : "I don't have a name";
    console.log(greeting);
}
mongoose.model('Auto', AutoSchema, 'collectionName');
```

Помните, что методы должны быть добавлены к схеме до компиляции ее с помощью Mongoose `.model()`, как показано выше.

Вставка документов

Для вставки нового документа в коллекцию мы создаем объект схемы:

```
var Auto = require('models/auto');
var autoObj = new Auto({
    name: "NewName",
    countOf: 10
});
```

Сохраняем его следующим образом:

```
autoObj.save(function(err, insertedAuto) {
    if (err) return console.error(err);
    insertedAuto.speak();
    // output: Hello this is NewName and I have counts of 10
});
```

Это вставит новый документ в коллекцию.

Чтение

Чтение данных из коллекции происходит очень просто. Получение всех данных из коллекции:

```
var Auto = require('models/auto');
Auto.find({}, function (err, autos) {
    if (err) return console.error(err);
    // вернет JSON-массив всех документов в коллекции
```

```
    console.log(autos);
});
```

Чтение данных с условием:

```
Auto.find({countOf: {$gte: 5}}, function (err, autos) {
    if (err) return console.error(err);
    // вернет JSON-массив всех документов в коллекции, где count боль-
ше или равен 5
    console.log(autos);
});
```

Также вы можете указать второй параметр как объект, определяющий, какие поля вам нужны:

```
Auto.find({}, {name:1}, function (err, autos) {
    if (err) return console.error(err);
    // вернет JSON-массив с полем name всех документов в коллекции
    console.log(autos);
});
```

Нахождение одного документа в коллекции:

```
Auto.findOne({name: "newName"}, function (err, auto) {
    if (err) return console.error(err);
    // вернет первый объект документа, у которого name равен "newName"
    console.log(auto);
});
```

Нахождение одного документа в коллекции по id:

```
Auto.findById(123, function (err, auto) {
    if (err) return console.error(err);
    // вернет первый JSON-объект документа, у которого id равен 123
    console.log(auto);
});
```

Обновление

Для обновления коллекций и документов можно использовать любой из этих методов

- `update()`
- `updateOne()`
- `updateMany()`
- `replaceOne()`

Update()

Метод update() модифицирует один или несколько документов (параметры обновления).

```
db.lights.update(
    { room: "Bedroom" },
    { status: "On" }
)
```

Эта операция ищет в коллекции lights документ, где room равно "Bedroom" (1-й параметр). Затем она обновляет свойство status совпадающих документов до "On" (2-й параметр) и возвращает объект WriteResult, который выглядит так:

```
{ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }
```

UpdateOne

Метод UpdateOne() модифицирует ОДИН документ (параметры обновления).

```
db.countries.update(
    { country: "Sweden" },
    { capital: "Stockholm" }
)
```

Эта операция ищет в коллекции countries документ, где country равно "Sweden" (1-й параметр). Затем она обновляет свойство capital совпадающего документа до "Stockholm" (2-й параметр) и возвращает объект WriteResult, который выглядит так:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

UpdateMany

Метод UpdateMany() модифицирует несколько документов (параметры обновления).

```
db.food.updateMany(
    { sold: { $lt: 10 } },
    { $set: { sold: 55 } }
)
```

Эта операция обновляет все документы (в коллекции food), где sold меньше 10 (1-й параметр), устанавливая sold на 55. Затем она возвращает объект WriteResult, который выглядит так:

```
{ "acknowledged" : true, "matchedCount" : a, "modifiedCount" : b }
```

где:

- a = количество совпадающих документов
- b = количество модифицированных документов

ReplaceOne

Заменяет первый совпадающий документ (документ для замены).
Эта коллекция `countries` содержит три документа:

```
{ "_id" : 1, "country" : "Sweden" }
{ "_id" : 2, "country" : "Norway" }
{ "_id" : 3, "country" : "Spain" }
```

Следующая операция заменяет документ `{ country: "Spain" }` на документ `{ country: "Finland" }`:

```
db.countries.replaceOne(
    { country: "Spain" },
    { country: "Finland" }
)
```

...и возвращает:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

Теперь коллекция `countries` содержит:

```
{ "_id" : 1, "country" : "Sweden" }
{ "_id" : 2, "country" : "Norway" }
{ "_id" : 3, "country" : "Finland" }
```

Удаление

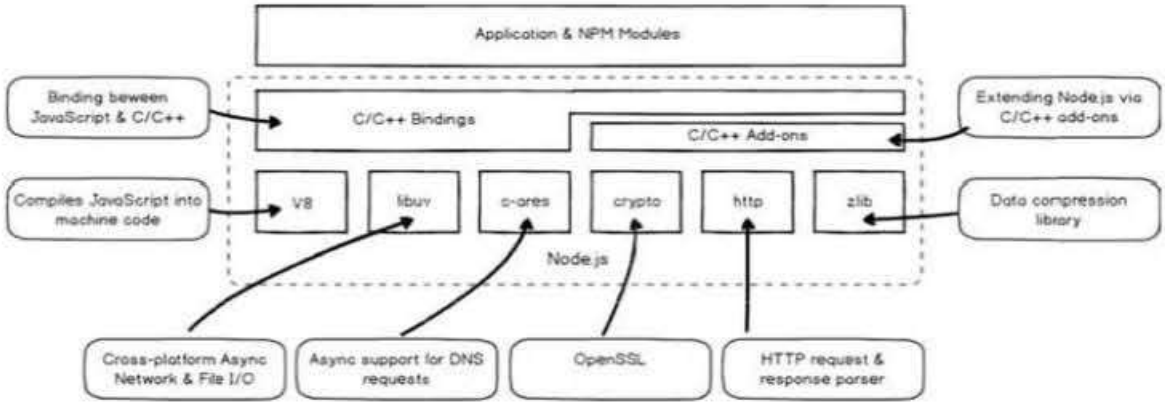
Удаление документов из коллекции в Mongoose выполняется следующим образом:

```
Auto.remove({_id:123}, function(err, result){
    if (err) return console.error(err);
    console.log(result); // это выведет результат удаления по умолча-
нию в MongoDB
});
```

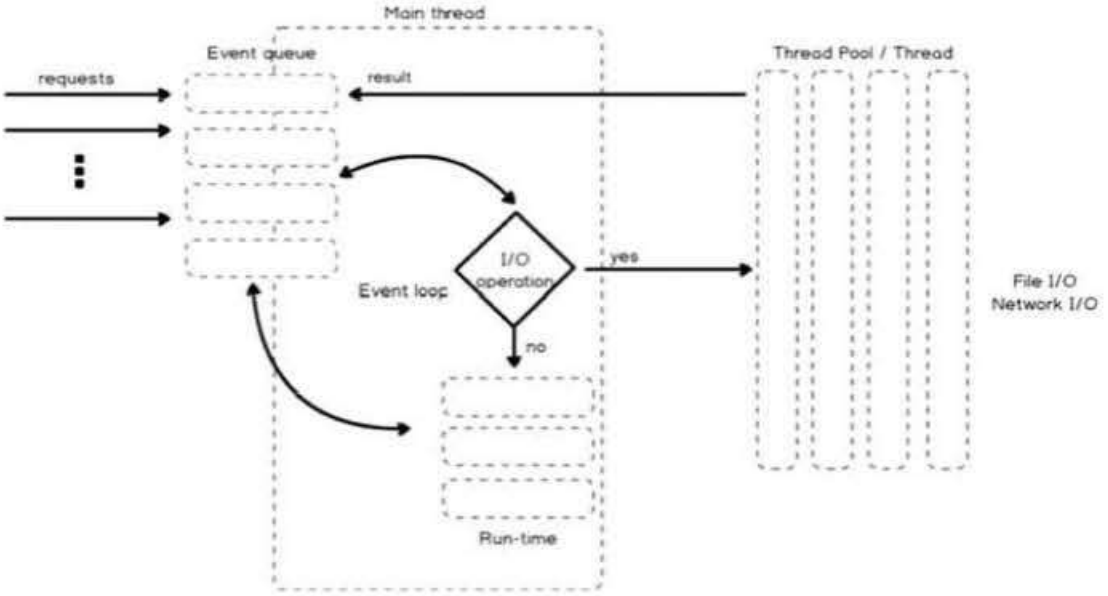
Глава 61.
Архитектура и внутренняя работа Node.js

Примеры

Node.js — «под капотом»



Node.js — «в движении»



Глава 62.
Код Node.js для STDIN и STDOUT без использования библиотек

Введение

Это простая программа на Node.js, которая принимает ввод от пользователя и выводит его в консоль. Объект `process` является глобальным и предоставляет информацию о текущем процессе Node.js, а также контроль над ним. Будучи глобальным, он всегда доступен в приложениях Node.js без использования `require()`.

Примеры

Программа

Свойство `process.stdin` возвращает поток, эквивалентный или ассоциированный с `stdin`. Свойство `process.stdout` возвращает поток, эквивалентный или ассоциированный с `stdout`:

```
process.stdin.resume();
console.log('Enter the data to be displayed');
process.stdin.on('data', function(data) {
    process.stdout.write(data);
});
```

Глава 63.
Основы проектирования Node.js

Примеры

Малое ядро, малые модули

Создавайте небольшие однопрофильные модули не только с точки зрения размера кода, но и с точки зрения охвата, чтобы они выполняли одну задачу:

```
a - "Small is beautiful"
b - "Make each program do one thing well."
```

Шаблон Reactor

Шаблон Reactor является «сердцем» асинхронной природы Node.js. Он позволяет системе работать как однопоточный процесс с серией генераторов событий и обработчиков событий, с помощью постоянно работающего цикла событий.

Неблокирующий движок ввода-вывода Node.js — libuv

Шаблон Observer (`EventEmitter`) поддерживает список зависимых/наблюдателей и уведомляет их:

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

var ringBell = function ringBell() {
    console.log('tring tring tring');
}

eventEmitter.on('doorOpen', ringBell);

eventEmitter.emit('doorOpen');
```

Глава 64.
Управление ошибками в Node.js

Введение

В этой главе мы изучим, как создавать объекты ошибок, а также как вызывать и обрабатывать ошибки в Node.js.

Примеры

Создание объекта ошибки (Error)

new Error(message)

Создает новый объект ошибки, где значение `message` устанавливается в свойство `message` созданного объекта. Обычно аргумент `message` передается конструктору `Error` как строка. Однако если аргумент `message` является объектом, а не строкой, конструктор `Error` вызывает метод `.toString()` у переданного объекта и устанавливает это значение в свойство `message` созданного объекта ошибки:

```
var err = new Error("The error message");
console.log(err.message); // Выводит: The error message
console.log(err);
// Вывод:
// Error: The error message
// at ...
```

Каждый объект ошибки имеет трассировку стека. Трассировка стека содержит сообщение об ошибке и показывает, где именно она произошла (выше приведен вывод трассировки стека). Как только объект ошибки создан, система захватывает трассировку стека ошибки на текущей строке. Чтобы получить трассировку стека, используйте свойство `stack` любого созданного объекта ошибки. Следующие две строки идентичны:

```
console.log(err);
console.log(err.stack);
```

Выброс ошибки

Выброс ошибки означает исключение. Если исключение не обработано, сервер Node.js завершится аварийно. Следующая строка выбрасывает ошибку:

```
throw new Error("Some error occurred");
```

или

```
var err = new Error("Some error occurred");
throw err;
```

или

```
throw "Some error occurred";
```

Последний пример (выброс строк) не является хорошей практикой и не рекомендуется (всегда выбрасывайте ошибки, которые являются экземплярами объекта `Error`).

Учтите, что если вы выбросите (throw) ошибку в вашем коде, система завершит работу на этой строке (если нет обработчиков исключений), и никакой код после этой строки не будет выполнен.

```
var a = 5;
var err = new Error("Some error message");
throw err; // это выведет стек ошибки, и сервер Node.js остановится
a++; // эта строка никогда не будет выполнена
console.log(a); // и эта также
```

Но в этом примере:

```
var a = 5;
var err = new Error("Some error message");
console.log(err); // это выведет стек ошибки
a++;
console.log(a); // эта строка будет выполнена и выведет 6
```

Блок try...catch

Блок `try...catch` используется для обработки исключений. Помните, исключение означает выброшенную ошибку, а не ошибку.

```
try {
    var a = 1;
    b++; // это вызовет ошибку, так как b не определена
    console.log(b); // эта строка не будет выполнена
} catch (error) {
```

```
    console.log(error); // здесь мы обрабатываем ошибку, вызванную
в блоке try
}
```

В блоке `try` `b++` вызывает ошибку, и эта ошибка передается в блок `catch`, где она может быть обработана или даже выброшена снова с некоторыми изменениями. Рассмотрим следующий пример:

```
try {
    var a = 1;
    b++;
    console.log(b);
} catch (error) {
    error.message = "b variable is undefined, so the undefined can't
be incremented";
    throw error;
}
```

В приведенном выше примере мы изменили свойство `message` объекта ошибки, а затем выбросили измененную ошибку.

Вы можете выбросить любую ошибку в вашем блоке `try` и обработать ее в блоке `catch`:

```
try {
    var a = 1;
    throw new Error("Some error message");
    console.log(a); // эта строка не будет выполнена
} catch (error) {
    console.log(error); // будет выброшена вышеуказанная ошибка
}
```

Глава 65.
Производительность Node.js

Примеры

Цикл событий

Пример блокирующей операции

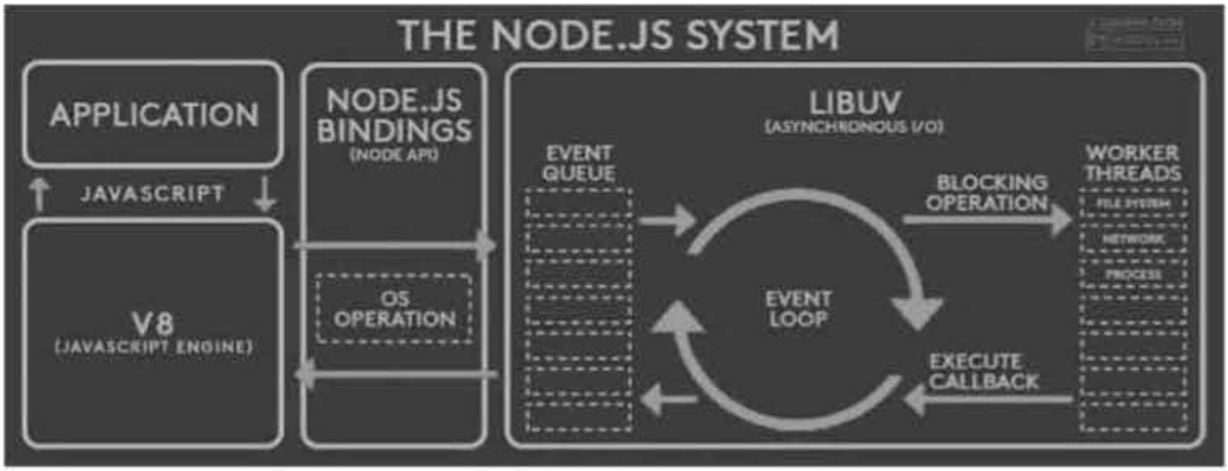
```
let loop = (i, max) => {
    while (i < max) i++
    return i
}

// Эта операция заблокирует Node.js,
// потому что она ориентирована на процессор (CPU-bound).
// Вам следует быть осторожными с подобным кодом.
loop(0, 1e+12)
```

Пример неблокирующей операции ввода-вывода (IO-bound)

```
let i = 0
const step = max => {
    while (i < max) i++
    console.log('i = %d', i)
}

const tick = max => process.nextTick(step, max)

// это отложит выполнение цикла while в step до циклов цикла событий
// любая другая операция, ориентированная на ввод-вывод (например,
чтение файловой системы), может выполняться параллельно
tick(1e+6)
tick(1e+7)
```

```
console.log('this will output before all of tick operations. i = %d',
i)
console.log('because tick operations will be postponed')
tick(1e+8)
```

Проще говоря, цикл событий — это однопоточный механизм очереди, который выполнет ваш код, ориентированный на процессор, до завершения его выполнения, и код, ориентированный на ввод-вывод, в неблокирующем режиме.



Однако «под капотом» Node.js использует многопоточность для некоторых своих операций через библиотеку libuv. Документацию по этой библиотеке можно найти по ссылке https://libuv.org/.

Рекомендации по производительности

- Неблокирующие операции не блокируют очередь и не влияют на производительность цикла.
- Однако операции, ориентированные на процессор, будут блокировать очередь, поэтому вам следует избегать таких операций в вашем коде на Node.js.

Node.js не блокирует операции ввода-вывода, так как передает их выполнение ядру операционной системы, и когда операция ввода-вывода предоставляет данные (как событие), она уведомляет ваш код с помощью переданных вами обратных вызовов.

Увеличение maxSockets

Основы

```
require('http').globalAgent.maxSockets = 25
// Вы можете изменить 25 на Infinity или другое значение, экспериментируя
```

По умолчанию Node.js использует `maxSockets` = Infinity (начиная с v0.12.0). До версии Node v0.12.0 значение по умолчанию было `maxSockets` = 5.

Так что после более чем пяти запросов они будут поставлены в очередь. Если вы хотите увеличить параллелизм, увеличьте это число.

Установка собственного агента

API http использует `Global Agent`. Вы можете предоставить свой собственный агент. Например:

```
const http = require('http')
const myGloriousAgent = new http.Agent({ keepAlive: true })
myGloriousAgent.maxSockets = Infinity
http.request({ ..., agent: myGloriousAgent }, ...)
```

Полное отключение пула сокетов

```
const http = require('http')
const options = {.....}
options.agent = false
const request = http.request(options)
```

Подводные камни

- Вам нужно проделать то же самое для API https, если вы хотите добиться таких же эффектов.
- Учтите, что, например, AWS будет использовать значение 50 вместо Infinity.

Включение gzip

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

```
const zlib = require('zlib')
http.createServer((request, response) => {
    const stream = fs.createReadStream('index.html')
    const acceptsEncoding = request.headers['accept-encoding']
    let encoder = {
        hasEncoder : false,
        contentEncoding: {},
        createEncoder : () => throw 'There is no encoder'
    }

    if (!acceptsEncoding) {
        acceptsEncoding = ''
    }

    if (acceptsEncoding.match(/\bdeflate\b/)) {
        encoder = {
            hasEncoder : true,
            contentEncoding: { 'content-encoding': 'deflate' },
            createEncoder : zlib.createDeflate
        }
    } else if (acceptsEncoding.match(/\bgzip\b/)) {
        encoder = {
            hasEncoder : true,
            contentEncoding: { 'content-encoding': 'gzip' },
            createEncoder : zlib.createGzip
        }
    }

    response.writeHead(200, encoder.contentEncoding)
    if (encoder.hasEncoder) {
        stream = stream.pipe(encoder.createEncoder())
    }
    stream.pipe(response)
}).listen(1337)
```

Глава 66.
Новые функции и улучшения Node.js v6

Введение

С выходом Node.js v6 в качестве новой версии LTS произошел ряд улучшений благодаря новым стандартам ES6. В этой главе мы рассмотрим некоторые из новых функций, введенных в Node.js v6, и примеры их реализации.

Примеры

Параметры по умолчанию для функций

```
function addTwo(a, b = 2) {
    return a + b;
}
addTwo(3) // Возвращает результат 5
```

С добавлением параметров по умолчанию для функций вы теперь можете сделать аргументы опциональными и задать им значения по умолчанию на ваше усмотрение.

Остаточные параметры

```
function argumentLength(...args) {
    return args.length;
}
argumentLength(5) // возвращает 1
argumentLength(5, 3) // возвращает 2
argumentLength(5, 3, 6) // возвращает 3
```

Если перед аргументом функции указать ..., то все переданные функции аргументы будут восприниматься как массив. В этом примере мы можем переда-

вать несколько аргументов и получать длину массива, созданного из этих аргументов.

Оператор расширения (Spread Operator)

```
function myFunction(x, y, z) { }

var args = [0, 1, 2];
myFunction(...args);
```

Синтаксис расширения позволяет расширить выражение в местах, где ожидаются несколько аргументов (для вызовов функций), несколько элементов (для массивов) или несколько переменных. Подобно остаточным параметрам, просто поставьте ... перед массивом.

Стрелочные функции (Arrow Functions)

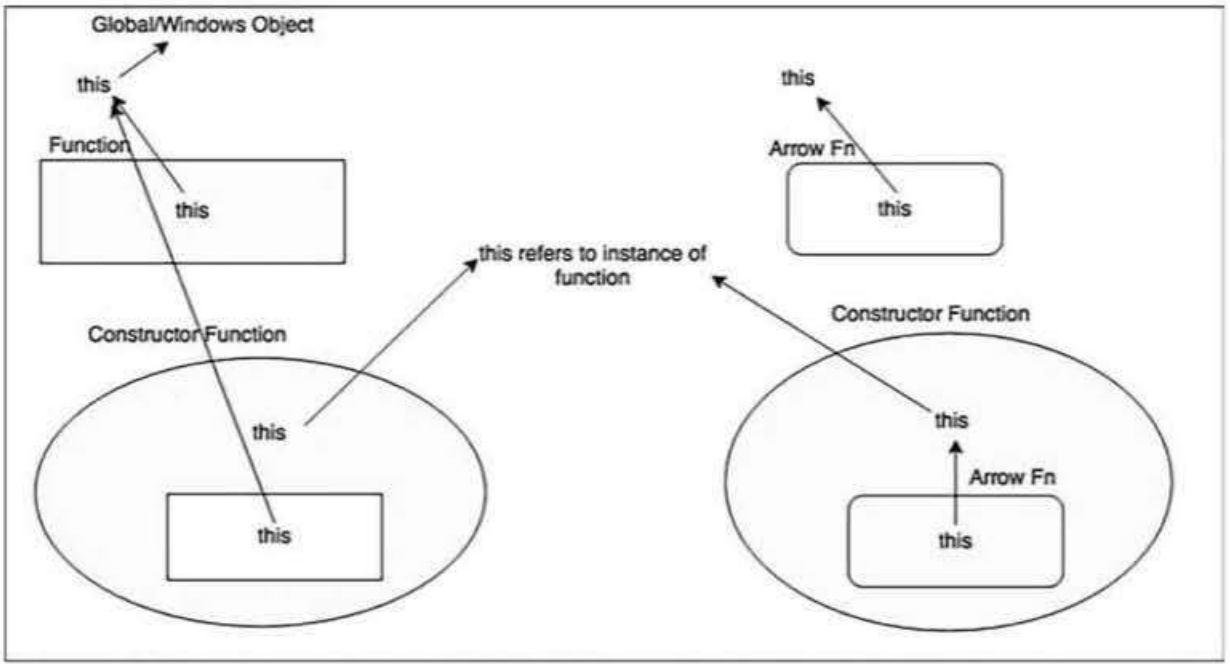
Стрелочная функция — это новый способ определения функции в ECMAScript 6.

```
// традиционный способ объявления и определения функции
var sum = function(a,b) {
    return a+b;
}

// Стрелочная функция
let sum = (a, b) => a+b;

// Определение функции с использованием нескольких строк
let checkIfEven = (a) => {
    if( a % 2 == 0 )
        return true;

    else
        return false;
}
```

this в стрелочных функциях

`this` в функции ссылается на объект-экземпляр, который использовался для вызова этой функции, но `this` в стрелочной функции равен `this` функции, в которой определена стрелочная функция.

Давайте попробуем разобраться с помощью диаграммы:



А теперь рассмотрим использование this на примере:

```
var normalFn = function(){
      console.log(this) // ссылается на глобальный объект (global/
window).
}
var arrowFn = () => console.log(this); // ссылается на глобальный объ-
ект, так как функция определена в области глобального объекта
var service = {
    constructorFn : function(){
       console.log(this); // ссылается на объект service, так как объ-
ект service используется для вызова метода.

        var nestedFn = function(){
            console.log(this); // ссылается на глобальный объект, по-
тому что объект-экземпляр не использовался для вызова этого метода.
        }
        nestedFn();
    },

    arrowFn : function(){
       console.log(this); // ссылается на объект service, так как объ-
ект service использовался для вызова метода.

          let fn = () => console.log(this); // ссылается на объект
service, так как стрелочная функция определена в функции, которая вы-
зывается с использованием объекта-экземпляра.
```

```
        fn();
    }
}

// вызов определенных функций
constructorFn();
arrowFn();
service.constructorFn();
service.arrowFn();
```

В стрелочной функции `this` — это лексическое окружение, которое представляет собой область видимости функции, в которой определена стрелочная функция.

Первый пример — это традиционный способ определения функций, и, соответственно, `this` ссылается на глобальный объект (global/window).

Во втором примере `this` используется внутри стрелочной функции, поэтому `this` ссылается на область, в которой она определена (что является глобальным объектом или объектом window). В третьем примере `this` ссылается на объект service, так как объект service используется для вызова функции.

В четвертом примере стрелочная функция определена и вызывается из функции, область видимости которой — объект service, поэтому она выводит объект service.

Примечание: в Node.js выводится глобальный объект, а в браузере — объект window.

Глава 67.
Node.js с CORS

Примеры

Включение CORS в express.js

Так как Node.js часто используется для создания API, правильная настройка CORS может стать спасением, если вы хотите иметь возможность отправлять запросы к API с разных доменов.

В этом примере мы настроим CORS для самой широкой конфигурации (разрешение всех типов запросов с любого домена).

В вашем `server.js` после инициализации `express`:

```
// Создание сервера express
const app = express();

app.use((req, res, next) => {
    res.header('Access-Control-Allow-Origin', '*');

    // разрешенные заголовки для preflight запросов
    // https://developer.mozilla.org/en-US/docs/Glossary/preflight_
request
    res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-
With, Content-Type, Accept');
    next();

    app.options('*', (req, res) => {
        // разрешенные методы XHR
        res.header('Access-Control-Allow-Methods', 'GET, PATCH, PUT,
POST, DELETE, OPTIONS');
        res.send();
    });
});
```

Обычно Node.js работает за прокси-сервером на продакшн-серверах. Следовательно, сервер обратного проксирования (например, Apache или Nginx) будет ответственен за настройку CORS.

Чтобы удобно адаптировать этот сценарий, можно включить CORS в Node.js только в режиме разработки.

Это легко сделать, проверив переменную окружения `NODE_ENV`:

```
const app = express();

if (process.env.NODE_ENV === 'development') {
    // настройки CORS
}
```

Глава 68.
Node.js с ES6

Введение

ES6, ECMAScript 6 или ES2015 — это спецификация JavaScript, которая позволяет использовать синтаксический сахар. Это большое обновление языка, которое предоставляет множество новых возможностей.

Более подробную информацию о Node и ES6 можно найти на сайте: https://nodejs.org/en/learn/getting-started/ecmascript-2015-es6-and-beyond.

Примеры

Поддержка ES6 в Node.js и создание проекта с Babel

Стандарт ES6 еще не реализован полностью, поэтому вы сможете использовать только некоторые из новых возможностей. Вы можете увидеть список текущих поддерживаемых функций ES6 на сайте http://node.green/.

Начиная с Node.js версии 6 появилась довольно хорошая поддержка. Поэтому, если вы работаете с Node.js версии 6 или выше, вы можете наслаждаться использованием ES6. Однако, возможно, вы так-

же захотите задействовать некоторые нереализованные функции и некоторые функции из будущих стандартов. Для этого вам потребуется транспайлер (транскомпилятор).

Можно запустить транспайлер во время выполнения и сборки, чтобы использовать все функции ES6 и даже больше. Самый популярный транспайлер для JavaScript называется Babel.

Babel позволяет вам использовать все функции из спецификации ES6 и некоторые дополнительные, не входящие в стандарт, функции с 'stage-0', такие как `import thing from 'thing'` вместо `var thing = require('thing')`.

Если мы захотим создать проект, в котором будем использовать функции `'stage-0'`, такие как `import`, нам нужно будет добавить `Babel` в качестве транспайлера. Вы увидите, что проекты, использующие `React` и `Vue`, а также другие шаблоны на основе `CommonJS`, часто реализуют `'stage-0'`.

```
mkdir my-es6-app
cd my-es6-app
npm init
```

Установите babel, пресет ES6 и stage-0:

```
npm install --save-dev babel-preset-es2015 babel-preset-stage-2 babel-
cli babel-register
```

Создайте новый файл с именем `server.js` и добавьте базовый HTTP-сервер:

```
import http from 'http'
http.createServer((req, res) => {
    res.writeHead(200, {'Content-Type': 'text/plain'})
    res.end('Hello World\n')
}).listen(3000, '127.0.0.1')
console.log('Server running at http://127.0.0.1:3000/')
```

Обратите внимание, что мы используем `import http from 'http'` — это функция `'stage-0'`, и если она работает, это значит, что мы правильно настроили транспайлер.

Если вы запустите `node server.js`, он завершится с ошибкой, не зная, как обрабатывать `import`.

Создайте файл .babelrc в корневом каталоге вашего проекта и добавьте следующие настройки:

```
{
    "presets": ["es2015", "stage-2"],
    "plugins": []
}
```

Теперь вы можете запустить сервер с помощью команды:

```
node src/index.js --exec babel-node
```

В завершение, запуск транспайлера во время выполнения на продакшн-приложении — не лучшая идея. Однако мы можем реализовать несколько скриптов в нашем package.json, чтобы облегчить работу.

```
"scripts": {
    "start": "node dist/index.js",
    "dev": "babel-node src/index.js",
    "build": "babel src -d dist",
    "postinstall": "npm run build"
},
```

Вышеописанное будет компилировать код в каталог dist при установке npm и позволять npm start использовать транспилированный код для нашего продакшн-приложения.

Команда npm run dev запустит сервер и Babel во время выполнения, что вполне допустимо и предпочтительно при работе над проектом локально.

Пойдя дальше, вы можете установить nodemon:

```
npm install nodemon --save-dev
```

для отслеживания изменений и перезапуска node-приложения.

Это действительно ускоряет работу с Babel и Node.js. В вашем package.json просто обновите скрипт dev, чтобы использовать nodemon:

```
"dev": "nodemon src/index.js --exec babel-node",
```

Использование ES6 в вашем Node.js-приложении

ES6 (также известный как ES2015) — это набор новых возможностей для языка JavaScript, направленный на то, чтобы сделать его более интуитивно понятным при использовании ООП или при решении современных задач разработки.

Предварительные условия

1. Ознакомьтесь с новыми возможностями ES6 на сайте http://es6-features.org — это может помочь вам решить, действительно ли вы хотите использовать его в вашем следующем Node.js-приложении.

2. Проверьте уровень совместимости вашей версии Node на сайте http://node.green.

3. Если все в порядке — давайте кодировать!

Вот очень короткий пример простого приложения «Hello World» с использованием ES6:

```
'use strict'

class Program {
    constructor() {
        this.message = 'hello es6 :)';
    }

    print() {
        setTimeout(() => {
            console.log(this.message);

            this.print();

        }, Math.random() * 1000);
    }
}

new Program().print();
```

Вы можете запустить эту программу и наблюдать, как она выводит одно и то же сообщение снова и снова.

Теперь давайте разберем ее построчно:

```
'use strict'
```

Эта строка необходима, если вы собираетесь использовать ES6. Строгий (strict) режим намеренно имеет другие семантические особенности по сравнению с обычным кодом.

```
class Program
```

Невероятно — ключевое слово `class`! Для справки: до ES6 единственным способом определения класса в JavaScript было использование ключевого слова `function`:

```
function MyClass() { // определение класса

}

var myClassObject = new MyClass();
// создание нового объекта типа MyClass
```

При использовании ООП класс — это очень важная возможность, которая помогает разработчику представлять определенную часть системы (разбиение кода на части критично, когда код становится большим, например, при написании серверного кода):

```
constructor() {

    this.message = 'hello es6 :)';
}
```

Признайте, это довольно интуитивно! Это конструктор класса — данная уникальная «функция» будет выполняться каждый раз, когда создается объект из этого конкретного класса (в нашей программе — только один раз):

```
print() {

    setTimeout(() => // это стрелочная функция
    {
        console.log(this.message);

        this.print(); // здесь мы вызываем метод 'print' из самого
шаблона класса (в данном случае это рекурсия)

    }, Math.random() * 1000);
}
```

Так как `print` определен в области видимости класса, это фактически метод, который можно вызывать как от объекта класса, так и из самого класса!

Итак, до сих пор мы определяли наш класс. Пришло время его использовать:

```
new Program().print();
```

...что эквивалентно:

```
var prog = new Program(); // определение нового объекта типа 'Program'

prog.print(); // использование программы для самовывода
```

В заключение: ES6 может упростить ваш код, сделать его более интуитивно понятным и легким для понимания по сравнению с предыдущей версией JavaScript. Попробуйте переписать свой существующий код, и вы сразу увидите разницу.

НАСЛАЖДАЙТЕСЬ :)

Глава 69.
Использование Node.js с Oracle

Примеры

Подключение к Oracle DB

Очень простой способ подключения к базе данных ORACLE — использование модуля `oracledb`. Этот модуль управляет подключением между вашим приложением на Node.js и сервером Oracle. Вы можете установить его, как любой другой модуль:

```
npm install oracledb
```

Теперь вам нужно создать подключение к ORACLE, которое вы сможете использовать для выполнения запросов

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

oracledb.getConnection(
 {
    user : "oli",
    password : "password",
    connectString : "ORACLE_DEV_DB_TNS_NAME"
 },

 connExecute
);
```

`connectString` «ORACLE_DEV_DB_TNA_NAME» может находиться в файле `tnsnames.org` в том же каталоге или в месте, где установлен ваш Oracle Instant Client.

Если на вашей машине для разработки не установлен клиент Oracle Instant, вы можете следовать руководству по установке Instant Client для вашей операционной системы.

Выполнение запроса на объекте соединения без параметров

Теперь вы можете использовать функцию connExecute для выполнения запроса. У вас есть возможность получить результат запроса как объект или массив. Результат выводится в console.log:

```
function connExecute(err, connection)
{
    if (err) {
        console.error(err.message);
        return;
    }

    sql = "select 'test' as c1, 'oracle' as c2 from dual";

     connection.execute(sql, {}, { outFormat: oracledb.OBJECT }, //
или oracledb.ARRAY
    function(err, result)

    {
        if (err) {
            console.error(err.message);
            connRelease(connection);
            return;
        }
        console.log(result.metaData);
        console.log(result.rows);
        connRelease(connection);

    });
}
```

Так как мы использовали соединение без пула, необходимо освободить его после использования:

```
function connRelease(connection)

{
    connection.close(
    function(err) {
        if (err) {
            console.error(err.message);

        }
    });
}
```

Вывод для объекта будет таким:

```
[ { name: 'C1' }, { name: 'C2' } ]
[ { C1: 'test', C2: 'oracle' } ]
```

А вывод для массива будет следующим:

```
[ { name: 'C1' }, { name: 'C2' } ]
[ [ 'test', 'oracle' ] ]
```

Использование локального модуля для упрощения запросов

Чтобы упростить выполнение запросов из ORACLE-DB, вы можете вызывать свой запрос следующим образом:

```
const oracle = require('./oracle.js');
const sql = "select 'test' as c1, 'oracle' as c2 from dual";
oracle.queryObject(sql, {}, {})
    .then(function(result) {
        console.log(result.rows[0]['C2']);
    })
    .catch(function(err) {
        next(err);
    });
```

Создание соединения и выполнение запроса включено в файл `oracle.js` со следующим содержанием:

```
'use strict';
const oracledb = require('oracledb');
const oracleDbRelease = function(conn) {
    conn.release(function (err) {
        if (err)
            console.log(err.message);
    });
};
function queryArray(sql, bindParams, options) {
   options.isAutoCommit = false; // мы выполняем только SELECT-запросы
    return new Promise(function(resolve, reject) {
        oracledb.getConnection(
        {
            user : "oli",
            password : "password",
            connectString : "ORACLE_DEV_DB_TNA_NAME"
        })
        .then(function(connection){
```

```
            //console.log("sql log: " + sql + " params " + bindParams);
             connection.execute(sql, bindParams, options)
             .then(function(results) {
                 resolve(results);
                 process.nextTick(function() {
                     oracleDbRelease(connection);
                 });
             })
             .catch(function(err) {
                 reject(err);
                 process.nextTick(function() {
                     oracleDbRelease(connection);
                 });
             });
         })
         .catch(function(err) {
             reject(err);
         });
    });
}
function queryObject(sql, bindParams, options) {
    options['outFormat'] = oracledb.OBJECT; // по умолчанию oracledb.
ARRAY
    return queryArray(sql, bindParams, options);
}
module.exports = queryArray;
module.exports.queryArray = queryArray;
module.exports.queryObject = queryObject;
```

Обратите внимание, что у вас есть два метода — `queryArray` и `queryObject` — для вызова на вашем объекте `oracle`.

Глава 70.
Руководство для начинающих по Node.js

Примеры

Hello World!

Разместите следующий код в файле с именем `helloworld.js`:

```
console.log("Hello World");
```

Сохраните файл и выполните его через Node.js:

```
node helloworld.js
```

Глава 71.
Фреймворки Node.js

Примеры

Фреймворки для веб-серверов

Express

```
var express = require('express');
var app = express();
app.get('/', function (req, res) {
    res.send('Hello World!');
});

app.listen(3000, function () {
    console.log('Example app listening on port 3000!');
});
```

Koa

```
var koa = require('koa');
var app = koa();
app.use(function *(next){
    var start = new Date;
    yield next;
    var ms = new Date - start;
    console.log('%s %s - %s', this.method, this.url, ms);
});

app.use(function *(){
    this.body = 'Hello World';
});
app.listen(3000);
```

Фреймворки для интерфейса командной строки

Commander.js

```
var program = require('commander');
program
    .version('0.0.1')
program
    .command('hi')
    .description('initialize project configuration')
    .action(function(){
        console.log('Hi my Friend!!!');
    });
program
    .command('bye [name]')
    .description('initialize project configuration')
    .action(function(name){
        console.log('Bye ' + name + '. It was good to see you!');
    });
program
    .command('*')
    .action(function(env){
        console.log('Enter a Valid command');
        terminate(true);
    });
program.parse(process.argv);
```

Vorpal.js

```
const vorpal = require('vorpal')();
vorpal
    .command('foo', 'Outputs "bar".')
    .action(function(args, callback) {
        this.log('bar');
        callback();
    });
vorpal
    .delimiter('myapp$')
    .show();
```

Глава 72.
История Node.js

Введение

Рассмотрим историю Node.js и информацию о версиях.

Ключевые события по годам

2009

- 3 марта — проект был назван «node».
- 1 октября — первый предварительный просмотр npm, менеджера пакетов Node.
- 8 ноября — оригинальный доклад Райана Даля (создателя Node.js) о Node.js на JSConf 2009.

2010

- Express: фреймворк для веб-разработки на Node.js.
- Первоначальный выпуск Socket.io.
- 28 апреля — экспериментальная поддержка Node.js на Heroku.
- 28 июля — технический доклад Райана Даля о Node.js в Google.
- 20 августа — выпуск Node.js 0.2.0.

2011

- 31 марта — руководство по Node.js.
- 1 мая — npm 1.0: выпуск.

- 1 мая — AMA («спроси меня о чем угодно») с Райаном Далем на Reddit.
- 10 июля — Завершена работа над книгой «The Node Beginner Book», введение в Node.js.
    - Комплексный учебник по Node.js для начинающих.
- 16 августа — LinkedIn использует Node.js.
    - LinkedIn запустил полностью обновленное мобильное приложение с новыми функциями и новыми компонентами.
- 5 октября — Райан Даль рассказывает об истории Node.js и о том, почему он его создал.
- 5 декабря — Node.js в производстве в Uber.
    - Менеджер по разработке Uber Кёртис Чамберс объясняет, почему его компания полностью перестроила свое приложение с использованием Node.js для повышения эффективности и улучшения опыта партнеров и клиентов.

2012

- 30 января — создатель Node.js Райан Даль уходит от повседневных дел, связанных с Node.
- 25 июня — вышла Node.js v0.8.0 [стабильная версия].
- 20 декабря — выпущен Hapi — фреймворк Node.js.

2013

- 30 апреля — стек MEAN: MongoDB, ExpressJS, AngularJS и Node.js.
- 17 мая — как мы создали первое приложение eBay на Node.js.
- 15 ноября — PayPal выпускает Kraken, фреймворк Node.js.
- 22 ноября — утечка памяти в Node.js в Walmart.
    - Эран Хаммер из лабораторий Walmart пришел к основной команде Node.js, жалуясь на утечку памяти, которую он отслеживал месяцами.
- 19 декабря — Koa — веб-фреймворк для Node.js.

2014

- 15 января — TJ Fontaine берет на себя проект Node.

- 23 октября — консультативный совет Node.js.
    - Joyent и несколько членов сообщества Node.js объявили предложение о создании Консультативного совета Node.js как следующего шага к полностью открытой модели управления проектом с открытым исходным кодом.
- 19 ноября — Node.js в Flame Graphs — Netflix.
- 28 ноября — IO.js — асинхронный I/O для V8 Javascript.

2015

Q1

- 14 января — IO.js 1.0.0.
- 10 февраля — Joyent стремится создать Node.js Foundation.
    - Joyent, IBM, Microsoft, PayPal, Fidelity, SAP и The Linux Foundation объединяют усилия, чтобы поддержать сообщество Node.js с нейтральным и открытым управлением.
- 27 февраля — предложение о примирении IO.js и Node.js.

Q2

- 14 апреля — приватные модули npm.
- 28 мая — руководитель проекта Node TJ Fontaine уходит и покидает Joyent.
- 13 мая — Node.js и io.js объединяются под эгидой Node Foundation.

Q3

- 2 августа — Trace — мониторинг производительности и отладка Node.js.
    - Trace — это визуализированный инструмент мониторинга микросервисов, который предоставляет все необходимые метрики при работе с микросервисами.
- 13 августа — 4.0 — это новая 1.0.

Q4

- 12 октября — Node v4.2.0, первый выпуск с долгосрочной поддержкой.

- 8 декабря — Apigee, RisingStack и Yahoo присоединяются к Node.js Foundation.
- 8 и 9 декабря — Node Interactive.
    - Первая ежегодная конференция Node.js от Node.js Foundation.

2016

Q1

- 10 февраля — Express становится инкубированным проектом.
- 23 марта — инцидент с leftpad.
- 29 марта — Google Cloud Platform присоединяется к Node.js Foundation.

Q2

- 26 апреля — у npm 210 000 пользователей.

Q3

- 18 июля — CJ Silverio становится CTO npm.
- 1 августа — Trace, решение для отладки Node.js, становится общедоступным.
- 15 сентября — первая Node Interactive в Европе.

Q4

- 11 октября — выпущен менеджер пакетов yarn.
- 18 октября — Node.js 6 становится версией с долгосрочной поддержкой.

Ссылка
«История Node.js на временной шкале»: https://blog.risingstack.com/history-of-node-js/.

Глава 73.
Маршрутизация в Node.js

Введение

Рассмотрим, как настроить базовый веб-сервер Express под Node.js, и исследуем маршрутизатор Express.

Примечания

В конечном итоге, задействуя маршрутизатор Express, вы можете использовать функции маршрутизации в своем приложении, и это легко реализовать.

Примеры

Маршрутизация веб-сервера Express

Создание веб-сервера Express

Сервер Express оказался удобным и получил широкое распространение. С каждым днем он становится все более популярным. Давайте создадим сервер Express. Для управления пакетами и гибкости в управлении зависимостями мы будем использовать NPM (Node Package Manager).

1. Перейдите в каталог проекта и создайте файл `package.json`.

`package.json`

```
{
  "name": "expressRouter",
  "version": "0.0.1",
  "scripts": {
    "start": "node Server.js"
  },
```

```
  "dependencies": {
    "express": "^4.12.3"
  }
}
```

2. Сохраните файл и установите зависимость Express, используя команду `npm install`. Это создаст папку `node_modules` в вашем каталоге проекта вместе с необходимыми зависимостями.

3. Создайте веб-сервер Express: перейдите в каталог проекта и создайте файл `server.js`.

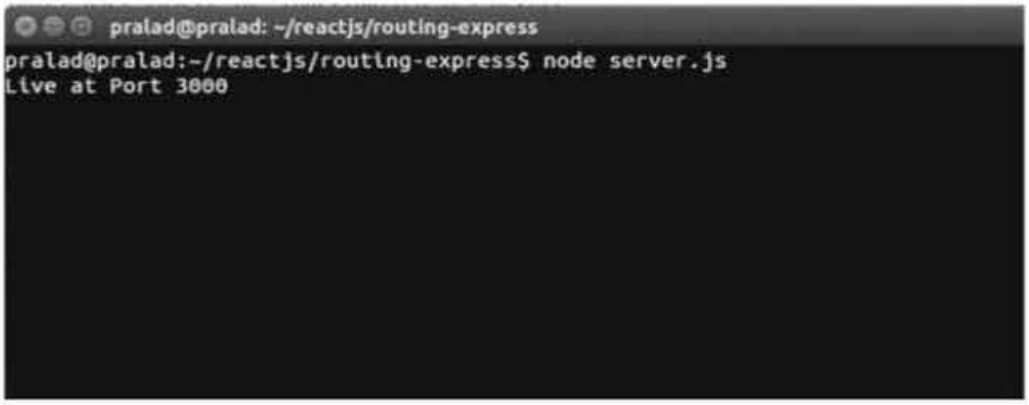
`server.js`

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

// Создание объекта Router()
var router = express.Router();

// Обеспечьте все маршруты здесь, это для главной страницы.
router.get("/", function(req, res) {
    res.json({"message" : "Hello World"});
});

app.use("/api", router);

// Слушайте этот порт
app.listen(3000, function() {
    console.log("Live at Port 3000");
});
```

4. Запустите сервер, введя следующую команду:

```
node server.js
```

Если сервер запущен успешно, вы увидите что-то вроде этого:



5. Теперь откройте браузер или Postman и сделайте запрос к http://localhost:3000/api/.

Вывод будет таким:

Это все, мы познакомились с основами маршрутизации Express.

Теперь давайте обработаем GET, POST и другие запросы.

Измените ваш файл `server.js` следующим образом:

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

// Создание объекта Router()
var router = express.Router();

// Промежуточное ПО маршрутизатора, упомянутое перед определением
маршрутов.
router.use(function(req, res, next) {
    console.log("/" + req.method);
    next();
});

// Обеспечьте все маршруты здесь, это для главной страницы.
router.get("/", function(req, res) {
    res.json({"message" : "Hello World"});
});

app.use("/api", router);

app.listen(3000, function() {
    console.log("Live at Port 3000");
});
```

Доступ к параметрам в маршрутизации

Вы также можете получить доступ к параметрам из URL, например, `http://example.com/api/:name/`. Таким образом можно получить доступ к параметру `name`. Добавьте следующий код в ваш файл `server.js`:

```
router.get("/user/:id", function(req, res) {
    res.json({"message" : "Hello " + req.params.id});
});
```

Теперь перезапустите сервер и перейдите по [`http://localhost:3000/api/user/Adem`][4], вывод будет таким:

Глава 74.
Node.js с Redis

Замечания

Рассмотрим основные и наиболее часто используемые операции в `node_redis`. Вы можете использовать этот модуль, чтобы задействовать всю мощь Redis и создавать действительно сложные приложения на Node.js. С этой библиотекой становятся доступными множество интересных вещей, таких как надежный слой кеширования, мощная система обмена сообщениями Pub/Sub и многое другое. Чтобы узнать больше о библиотеке, ознакомьтесь с документацией: https://www.npmjs.com/package/redis.

Примеры

Начало работы

`node_redis`, как вы могли догадаться, является клиентом Redis для Node.js. Вы можете установить его через `npm`, используя следующую команду:

```
npm install redis
```

Как только вы установили модуль `node_redis`, вы готовы к работе. Давайте создадим простой файл `app.js` и посмотрим, как подключиться к Redis из Node.js.

app.js

```
var redis = require('redis');
client = redis.createClient(); //создает нового клиента
```

По умолчанию `redis.createClient()` будет использовать `127.0.0.1` и 6379 в качестве хоста и порта соответственно. Если у вас другой хост/порт, вы можете указать их следующим образом:

```
var client = redis.createClient(port, host);
```

Теперь вы можете выполнить какое-то действие, после того как соединение будет установлено. В основном вам просто нужно прослушивать события подключения, как показано ниже.

```
client.on('connect', function() {
    console.log('connected');
});
```

Итак, следующий фрагмент кода добавляется в `app.js`:

```
var redis = require('redis');
var client = redis.createClient();
client.on('connect', function() {
    console.log('connected');
});
```

Теперь введите `node app` в терминале, чтобы запустить приложение. Перед выполнением этого кода убедитесь, что ваш сервер Redis запущен.

Сохранение пар «ключ-значение»

Теперь, когда вы знаете, как подключиться к Redis из Node.js, давайте посмотрим, как сохранять пары «ключ-значение» в хранилище Redis.

Сохранение строк

Все команды Redis представлены в виде различных функций на объекте `client`. Чтобы сохранить простую строку, используйте следующий синтаксис:

```
client.set('framework', 'AngularJS');
```

или

```
client.set(['framework', 'AngularJS']);
```

В приведенных выше примерах строка 'AngularJS' сохраняется с ключом 'framework'. Обратите внимание, что оба примера делают одно и то же. Единственное различие в том, что первый передает переменное количество аргумен-

тов, в то время как второй передает массив args в функцию client.set(). Вы также можете передать необязательный обратный вызов, чтобы получить уведомление, когда операция будет завершена:

```
client.set('framework', 'AngularJS', function(err, reply) {
    console.log(reply);
});
```

Если операция не удалась по какой-либо причине, аргумент err в обратном вызове будет содержать ошибку. Чтобы получить значение ключа, выполните следующее:

```
client.get('framework', function(err, reply) {
    console.log(reply);
});
```

Функция client.get() позволяет получить ключ, сохраненный в Redis. Значение ключа можно получить через аргумент обратного вызова reply. Если ключ не существует, значение reply будет пустым.

Сохранение хешей

Часто сохранение простых значений не решает вашу задачу. Вам может понадобиться сохранять хеши (объекты) в Redis. Для этого вы можете использовать функцию hmset(), как показано ниже:

```
client.hmset('frameworks', 'javascript', 'AngularJS', 'css',
'Bootstrap', 'node', 'Express');
client.hgetall('frameworks', function(err, object) {
    console.log(object);
});
```

Приведенный выше пример сохраняет хеш в Redis, который сопоставляет каждую технологию с ее фреймворком. Первый аргумент для hmset() — это имя ключа. Последующие аргументы представляют пары «ключ-значение». Аналогично hgetall() используется для получения значения ключа. Если ключ найден, второй аргумент в обратном вызове будет содержать значение, которое является объектом.

Учтите, что Redis не поддерживает вложенные объекты. Все значения свойств в объекте будут преобразованы в строки перед сохранением. Вы также можете использовать следующий синтаксис для сохранения объектов в Redis:

```
client.hmset('frameworks', {
    'javascript': 'AngularJS',
    'css': 'Bootstrap',
    'node': 'Express'
});
```

Необязательный обратный вызов также может быть передан для получения уведомления, когда операция будет завершена.

Все функции (команды) могут быть вызваны с использованием эквивалентов в верхнем/нижнем регистре. Например, `client.hmset()` и `client.HMSET()` — это одно и то же.

Сохранение списков

Если вы хотите сохранить список элементов, вы можете использовать списки Redis. Чтобы сохранить список, используйте следующий синтаксис:

```
client.rpush(['frameworks', 'angularjs', 'backbone'], function(err,
reply)

{
    console.log(reply); // выводит 2
});
```

Приведенный выше пример создает список с именем `frameworks` и добавляет в него два элемента. Таким образом, длина списка теперь равна двум. Как вы можете заметить, мы передали массив `args` в `rpush`. Первый элемент массива представляет имя ключа, а остальные представляют элементы списка. Вы также можете использовать `lpush()` вместо `rpush()`, чтобы добавлять элементы слева.

Чтобы получить элементы списка, вы можете использовать функцию `lrange()`, как показано ниже:

```
client.lrange('frameworks', 0, -1, function(err, reply) {
    console.log(reply); // ['angularjs', 'backbone']
});
```

Обратите внимание, что вы получаете все элементы списка, передавая -1 в качестве третьего аргумента в `lrange()`. Если вам нужно подмножество списка, вы должны передать конечный индекс.

Сохранение множеств

Множества похожи на списки, но разница в том, что они не допускают дубликатов. Таким образом, если вы не хотите иметь повторяющихся элементов в вашем списке, вы можете использовать множество. Вот как мы можем изменить наш предыдущий пример, чтобы использовать множество вместо списка:

```
client.sadd(['tags', 'angularjs', 'backbonejs', 'emberjs'],
function(err, reply) {

    console.log(reply); // выводит 3
});
```

Как видите, функция sadd() создает новое множество с указанными элементами. Здесь длина множества равна трем. Чтобы получить элементы множества, используйте функцию smembers(), как показано ниже:

```
client.smembers('tags', function(err, reply) {
    console.log(reply);
});
```

Этот фрагмент кода получит все элементы множества. Обратите внимание, что порядок не сохраняется при получении элементов.

Это был список наиболее важных структур данных, используемых в каждом приложении, работающем на Redis. Помимо строк, списков, множеств и хешей вы можете сохранять упорядоченные множества, hyperLogLogs и многое другое в Redis. Если вы хотите получить полный список команд и структур данных, изучите официальную документацию по Redis. Помните, что почти каждая команда Redis доступна на объекте client, предоставляемом модулем node_redis.

Некоторые дополнительные операции, поддерживаемые node_redis

Проверка существования ключей

Иногда может понадобиться проверка того, существует ли ключ, чтобы действовать соответственно. Для этого можно использовать функцию exists(), как показано ниже:

```
client.exists('key', function(err, reply) {
    if (reply === 1) {
        console.log('exists');
    } else {
        console.log('doesn\'t exist');
    }
});
```

Удаление и установка времени жизни ключей

Иногда вам нужно очистить некоторые ключи и инициализировать их заново. Чтобы удалить ключи, вы можете использовать команду del, как показано ниже:

```
client.del('frameworks', function(err, reply) {
    console.log(reply);
});
```

Вы также можете задать время истечения срока действия для существующего ключа следующим образом:

```
client.set('key1', 'val1');
```

```
client.expire('key1', 30);
```

Приведенный выше пример устанавливает время истечения срока действия 30 секунд для ключа key1.

Увеличение и уменьшение значений

Redis также поддерживает увеличение и уменьшение значений ключей. Чтобы увеличить значение ключа, используйте функцию `incr()`, как показано ниже:

```
client.set('key1', 10, function() {
    client.incr('key1', function(err, reply) {
        console.log(reply); // выводит 11
    });
});
```

Функция `incr()` увеличивает значение ключа на 1. Если вам нужно увеличить его на другое количество, вы можете использовать функцию `incrby()`. Аналогично, для уменьшения значения ключа можно использовать функции `decr()` и `decrby()`.

Глава 75.
Node Package Manager (npm)

Введение

Node Package Manager (npm) предоставляет две основные функции:

- Онлайн-репозитории для пакетов/модулей Node.js, которые можно найти на сайте search.nodejs.org.
- Утилита командной строки для установки пакетов Node.js, управления версиями и зависимостями пакетов Node.js.

Синтаксис

npm <command>, где <command> — одна из следующих команд:

- add-user
- adduser
- apihelp
- author
- bin
- bugs
- c
- cache
- completion
- config
- ddp
- dedupe
- deprecate
- docs
- edit
- explore
- faq
- find
- find-dupes
- get
- help
- help-search
- home
- i
- install
- info
- init
- isntall
- issues
- la
- link
- list
- ll
- ln
- login
- ls

- outdated
- owner
- pack
- prefix
- prune
- publish
- r
- rb
- rebuild
- remove
- repo
- restart
- rm
- root
- run-script
- s
- se
- search
- set
- show
- shrinkwrap
- star
- stars
- start
- stop
- submodule
- tag
- test
- tst
- un
- uninstall
- unlink
- unpublish
- unstar
- up
- update
- v
- version
- view
- whoami

Параметры

Параметр	Пример
access	npm publish --access=public
bin	npm bin -g
edit	npm edit connect
help	npm help init
init	npm init
install	npm install
link	npm link
prune	npm prune
publish	npm publish ./
restart	npm restart
start	npm start
stop	npm start
update	npm update
version	npm version

Примеры

Установка пакетов

Введение

Пакет — это термин, используемый в `npm` для обозначения инструментов, которые разработчики могут задействовать в своих проектах. Это включает в себя все, начиная от библиотек и фреймворков, таких как jQuery и AngularJS, и заканчивая такими инструментами как `Gulp.js`. Пакеты размещаются в папке, которая обычно называется `node_modules` и содержит файл `package.json`. Этот файл содержит информацию обо всех пакетах, включая любые зависимости, которые являются дополнительными модулями, необходимыми для использования конкретного пакета.

Npm использует командную строку как для установки, так и для управления пакетами, поэтому пользователи, желающие освоить его, должны быть знакомы с основными командами своей операционной системы, например, с переходом по директориям, а также с возможностью видеть содержимое директорий.

Установка NPM

Обратите внимание, что для установки пакетов необходимо, чтобы NPM был установлен.

Рекомендуемый способ установки NPM — использование одного из установочных пакетов со страницы загрузки Node.js: https://nodejs.org/en/download/package-manager.

Вы можете проверить, установлена ли у вас уже Node.js, выполнив команду `npm -v` или `npm version`.

После установки NPM через установочный пакет Node.js обязательно проверьте наличие обновлений. Это связано с тем, что NPM обновляется чаще, чем установочный пакет Node.js. Чтобы проверить наличие обновлений, выполните следующую команду:

```
npm install npm@latest -g
```

Как установить пакеты

Чтобы установить один или несколько пакетов, используйте следующую команду:

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

или

```
npm i <package-name>...
```

например, чтобы установить `lodash` и `express`

```
npm install lodash express
```

> ***Примечание:*** эта команда установит пакет в каталоге, в котором в данный момент находится командная строка, поэтому важно убедиться, что выбран правильный каталог.

Если в вашем текущем рабочем каталоге уже есть файл `package.json` и в нем определены зависимости, то `npm install` автоматически разрешит и установит все зависимости, перечисленные в файле. Вы также можете использовать сокращенную версию команды `npm install`, которая выглядит так: `npm i`.

Если вы хотите установить конкретную версию пакета, используйте:

```
npm install <name>@<version>
```

Например, чтобы установить версию 4.11.1 пакета `lodash`:

```
npm install lodash@4.11.1
```

Если вы хотите установить версию, которая соответствует определенному диапазону версий, используйте:

```
npm install <name>@<version range>
```

Например, чтобы установить версию, которая соответствует «version >= 4.10.1» и «version < 4.11.1» пакета `lodash`:

```
npm install lodash@">=4.10.1 <4.11.1"
```

Если вы хотите установить последнюю версию, используйте:

```
npm install <name>@latest
```

Команды выше будут искать пакеты в центральном репозитории `npm` на сайте https://www.npmjs.com/.

Если вы не хотите устанавливать из реестра `npm`, поддерживаются и другие варианты, такие как:

```
# пакеты, распространяемые в виде tarball
npm install <tarball file>
npm install <tarball url>

# пакеты, доступные локально
npm install <local path>

# пакеты, доступные как git-репозиторий
npm install <git remote url>

# пакеты, доступные на GitHub
npm install <username>/<repository>

# пакеты, доступные в виде gist (требуется package.json)
npm install gist:<gist-id>

# пакеты из конкретного репозитория
npm install --registry=http://myreg.mycompany.com <package name>

# пакеты из определенной группы пакетов
# см. npm scope
npm install @<scope>/<name>(@<version>)

#"Скоупинг" полезен для отделения приватных пакетов, размещенных в при-
ватном
#реестре, от публичных, устанавливая реестр для определенной области
видимости:
npm config set @mycompany:registry http://myreg.mycompany.com

npm install @mycompany/<package name>
```

Обычно модули устанавливаются локально в папке `node_modules`, которую можно найти в вашем текущем рабочем каталоге. Это тот каталог, который будет использовать `require()` для загрузки модулей, чтобы сделать их доступными для вас.

Если вы уже создали файл `package.json`, вы можете использовать опцию `--save` (сокращение `-s`) или один из ее вариантов, чтобы автоматически добавить установленный пакет в ваш `package.json` как зависимость. Если кто-то еще установит ваш пакет, `npm` автоматически прочитает зависимости из файла `package.json` и установит указанные версии. Обратите внимание, что вы все еще можете добавлять и управлять своими зависимостями, редактируя файл позже, поэтому обычно полезно отслеживать зависимости, например, так:

```
npm install --save <name> # Установить зависимости
```

или

```
npm install -s <name> # сокращенная версия -save
```

или

```
npm i -S <name>
```

Чтобы установить пакеты и сохранить их только в том случае, если они нужны для разработки, а не для работы приложения, выполните следующую команду:

```
npm install --save-dev <name> # Установить зависимости для разработки
```

или

```
npm install -D <name> # сокращенная версия --save-dev
```

или

```
npm i -D <name>
```

Установка зависимостей

Некоторые модули не только предоставляют библиотеку для использования, но и содержат один или несколько бинарных файлов, которые предназначены для использования через командную строку. Хотя вы все еще можете установить эти пакеты локально, часто предпочтительнее устанавливать их глобально, чтобы можно было использовать инструменты командной строки. В этом случае npm автоматически свяжет бинарные файлы с соответствующими путями (например, `/usr/local/bin/<name>`), чтобы их можно было запускать из командной строки. Чтобы установить пакет глобально, используйте:

```
npm install --global <name>
```

или

```
npm install -g <name>
```

или

```
npm i -g <name>
```

Например, чтобы установить инструмент командной строки `grunt`:

```
npm install -g grunt-cli
```

Если вы хотите увидеть список всех установленных пакетов и их соответствующих версий в текущем рабочем пространстве, используйте:

```
npm list
npm list <name>
```

Добавление необязательного аргумента имени может проверить версию конкретного пакета.

Примечание: если у вас возникают проблемы с разрешениями при попытке установить модуль npm глобально, не поддавайтесь искушению использовать sudo `npm install -g ...`, чтобы обойти проблему. Предоставление сторонним скриптам возможности выполняться в вашей системе с повышенными привилегиями опасно. Проблема с разрешениями может означать, что у вас есть проблемы с тем, как сам npm был установлен. Если вас интересует установка Node в «песочнице пользователя», вы можете попробовать использовать nvm.

Если у вас есть инструменты сборки или другие зависимости, предназначенные только для разработки (например, Grunt), вы, возможно, не захотите включать их в приложение, которое вы развертываете. В этом случае вы захотите иметь их в качестве зависимости, предназначенной только для разработки, которая указана в `package.json` под `devDependencies`. Чтобы установить пакет как зависимость, предназначенную только для разработки, используйте `--save-dev` (или `-D`).

```
npm install --save-dev <name> // Установить зависимости для разработки, которые не включаются в production
```

или

```
npm install -D <name>
```

Вы увидите, что пакет добавлен в `devDependencies` вашего `package.json`.

Чтобы установить зависимости загруженного/клонированного проекта Node.js, вы можете использовать:

```
npm install
```

или

```
npm i
```

`npm` автоматически прочитает зависимости из `package.json` и установит их.

NPM за прокси-сервером

Если ваш доступ в интернет осуществляется через прокси-сервер, возможно, вам потребуется изменить команды `npm install`, которые обращаются к удаленным репозиториям. `npm` использует файл конфигурации, который можно обновить через командную строку:

```
npm config set
```

Вы можете найти настройки прокси-сервера в панели настроек вашего браузера. Как только вы получите настройки прокси (URL сервера, порт, имя пользователя и пароль), вам нужно настроить конфигурации npm следующим образом.

```
$ npm config set proxy http://<username>:<password>@<proxy-server-url>:<port>

$ npm config set https-proxy http://<username>:<password>@<proxy-server-url>:<port>
```

Поля username, password, port являются необязательными. После того как вы установили эти параметры, команды npm install, npm i -g и другие будут работать корректно.

Скоупы и репозитории

```
# Установить репозиторий для скоупа "myscope"
npm config set @myscope:registry http://registry.corporation.com
# Войти в репозиторий и ассоциировать его со скоупом "myscope"
npm adduser --registry=http://registry.corporation.com --scope=@myscope

# Установить пакет "mylib" из скоупа "myscope"
npm install @myscope/mylib
```

Если имя вашего пакета начинается с @myscope и скоуп myscope ассоциирован с другим репозиторием, npm publish загрузит ваш пакет в этот репозиторий.

```
Вы также можете сохранить эти настройки в файле .npmrc:
@myscope:registry=http://registry.corporation.com

//registry.corporation.com/:_authToken=xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxxxxx
```

Это полезно, к примеру, при автоматизации сборки на CI-сервере.

Удаление пакетов

Чтобы удалить один или несколько локально установленных пакетов, используйте:

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

Команда uninstall в NPM имеет пять алиасов, которые также можно использовать:

```
npm remove <package name>
npm rm <package name>
npm r <package name>
npm unlink <package name>
npm un <package name>
```

Если вы хотите удалить пакет из файла package.json в рамках удаления, используйте флаг --save (сокращение: -S):

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

Для зависимостей разработки используйте флаг --save-dev (сокращение: -D):

```
npm uninstall --save-dev <package name>
npm uninstall -D <package name>
```

Для необязательных зависимостей используйте флаг --save-optional (сокращение: -O):

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

npm uninstall -O <package name>
```

Для пакетов, установленных глобально, используйте флаг --global (сокращение: -g):

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

Основы семантического версионирования

Прежде чем публиковать пакет, необходимо его версионировать. npm поддерживает семантическое версионирование, что означает наличие патчей, минорных и мажорных релизов.

Например, если версия вашего пакета 1.2.3, чтобы изменить версию, необходимо:

1. patch release: `npm version patch` => 1.2.4
2. minor release: `npm version minor` => 1.3.0
3. major release: `npm version major` => 2.0.0

Вы также можете указать версию напрямую:

```
npm version 3.1.4 => 3.1.4
```

Когда вы устанавливаете версию пакета с помощью одной из команд npm, указанных выше, npm изменяет поле версии в файле package.json, выполняет ком-

мит, а также создает новый тег Git с версией с предшествующей «v», как если бы вы выполнили команду:

```
git tag v3.1.4
```

В отличие от других менеджеров пакетов, таких как Bower, регистр npm не требует создания тегов Git для каждой версии. Но если вам нравится использовать теги, вы должны помнить о необходимости отправки нового тега после увеличения версии пакета:

```
git push origin master (для отправки изменений в package.json)
git push origin v3.1.4 (для отправки нового тега)
```

Или вы можете сделать это сразу с помощью команды:

```
git push origin master -tags
```

Настройка конфигурации пакетов

Конфигурации пакетов Node.js содержатся в файле `package.json`, который можно найти в корневом каталоге каждого проекта. Вы можете создать новый файл конфигурации, вызвав

```
npm init
```

Он попытается прочитать текущий рабочий каталог для информации о репозитории Git (если он существует) и переменных окружения, чтобы попытаться автоматически заполнить некоторые значения-заглушки. В противном случае он предоставит диалог ввода для основных параметров.

Если вы хотите создать `package.json` с настройками по умолчанию, используйте:

```
npm init --yes
# или

npm init -y
```

Если вы создаете `package.json` для проекта, который вы не собираетесь публиковать как пакет NPM (например, только для управления вашими зависимостями), вы можете указать это намерение в файле `package.json`:

1. Дополнительно установите свойство `private` в `true`, чтобы предотвратить случайную публикацию.

2. Опционально установите свойство `license` в «UNLICENSED», чтобы запретить другим использовать ваш пакет.

Чтобы установить пакет и автоматически сохранить его в package.json, используйте:

```
npm install --save <package>
```

Пакет и связанная с ним метаинформация (например, версия пакета) появятся в ваших зависимостях. Если вы сохраните его как зависимость для разработки (используя --save-dev), пакет вместо этого появится в ваших devDependencies.

При использовании такого минимального файла package.json вы можете столкнуться с предупреждениями при установке или обновлении пакетов, сообщающими, что у вас отсутствуют описание и поле репозитория. Хотя эти сообщения можно игнорировать, вы можете избавиться от них, открыв package.json в любом текстовом редакторе и добавив следующие строки в объект JSON:

```
[...]
"description": "No description",

"repository": {
    "private": true
},
[...]
```

Публикация пакета

Сначала убедитесь, что вы настроили свой пакет. Затем вам нужно войти в систему npmjs.

Если у вас уже есть пользователь NPM:

```
npm login
```

Если у вас нет пользователя:

```
npm adduser
```

Чтобы проверить, что ваш пользователь зарегистрирован в текущем клиенте:

```
npm config ls
```

После этого, когда ваш пакет будет готов к публикации, используйте:

```
npm publish
```

И все готово.

Если вам нужно опубликовать новую версию, убедитесь, что вы обновили версию вашего пакета. В противном случае NPM не позволит вам опубликовать пакет.

```
{
    "name": "package-name",
    "version": "1.0.4"
}
```

Запуск скриптов

Вы можете определить скрипты в вашем `package.json`, например:

```
{
    "name": "your-package",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "author": "",
    "license": "ISC",
    "dependencies": {},
    "devDependencies": {},
    "scripts": {
        "echo": "echo hello!"
    }
}
```

Чтобы запустить скрипт `echo`, выполните команду `npm run echo` из командной строки. Произвольные скрипты, такие как `echo` выше, должны запускаться с помощью команды `npm run <script name>`. `npm` также имеет ряд официальных скриптов, которые он запускает на определенных этапах жизненного цикла пакета (например, `preinstall`). Полный обзор того, как `NPM` обрабатывает поля скриптов, можно найти в документации: (https://docs.npmjs.com/cli/v10/using-npm/scripts).

Скрипты NPM чаще всего используются для таких задач, как запуск сервера, сборка проекта и запуск тестов. Вот более реалистичный пример:

```
"scripts": {
    "test": "mocha tests",
    "start": "pm2 start index.js"
}
```

В записях скриптов командные программы, такие как mocha, будут работать при установке как глобально, так и локально. Если запись командной строки не существует в системном PATH, npm также проверит ваши локально установленные пакеты.

Если ваши скрипты становятся слишком длинными, их можно разделить на части, например:

```
"scripts": {
    "very-complex-command": "npm run chain-1 && npm run chain-2",
    "chain-1": "webpack",
    "chain-2": "node app.js"
}
```

Удаление лишних пакетов

Чтобы удалить лишние пакеты (пакеты, которые установлены, но не указаны в списке зависимостей), выполните следующую команду:

```
npm prune
```

Чтобы удалить все пакеты разработки, добавьте флаг `--production`:

```
npm prune -production
```

Список установленных пакетов

Чтобы сгенерировать список (в виде дерева) текущих установленных пакетов, используйте:

```
npm list
```

Команды `ls`, `la` и `ll` являются синонимами команды `list`. Команды `la` и `ll` отображают дополнительную информацию, такую как описание и репозиторий.

Опции

Формат ответа может быть изменен с помощью передачи опций.

```
npm list -json
```

- `json` — показывает информацию в формате JSON

- `long` — показывает расширенную информацию
- `parseable` — показывает парсируемый список вместо дерева
- `global` — показывает глобально установленные пакеты
- `depth` — максимальная глубина отображения дерева зависимостей
- `dev/development` — показывает devDependencies
- `prod/production` — показывает dependencies

При желании вы также можете перейти на домашнюю страницу пакета:

```
npm home <package name>
```

Обновление npm и пакетов

Так как NPM сам по себе является модулем Node.js, его можно обновить с помощью него самого.

Если операционная система — Windows, командная строка должна быть запущена от имени администратора:

```
npm install -g npm@latest
```

Если вы хотите проверить наличие обновленных версий, вы можете выполнить:

```
npm outdated
```

Для обновления конкретного пакета:

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

Это обновит пакет до последней версии в соответствии с ограничениями в `package.json`.

Если вы также хотите зафиксировать обновленную версию в `package.json`:

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

Закрепление модулей на конкретных версиях

По умолчанию `npm` устанавливает последнюю доступную версию модулей в соответствии с семантическим версионированием зависимостей. Это может стать проблемой, если автор модуля не придерживается правил семантического версионирования и, например, вводит разрушающие изменения в обновлении модуля.

Чтобы закрепить версию каждой зависимости (и версию их зависимостей и т. д.) за конкретной версией, установленной локально в папке node_modules, используйте:

```
npm shrinkwrap
```

Это создаст файл npm-shrinkwrap.json рядом с вашим package.json, в котором будут перечислены конкретные версии зависимостей.

Настройка для глобально установленных пакетов

Вы можете использовать npm install -g, чтобы установить пакет глобально. Обычно это делается для установки исполняемого файла, который вы можете добавить в свой путь для запуска. Например:

```
npm install -g gulp-cli
```

Если вы обновите свой путь, вы сможете вызывать gulp напрямую.

На многих операционных системах npm install -g попытается записать в каталог, в который ваш пользователь, возможно, не сможет записать, например, /usr/bin. В этом случае не следует использовать sudo npm install, так как существует потенциальный риск для безопасности при запуске произвольных скриптов с sudo, а пользователь root может создавать каталоги в вашем домашнем каталоге, в которые вы не сможете записывать, что усложнит будущие установки.

Вы можете указать NPM, куда устанавливать глобальные модули, через файл конфигурации ~/.npmrc. Это называется префиксом, который можно просмотреть с помощью npm prefix:

```
prefix=~/.npm-global-modules
```

Этот префикс будет использоваться всякий раз, когда вы запускаете npm install -g. Вы также можете использовать npm install --prefix ~/.npm-global-modules, чтобы установить префикс при установке. Если префикс совпадает с вашим конфигурационным файлом, вам не нужно использовать -g.

Чтобы глобально установленный модуль мог быть задействован, он должен находиться в вашем пути:

```
export PATH=$PATH:~/.npm-global-modules/bin
```

Теперь, когда вы запустите npm install -g gulp-cli, вы сможете использовать gulp.

Примечание: когда вы выполняете `npm install` (без `-g`), префикс будет каталогом с `package.json` или текущим каталогом, если он не найден в иерархии. Это также создаст каталог `node_modules/.bin`, в котором будут находиться исполняемые файлы. Если вы хотите использовать исполняемый файл, специфичный для проекта, нет необходимости использовать `npm install -g`. Вы можете использовать тот, что находится в `node_modules/.bin`.

Связывание проектов для более быстрой отладки и разработки

Сборка зависимостей проекта иногда может быть утомительным занятием. Вместо публикации версии пакета в NPM и установки зависимости для тестирования изменений используйте `npm link`. Он создает символическую ссылку, чтобы последний код можно было протестировать в локальной среде. Это облегчает тестирование глобальных инструментов и зависимостей проекта, позволяя запускать последний код перед созданием опубликованной версии.

Код помощи

```
NAME
    npm-link - Symlink a package folder

SYNOPSIS
    npm link (in package dir)
    npm link [/][@]

    alias: npm ln
```

Шаги для связывания зависимостей проекта

При создании ссылки на зависимость обратите внимание на то, что будет ссылаться на имя пакета в родительском проекте.

1. Перейдите в каталог зависимости (например, `cd ../my-dep`).
2. Выполните `npm link`.
3. Перейдите в проект, который будет использовать зависимость.
4. Выполните `npm link my-dep` или, если используется пространство имен, `npm link @namespace/my-dep`.

Шаги для связывания глобального инструмента

1. Перейдите в каталог проекта (например, `cd eslint-watch`).
2. Выполните `npm link`.
3. Используйте инструмент.
4. Выполните `esw --quiet`.

Проблемы, которые могут возникнуть

Связывание проектов иногда может вызвать проблемы, если зависимость или глобальный инструмент уже установлены. Использование `npm uninstall (-g) <pkg>` с последующим выполнением `npm link` обычно решает любые возникающие проблемы.

Глава 76.
Node Version Manager (nvm)

Примечания

Чтобы установить nvm с использованием последней версии, перейдите по указанной ссылке на GitHub (https://github.com/nvm-sh/nvm), где будут предоставлены актуальные url.

Примеры

Установка NVM

Вы можете использовать curl:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/
install.sh | bash
```

или wget:

```
wget  -qO-  https://raw.githubusercontent.com/creationix/nvm/v0.31.3/
install.sh | bash
```

Проверка версии NVM

Чтобы проверить, что nvm был установлен, выполните:

```
command -v nvm
```

Если установка прошла успешно, должно выводиться ‘nvm’.

Установка конкретной версии Node

Список доступных удаленных версий для установки

```
nvm ls-remote
```

Установка удаленной версии

```
nvm install <version>
```

Например:

```
nvm install 0.10.13
```

Использование уже установленной версии node

Чтобы просмотреть доступные локальные версии node через NVM:

```
nvm ls
```

Например, если `nvm ls` возвращает:

```
$ nvm ls

 v4.3.0
 v5.5.0
```

...вы можете переключиться на версию v5.5.0 с помощью:

```
nvm use v5.5.0
```

Установка nvm на Mac OSX

Процесс установки

Вы можете установить Node Version Manager, используя `git`, `curl` или `wget`. Запустите эти команды в терминале на Mac OSX.

Пример с curl:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/
install.sh | bash
```

Пример с wget:

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/
install.sh | bash
```

Проверка корректности установки NVM

Чтобы проверить правильность установки nvm, закройте и снова откройте терминал и введите `nvm`. Если появляется сообщение `nvm: command not found`,

возможно, ваша ОС не имеет необходимого файла .bash_profile. Введите в терминале touch ~/.bash_profile и запустите вышеуказанный скрипт установки снова.

Если вы все еще получаете сообщение nvm: command not found, попробуйте следующее:

- Введите в терминал nano .bashrc. Вы должны увидеть export-скрипт, почти идентичный следующему:

```
export NVM_DIR="/Users/johndoe/.nvm" [ -s "$NVM_DIR/nvm.sh" ] && .
"$NVM_DIR/nvm.sh"
```

- Скопируйте export-скрипт и удалите его из .bashrc.
- Сохраните и закройте файл .bashrc (CTRL+O — Enter — CTRL+X).
- Затем введите nano .bash_profile, чтобы открыть Bash Profile.
- Вставьте скопированный export-скрипт в Bash Profile на новой строке.
- Сохраните и закройте Bash Profile (CTRL+O — Enter — CTRL+X).
- Наконец, введите nano .bashrc, чтобы снова открыть файл .bashrc.
- Вставьте следующую строку в файл:

```
source ~/.nvm/nvm.sh
```

- Сохраните и закройте (CTRL+O — Enter — CTRL+X).
- Перезапустите терминал и введите nvm, чтобы проверить, работает ли он.

Установка псевдонима для версии node

Если вы хотите задать псевдоним (alias) для установленной версии node, выполните:

```
nvm alias <name> <version>
```

Для удаления псевдонима используйте:

```
nvm unalias <name>
```

Подходящий пример использования — если вы хотите задать какую-то версию, отличную от стабильной версии, как версию по умолчанию. Версии, для которых заданы псевдонимы по умолчанию, загружаются в консоли по умолчанию.

Пример:

```
nvm alias default 5.0.1
```

Тогда каждый раз при запуске консоли/терминала будет присутствовать версия 5.0.1 по умолчанию.

Примечание:

```
nvm alias # lists all aliases created on nvm
```

Запуск любой произвольной команды в подсистеме с нужной версией Node

Список всех установленных версий Node:

```
nvm ls

 v4.5.0
 v6.7.0
```

Запуск команды с использованием любой установленной версии Node:

```
nvm run 4.5.0 --version or nvm exec 4.5.0 node --version
Running node v4.5.0 (npm v2.15.9)
v4.5.0
nvm run 6.7.0 --version or nvm exec 6.7.0 node --version
Running node v6.7.0 (npm v3.10.3)
v6.7.0
```

Использование псевдонима

```
nvm run default --version or nvm exec default node --version
Running node v6.7.0 (npm v3.10.3)
v6.7.0
```

Установка версии Node LTS

```
nvm install –lts
```

Переключение версий

```
nvm use v4.5.0 or nvm use stable ( alias )
```

Глава 77.
OAuth 2.0

Примеры

OAuth 2 с реализацией Redis — grant_type: password

В этом примере мы будем использовать OAuth 2 в REST API с базой данных Redis.

Важно: вам нужно будет установить базу данных Redis на вашем компьютере. Загрузите ее отсюда для пользователей Linux: https://redis.io/downloads/.

...или отсюда для установки версии на Windows: https://github.com/ServiceStack/redis-windows.

Также мы будем использовать настольное приложение Redis Manager, установите его отсюда: https://redis.io/insight/.

Теперь нам нужно настроить наш сервер Node.js для использования базы данных Redis.

- Создание серверного файла: `app.js`

```
var express = require('express'),
    bodyParser = require('body-parser'),
      oauthserver = require('oauth2-server'); // Would be: 'oauth2-
server'

var app = express();
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

app.oauth = oauthserver({
    model: require('./routes/Oauth2/model'),
    grants: ['password', 'refresh_token'],
    debug: true
});

// Обработка запросов на выдачу токенов
app.all('/oauth/token', app.oauth.grant());

app.get('/secret', app.oauth.authorise(), function (req, res) {
    // Требуется действительный access_token
    res.send('Secret area');
});

app.get('/public', function (req, res) {
    // Не требует access_token
    res.send('Public area');
});
```

```
// Обработка ошибок
app.use(app.oauth.errorHandler());
app.listen(3000);
```

- Создание модели OAuth2 в routes/Oauth2/model.js

```
var model = module.exports,
    util = require('util'),
    redis = require('redis');

var db = redis.createClient();

var keys = {
    token: 'tokens:%s',
    client: 'clients:%s',
    refreshToken: 'refresh_tokens:%s',
    grantTypes: 'clients:%s:grant_types',
    user: 'users:%s'
};

model.getAccessToken = function (bearerToken, callback) {
     db.hgetall(util.format(keys.token, bearerToken), function (err,
token) {
        if (err) return callback(err);
        if (!token) return callback();
        callback(null, {
            accessToken: token.accessToken,
            clientId: token.clientId,
            expires: token.expires ? new Date(token.expires) : null,
            userId: token.userId
        });
    });
};

model.getClient = function (clientId, clientSecret, callback) {
      db.hgetall(util.format(keys.client, clientId), function (err,
client) {
        if (err) return callback(err);
         if (!client || client.clientSecret !== clientSecret) return
callback();
        callback(null, {
            clientId: client.clientId,
            clientSecret: client.clientSecret
        });
    });
};
```

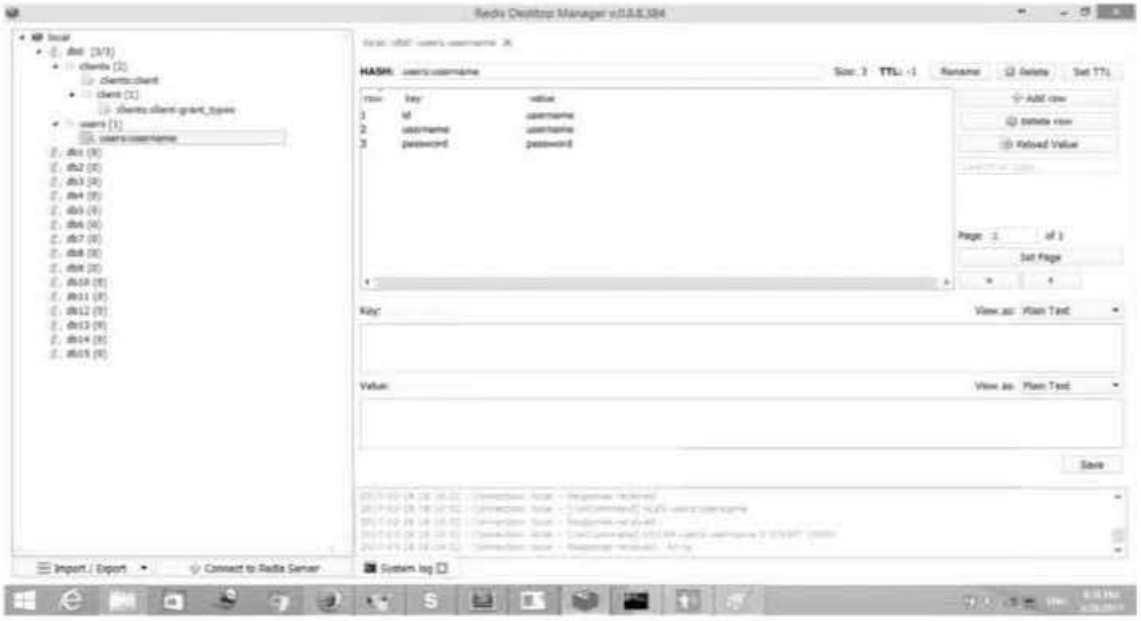
```
model.getRefreshToken = function (bearerToken, callback) {
     db.hgetall(util.format(keys.refreshToken, bearerToken), function
(err, token) {
        if (err) return callback(err);
        if (!token) return callback();
        callback(null, {
            refreshToken: token.accessToken,
            clientId: token.clientId,
            expires: token.expires ? new Date(token.expires) : null,
            userId: token.userId
        });
    });
};

model.grantTypeAllowed = function (clientId, grantType, callback) {
     db.sismember(util.format(keys.grantTypes, clientId), grantType,
callback);
};

model.saveAccessToken = function (accessToken, clientId, expires,
user, callback) {
    db.hmset(util.format(keys.token, accessToken), {
        accessToken: accessToken,
        clientId: clientId,
        expires: expires ? expires.toISOString() : null,
        userId: user.id
    }, callback);
};

model.saveRefreshToken = function (refreshToken, clientId, expires,
user, callback) {
    db.hmset(util.format(keys.refreshToken, refreshToken), {
        refreshToken: refreshToken,
        clientId: clientId,
        expires: expires ? expires.toISOString() : null,
        userId: user.id
    }, callback);
};
model.getUser = function (username, password, callback) {
   db.hgetall(util.format(keys.user, username), function (err, user) {
        if (err) return callback(err);
```

```
        if (!user || password !== user.password) return callback();
        callback(null, {
            id: username
        });
    });
};
```

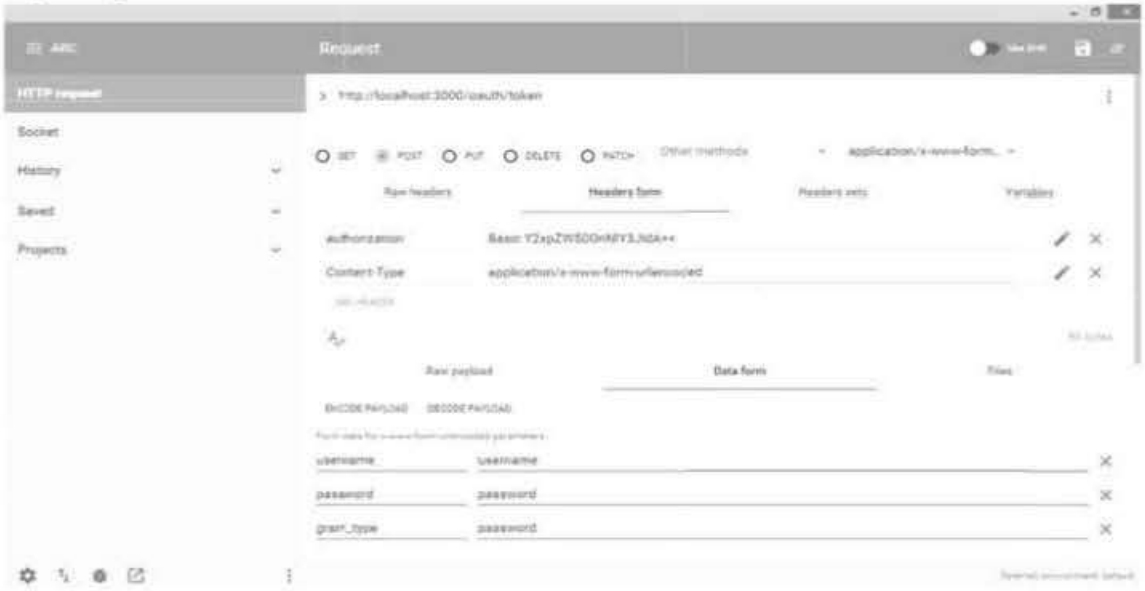
Вам нужно только установить Redis на вашем компьютере и запустить следующий файл Node.js:

```
#! /usr/bin/env node
var db = require('redis').createClient();
db.multi()

    .hmset('users:username', {
        id: 'username',
        username: 'username',
        password: 'password'
    })
    .hmset('clients:client', {
        clientId: 'client',
        clientSecret: 'secret'
    }) // clientId + clientSecret to base 64 will generate
Y2xpZW50OnNlY3JldA==
    .sadd('clients:client:grant_types', [
        'password',
        'refresh_token'
    ])

    .exec(function (errs) {
        if (errs) {
            console.error(errs[0].message);
            return process.exit(1);
        }
        console.log('Client and user added successfully');
        process.exit();
    });
```

Примечание: этот файл установит учетные данные для вашего фронтенда для запроса токена. Итак, ваш запрос из выборки базы данных Redis после вызова вышеуказанного файла будет следующим:



Пример вызова API



Заголовок:

1. `authorization:` Basic, за которым следует пароль, установленный при первой настройке Redis:

   a. `clientId + secretId в base64`

2. Форма данных:

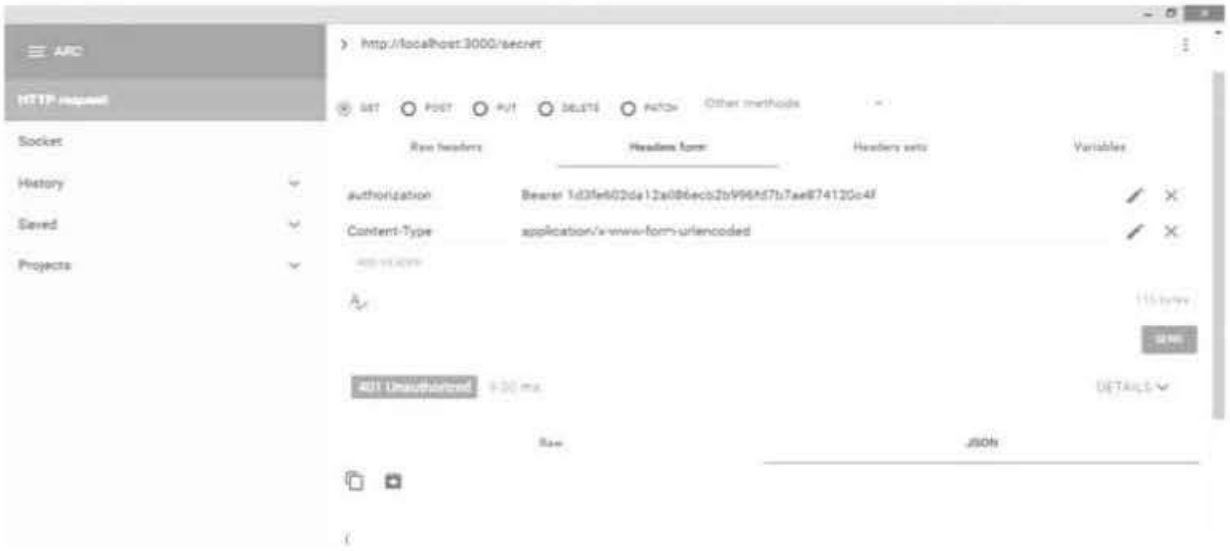
   `username`: пользователь, запрашивающий токен
   `password`: пароль пользователя
   `grant_type`: зависит от того, какие опции вы хотите выбрать, обычно используется password, который требует только username и password, чтобы быть созданными в Redis.

Данные в Redis будут такими:

```
{
    "access_token": "1d3fe602da12a086ecb2b996fd7b7ae874120c4f",
    "token_type": "bearer", // Будет использоваться для доступа к API
+ access+token, например: bearer 1d3fe602da12a086ecb2b996fd7b7ae87412
0c4f
    "expires_in": 3600,
    "refresh_token": "b6ad56e5c9aba63c85d7e21b1514680bbf711450"
}
```

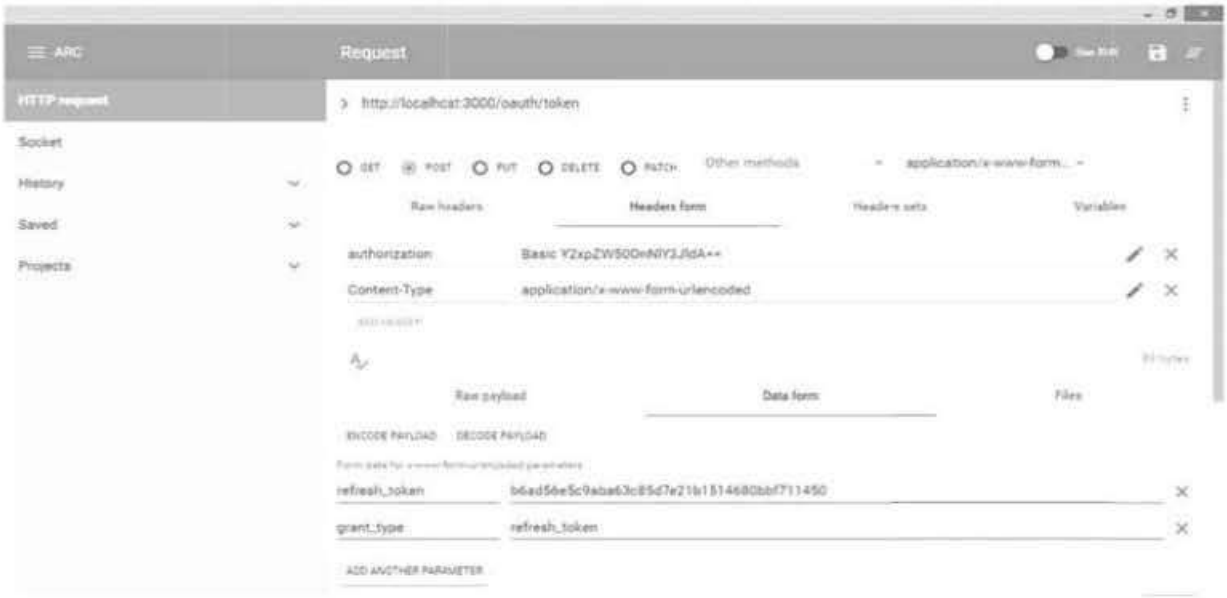
Теперь нам нужно вызвать наш API и получить некоторые защищенные данные с помощью токена доступа, который мы только что создали:

Когда токен истекает, API выдает ошибку о том, что токен истек, и вы не сможете получить доступ ни к одному из вызовов API:



Далее рассмотрим, какие действия следует предпринять, если токен истекает. Для начала поясним: если токен доступа истекает, в Redis есть возможность вызова `refresh_token`, который ссылается на истекший токен доступа. Поэтому нам нужно снова вызвать oauth/token с `refresh_token` в `grant_type` и установить авторизацию на Basic `clientId:clientsecret` (в base 64!), а затем отправить `refresh_token`. Это сгенерирует новый токен доступа с новой датой истечения срока действия.

На следующем изображении показано, как получить новый токен доступа:



Надеемся, что это поможет!

Глава 78. package.json

Замечания

Вы можете создать `package.json` с помощью команды

```
npm init
```

...которая затребует у вас базовую информацию о вашем проекте, включая идентификатор лицензии.

Примеры

Основное описание проекта

```
{
    "name": "my-project",
    "version": "0.0.1",
    "description": "This is a project.",
    "author": "Someone <someone@example.com>",
    "contributors": [{
        "name": "Someone Else",
        "email": "else@example.com"
    }],
    "keywords": ["improves", "searching"]
}
```

Поле	Описание
name	Обязательное поле для установки пакета. Должно быть в нижнем регистре, одним словом без пробелов (допускаются дефисы и подчеркивания).
version	Обязательное поле для версии пакета, использующее семантическое версионирование.

Поле	Описание
description	Краткое описание проекта.
author	Указывает автора пакета.
contributors	Массив объектов, по одному для каждого участника.
keywords	Массив строк, который поможет пользователям находить ваш пакет.

Зависимости

```
"dependencies": { "module-name": "0.1.0" }
```

- `exact`: 0.1.0 установит конкретную версию модуля.
- `newest minor version`: `^0.1.0` установит новейшую минорную версию, например `0.2.0`, но не установит модуль с более высокой мажорной версией, например `1.0.0`.
- `newest patch`: `0.1.x` или `~0.1.0` установит самую новую патч-версию, например `0.1.4`, но не установит модуль с более высокой мажорной или минорной версией, например `0.2.0` или `1.0.0`.
- `wildcard`: * установит последнюю версию модуля.
- `git repository`: установит tarball с master-ветки репозитория git. Можно также указать #sha, `#tag` или `#branch`:
  - GitHub: `user/project` или `user/project#v1.0.0`
  - url: `git://gitlab.com/user/project.git` или `git://gitlab.com/user/project.git#develop`
- локальный путь: file:../lib/project

После добавления их в ваш `package.json` используйте команду `npm install` в каталоге вашего проекта в терминале.

devDependencies

```
"devDependencies": {
    "module-name": "0.1.0"
}
```

Для зависимостей, необходимых только для разработки, таких как тестирование, стилизация, прокси и т.д. Эти dev-зависимости не будут установлены при выполнении команды `npm install` в режиме `production`.

Скрипты

Вы можете определить скрипты, которые могут быть выполнены или запускаются до или после другого скрипта:

```
{
    "scripts": {
        "pretest": "scripts/pretest.js",
        "test": "scripts/test.js",
        "posttest": "scripts/posttest.js"
    }
}
```

В этом случае вы можете выполнить скрипт, запустив любую из этих команд:

```
$ npm run-script test
$ npm run test
$ npm test
$ npm t
```

Предопределенные скрипты

Имя скрипта	Описание
prepublish	Запускается перед публикацией пакета.
publish, postpublish	Запускается после публикации пакета.
preinstall	Запускается перед установкой пакета.
install, postinstall	Запускается после установки пакета.
preuninstall, uninstall	Запускается перед удалением пакета.
postuninstall	Запускается после удаления пакета.
preversion, version	Запускается перед изменением версии пакета.
postversion	Запускается после изменения версии пакета.
pretest, test, posttest	Запускается командой npm test.
prestop, stop, poststop	Запускается командой npm stop.
prestart, start, poststart	Запускается командой npm start.
prerestart, restart, postrestart	Запускается командой npm restart.

Пользовательские скрипты

Вы также можете определить свои собственные скрипты так же, как и предопределенные скрипты:

```
{
    "scripts": {
        "preci": "scripts/preci.js",
        "ci": "scripts/ci.js",
        "postci": "scripts/postci.js"
    }
}
```

В данном случае вы можете выполнить скрипт, запустив одну из этих команд:

```
$ npm run-script ci
$ npm run ci
```

Пользовательские скрипты также поддерживают пред- и пост-скрипты, как показано в примере выше.

Расширенное определение проекта

Некоторые дополнительные атрибуты, такие как `repository`, `bugs` или `homepage`, анализируются веб-сайтом NPM и отображаются в инфобоксе для этих пакетов

```
{
    "main": "server.js",
    "repository": {
        "type": "git",
       "url": "git+https://github.com/<accountname>/<repositoryname>.
git"
    },
    "bugs": {
           "url": "https://github.com/<accountname>/<repositoryname>/
issues"
    },
    "homepage": "https://github.com/<accountname>/<repositoryname>#re
adme",
    "files": [
        "server.js", // исходные файлы
        "README.md", // дополнительные файлы
        "lib" // папка со всеми включенными файлами
    ]
}
```

Поле	Описание
main	Основной скрипт для этого пакета. Этот скрипт возвращается, когда пользователь требует пакет.
repository	Расположение и тип публичного репозитория.
bugs	Трекер ошибок для этого пакета (например, GitHub).
homepage	Домашняя страница для этого пакета или общего проекта.
files	Список файлов и папок, которые должны быть загружены, когда пользователь выполняет команду npm install <packagename>.

Изучение файла package.json

Файл package.json, обычно находящийся в корне проекта, содержит метаданные о вашем приложении или модуле, а также список зависимостей для установки из npm при выполнении npm install.

Чтобы инициализировать файл package.json, введите npm init в командной строке.

Чтобы создать package.json с настройками по умолчанию, используйте:

```
npm init --yes
```

или

```
npm init -y
```

Чтобы установить пакет и сохранить его в package.json, используйте:

```
npm install {package name} –save
```

Вы также можете использовать сокращенную запись:

```
npm i -S {package name}
```

NPM-алиасы -S для --save и -D для --save-dev сохраняют в производственных зависимостях или в зависимостях разработки соответственно. Пакет появится в ваших зависимостях; если вы используете --save-dev вместо --save, пакет появится в devDependencies.

Важные свойства package.json:

```
{
    "name": "module-name",
    "version": "10.3.1",
     "description": "An example module to illustrate the usage of a
package.json",
```

```
    "author": "Your Name <your.name@example.org>",
    "contributors": [{
        "name": "Foo Bar",
        "email": "foo.bar@example.com"
    }],
    "bin": {
        "module-name": "./bin/module-name"
    },
    "scripts": {
        "test": "vows --spec --isolate",
        "start": "node index.js",
        "predeploy": "echo About to deploy",
        "postdeploy": "echo Deployed",
         "prepublish": "coffee --bare --compile --output lib/foo src/
foo/*.coffee"
    },
    "main": "lib/foo.js",
    "repository": {
        "type": "git",
        "url": "https://github.com/username/repo"
    },
    "bugs": {
        "url": "https://github.com/username/issues"
    },
    "keywords": [
        "example"
    ],
    "dependencies": {
        "express": "4.2.x"
    },
    "devDependencies": {
        "assume": "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0"
    },
    "peerDependencies": {
        "moment": ">2.0.0"
    },
    "preferGlobal": true,
    "private": true,
    "publishConfig": {
        "registry": "https://your-private-hosted-npm.registry.domain.
com"
    },
    "subdomain": "foobar",
    "analyze": true,
    "license": "MIT",
```

```
    "files": [
        "lib/foo.js"
    ]
}
```

Информация о некоторых важных свойствах:

`name` — уникальное имя вашего пакета, которое должно быть написано строчными буквами. Это свойство обязательно, и без него ваш пакет не установится.

1. Имя должно содержать не более 214 символов.
2. Имя не может начинаться с точки или нижнего подчеркивания.
3. Новые пакеты не должны содержать заглавных букв в имени.

`version` — версия пакета указывается по семантическому версионированию (`semver`). Предполагается, что номер версии записывается как MAJOR.MINOR. PATCH, и вы увеличиваете:

1. MAJOR-версию при внесении несовместимых изменений в API.
2. MINOR-версию при добавлении функционала, совместимого с предыдущими версиями.
3. PATCH-версию при внесении обратно совместимых исправлений ошибок.

`description` — описание проекта. Старайтесь делать его кратким и точным.

`author` — автор этого пакета.

`bin` — объект, который используется для экспорта бинарных скриптов из вашего пакета. Предполагается, что ключ — это имя бинарного скрипта, а значение — относительный путь к скрипту. Это свойство используется пакетами, которые содержат интерфейс командной строки (CLI).

`scripts` — объект, который предоставляет дополнительные `npm`-команды. Предполагается, что ключ — это `npm`-команда, а значение — путь к скрипту. Эти скрипты могут быть выполнены при запуске `npm run {command name}` или `npm run-script {command name}`.

`main` — основная точка входа в ваш пакет. При вызове `require('{module name}')` в Node.js этот файл будет фактически загружен. Настоятельно рекомендуется, чтобы требование основного файла не вызывало побочных эффектов. Например, требование основного файла не должно запускать HTTP-сервер или подключаться к базе данных. Вместо этого вы должны создать что-то вроде `exports.init = function () {...}` в вашем основном скрипте.

`keywords` — массив ключевых слов, описывающих ваш пакет. Они помогут пользователям найти ваш пакет.

`devDependencies` — это зависимости, которые предназначены только для разработки и тестирования вашего модуля. Зависимости будут установлены авто-

матически, если только переменная окружения NODE_ENV=production не установлена. В этом случае вы все равно можете установить эти пакеты с помощью npm install --dev.

peerDependencies — если вы используете этот модуль, то peerDependencies перечисляет модули, которые вы должны установить вместе с этим. Например, moment-timezone должен быть установлен вместе с moment, потому что он является плагином для moment, даже если он не требует require("moment") напрямую.

preferGlobal — свойство, указывающее, что этот пакет предпочтительно устанавливать глобально с использованием npm install -g {module-name}. Это свойство используется пакетами, которые содержат интерфейс командной строки (CLI). В остальных случаях не следует использовать это свойство.

publishConfig — это объект с конфигурационными значениями, которые будут использоваться для публикации модулей. Установленные значения конфигурации переопределяют вашу конфигурацию npm по умолчанию. Наиболее распространенное использование publishConfig — публикация вашего пакета в частный npm-реестр, чтобы у вас по-прежнему были преимущества npm, но для частных пакетов. Это делается путем простого указания URL вашего частного npm в качестве значения для ключа registry.

files — это массив всех файлов, которые должны быть включены в публикуемый пакет. Может использоваться как путь к файлу, так и путь к папке. Все содержимое папки будет включено. Это уменьшает общий размер вашего пакета, включая только правильные файлы для распространения. Это поле работает совместно с файлом правил .npmignore.

Источник: https://browsenpm.org/.

Глава 79.
Разбор аргументов командной строки

Примеры

Передача действия (глагола) и значений

```
const options = require("commander");
options
    .option("-v, --verbose", "Be verbose");
options
    .command("convert")
    .alias("c")
    .description("Converts input file to output file")
    .option("-i, --in-file <file_name>", "Input file")
    .option("-o, --out-file <file_name>", "Output file")
    .action(doConvert);
options.parse(process.argv);
if (!options.args.length) options.help();

function doConvert(options) {
    // do something with options.inFile и options.outFile
}
```

Передача булевых переключателей

```
const options = require("commander");
options
    .option("-v, --verbose")
    .parse(process.argv);

if (options.verbose) {
    console.log("Let's make some noise!");
}
```

Глава 80.
Интеграция Passport

Примечания

Пароли всегда должны быть хешированы. Простым способом защиты паролей в Node.js будет использование модуля `bcrypt-nodejs`.

Примеры

Начало работы

Passport необходимо инициализировать с помощью промежуточного ПО `passport.initialize()`. Для использования сеансов входа требуется промежуточное ПО `passport.session()`.

Обратите внимание, что методы `passport.serialize()` и `passport.deserializeUser()` должны быть определены. Passport будет сериализовать и десериализовать экземпляры пользователя в и из сессии.

```
const express = require('express');
const session = require('express-session');
const passport = require('passport');
const cookieParser = require('cookie-parser');
const app = express();

// Требуется для чтения cookies
app.use(cookieParser());

passport.serializeUser(function(user, next) {
    // Сериализация пользователя в сессии
    next(null, user);
});

passport.deserializeUser(function(user, next) {
    // Использование ранее сериализованного пользователя
```

```
    next(null, user);
});

// Настройка промежуточного ПО express-session
app.use(session({
    secret: 'The cake is a lie',
    resave: true,
    saveUninitialized: true
}));

// Инициализация passport
app.use(passport.initialize());
app.use(passport.session());

// Запуск сервера express на порту 3000
app.listen(3000);
```

Локальная аутентификация

Модуль passport-local используется для реализации локальной аутентификации. Этот модуль позволяет вам аутентифицировать пользователей, используя имя пользователя и пароль в ваших приложениях на Node.js.

Регистрация пользователя:

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

// Используется именованная стратегия, так как используются две ло-
кальные стратегии:
// одна для регистрации, другая – для входа
passport.use('localSignup', new LocalStrategy({
    // Переопределение параметров по умолчанию,
    // которые – 'username' и 'password'
    usernameField: 'email',
    passwordField: 'password',
   passReqToCallback: true // позволяет нам передать весь запрос в об-
ратный вызов.
},

function(req, email, password, next) {
    // Проверка в базе данных, зарегистрирован ли пользователь
    findUserByEmail(email, function(user) {
        // Если email уже существует, прекращаем процесс регистрации и
        // передаем 'false' в обратный вызов
```

```
        if (user) return next(null, false);
        // Иначе создаем пользователя
        else {
            // Пароль должен быть хеширован!
            let newUser = createUser(email, password);
            newUser.save(function() {
                // Передаем пользователя в обратный вызов
                return next(null, newUser);
            });
        }
    });
}));
```

Вход пользователя:

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

passport.use('localSignin', new LocalStrategy({
    usernameField : 'email',
    passwordField : 'password',
},
function(email, password, next) {
    // Поиск пользователя
    findUserByEmail(email, function(user) {

        // Если пользователь не найден, прекращаем процесс входа
        // Пользовательские сообщения могут быть предоставлены в функ-
ции проверки,
        // чтобы дать пользователю больше деталей относительно неудач-
ной аутентификации
        if (!user)

            return next(null, false, {message: 'This e-mail address is
not associated with any account.'});
        // Иначе проверяем, правильный ли пароль
        else {

            // Если пароль неверный, прекращаем процесс входа
            if (!isPasswordValid(password)) return next(null, false);
            // Иначе передаем пользователя в обратный вызов
            else return next(null, user);
        }
    });
}));
```

Создание маршрутов

```
// ...
app.use(passport.initialize());
app.use(passport.session());

// Маршрут для входа
// Стратегии Passport являются middleware
app.post('/login', passport.authenticate('localSignin', {
    successRedirect: '/me',
    failureRedirect: '/login'
}));

// Маршрут для регистрации
app.post('/register', passport.authenticate('localSignup', {
    successRedirect: '/',
    failureRedirect: '/signup'
}));

// Вызов req.logout() для выхода
app.get('/logout', function(req, res) {
    req.logout();
    res.redirect('/');
});

app.listen(3000);
```

Аутентификация через Facebook

Модуль passport-facebook используется для реализации аутентификации через Facebook. В этом примере, если пользователь не существует при входе, он создается.

Реализация стратегии:

```
const passport = require('passport');
const FacebookStrategy = require('passport-facebook').Strategy;

// Стратегия по умолчанию называется 'facebook'
passport.use(new FacebookStrategy({
    clientID: 'yourclientid',
    clientSecret: 'yourclientsecret',
    callbackURL: '/auth/facebook/callback'
},
// Facebook отправит токен и профиль пользователя
```

```
function(token, refreshToken, profile, next) {
    // Проверяем в базе данных, зарегистрирован ли пользователь
    findUserByFacebookId(profile.id, function(user) {
      // Если пользователь существует, возвращаем его данные в callback
        if (user) return next(null, user);
        // В противном случае создаем пользователя
        else {
            let newUser = createUserFromFacebook(profile, token);

            newUser.save(function() {
                // Передаем пользователя в callback
                return next(null, newUser);
            });
        }
    });
}));
```

Создание маршрутов

```
// ...
app.use(passport.initialize());
app.use(passport.session());

// Маршрут для аутентификации
app.get('/auth/facebook', passport.authenticate('facebook', {
    // Запрос дополнительных разрешений у Facebook
    scope: 'email'
}));

// Маршрут, вызываемый после аутентификации пользователя через Facebook
app.get('/auth/facebook/callback',
    passport.authenticate('facebook', {
        successRedirect: '/me',
        failureRedirect: '/'
    })
);
//...
app.listen(3000);
```

Простая аутентификация по имени пользователя и паролю

В вашем файле `routes/index.js`:
Здесь `user` — это модель для userSchema.

```
router.post('/login', function(req, res, next) {
```

```
    if (!req.body.username || !req.body.password) {
        return res.status(400).json({
            message: 'Please fill out all fields'
        });
    }

    passport.authenticate('local', function(err, user, info) {
        if (err) {
            console.log("ERROR : " + err);
            return next(err);
        }

        if(user) {
            console.log("User Exists!");
                  // Все данные пользователя можно получить через
user.x res.json({"success" : true});
            return;

        } else {
            res.json({"success" : false});
            console.log("Error" + errorResponse());
            return;
        }

    })(req, res, next);

});
```

Аутентификация через Google с использованием Passport

Мы можем использовать простой модуль npm для аутентификации через Google, называемый `passport-google-oauth20`.

Рассмотрим пример, в котором примере создана папка `config`, в которой находятся файлы `passport.js` и `google.js` в корневом каталоге.

В вашем файле `app.js` включите следующий код:

```
var express = require('express');
var session = require('express-session');
var passport = require('./config/passport'); // путь к файлу passport
var app = express();
passport(app);
// остальной код для инициализации сервера и обработки ошибок
```

В файле `passport.js` в папке `config` включите следующий код:

```
var passport = require('passport'),
```

```
    google = require('./google'),
    User = require('./../model/user'); // User - это модель mongoose

module.exports = function(app){
    app.use(passport.initialize());
    app.use(passport.session());

    passport.serializeUser(function(user, done){
        done(null, user);
    });

    passport.deserializeUser(function (user, done) {
        done(null, user);
    });

    google();
};
```

В файле google.js в той же папке config включите следующий код:

```
var passport = require('passport'),
    GoogleStrategy = require('passport-google-oauth20').Strategy,
    User = require('./../model/user');

module.exports = function () {
    passport.use(new GoogleStrategy({
        clientID: 'CLIENT ID',
        clientSecret: 'CLIENT SECRET',
        callbackURL: "http://localhost:3000/auth/google/callback"
    },

    function(accessToken, refreshToken, profile, cb) {
       User.findOne({ googleId : profile.id }, function (err, user) {

            if(err){
                return cb(err, false, {message : err});

            }else {

                if (user != '' && user != null) {
                    return cb(null, user, {message : "User "});

                } else {
                    var username = profile.displayName.split(' ');
                    var userData = new User({
```

```
                        name : profile.displayName,
                        username : username[0],
                        password : username[0],
                        facebookId : '',
                        googleId : profile.id,
                    });

                    // Отправляем email пользователю на случай, если
нужно отправить
                    // вновь созданные учетные данные для последующе-
го входа без использования Google
                    userData.save(function (err, newuser) {

                        if (err) {
                            return cb(null, false, {message : err + "
!!! Please try again"});

                        } else {
                            return cb(null, newuser);
                        }
                    });
                }
            }
        });
    }
    ));
};
```

В этом примере, если пользователь отсутствует в базе данных, создается новый пользователь для локального использования, используя поле `googleId` в модели пользователя.

Глава 81. passport.js

Введение

Passport — популярный модуль авторизации для Node.js. Проще говоря, он обрабатывает все запросы на авторизацию пользователей в вашем приложении. Passport поддерживает более 300 стратегий, что позволяет легко интегрировать вход через Facebook, Google или любую другую социальную сеть. Стратегия, которую мы рассмотрим здесь, — это Local, где вы аутентифицируете пользователя, используя собственную базу данных зарегистрированных пользователей (с использованием имени пользователя и пароля).

Примеры

Пример LocalStrategy в passport.js

```
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;

passport.serializeUser(function(user, done) {
    // В serializeUser вы решаете, что хранить в сессии. Здесь мы сохраняем только id пользователя.
    done(null, user.id);
});

passport.deserializeUser(function(id, done) {
     // Здесь вы извлекаете всю информацию о пользователе из хранилища сессии, используя id пользователя, сохраненный ранее с помощью serializeUser.
    db.findById(id, function(err, user) {
        done(err, user);
    });
});
```

```
passport.use(new LocalStrategy(function(username, password, done) {
    db.findOne({'username':username},function(err,student){
          if(err) return done(err,{message:message}); // Неправильный
номер или пароль;
        var pass_retrieved = student.pass_word;
      bcrypt.compare(password, pass_retrieved, function(err3, correct)
{
            if(err3){
                message = [{"msg": "Incorrect Password!"}];
                 return done(null,false,{message:message}); // Непра-
вильный пароль
            }
            if(correct){
                return done(null,student);
            }
        });
    });
}));

app.use(session({ secret: 'super secret' }));
// Чтобы passport запомнил пользователя на других страницах. (Прочтите
о хранилище сессий. Мы использовали express-sessions.)
app.use(passport.initialize());
app.use(passport.session());

app.post('/',    passport.authenticate('local',    {successRedirect:'/
users', failureRedirect: '/'}),
    function(req,res,next){
});
```

Глава 82.
Проблемы производительности

Примеры

Обработка длительных запросов в Node.js

Так как Node.js однопоточный, требуется обходной путь для длительных вычислений.

Примечание: этот пример «готов к запуску». Не забудьте подключить jQuery и установить необходимые модули.

Основная логика этого примера:

1. Клиент отправляет запрос на сервер.
2. Сервер запускает процедуру в отдельном экземпляре Node.js и немедленно отправляет ответ с соответствующим ID задачи.
3. Клиент постоянно отправляет запросы на сервер для получения обновлений статуса данной задачи.

Структура проекта:

```
project
|   package.json
|   index.html
|
├──js
|   main.js
|   jquery-1.12.0.min.js
|
└──srv
    |   app.js
    ├──models
    |   task.js
    └──tasks
        data-processor.js
```

app.js:

```
var express = require('express');
var app = express();
var http = require('http').Server(app);
var mongoose = require('mongoose');
var bodyParser = require('body-parser');
var childProcess = require('child_process');
var Task = require('./models/task');

app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
app.use(express.static(__dirname + '/../'));

app.get('/', function(request, response){
    response.render('index.html');
});

// Маршрут для самого запроса
app.post('/long-running-request', function(request, response){
    // Создаем новый элемент задачи для отслеживания статуса
    var t = new Task({ status: 'Starting ...' });
    t.save(function(err, task){
       // Создаем новый экземпляр Node.js для выполнения отдельной за-
дачи в другом потоке
       taskProcessor = childProcess.fork('./srv/tasks/data-processor.js');

        // Обрабатываем сообщения, приходящие от процессора задач
        taskProcessor.on('message', function(msg){
            task.status = msg.status;
            task.save();
        }.bind(this));

         // Закрываем ранее открытый экземпляр Node.js после заверше-
ния задачи
        taskProcessor.on('close', function(msg){
            this.kill();
        });

        // Отправляем некоторые параметры для нашей отдельной задачи
        var params = {
            message: 'Hello from main thread'
        };
        taskProcessor.send(params);
```

```
        response.status(200).json(task);
    });
});

// Маршрут для проверки завершения вычислений по запросу
app.post('/is-ready', function(request, response){
    Task
        .findById(request.body.id)
        .exec(function(err, task){
            response.status(200).json(task);
        });
});

mongoose.connect('mongodb://localhost/test');
http.listen('1234');
```

task.js:

```
var mongoose = require('mongoose');

var taskSchema = mongoose.Schema({
    status: {
        type: String
    }
});

mongoose.model('Task', taskSchema);
module.exports = mongoose.model('Task');
```

data-processor.js:

```
process.on('message', function(msg){
    init = function(){
        processData(msg.message);
    }.bind(this)();

    function processData(message){
        // Отправляем обновление статуса в основное приложение
        process.send({ status: 'We have started processing your data.'
});
        // Долгие вычисления...
        setTimeout(function(){
            process.send({ status: 'Done!' });
```

```
            // Уведомляем Node.js, что задача завершена
            process.disconnect();
        }, 5000);
    }
});
process.on('uncaughtException', function(err){
    console.log("Error happened: " + err.message + "\n" + err.stack +
".\n");
    console.log("Gracefully finish the routine.");
});
```

index.html:

```
<!DOCTYPE html>
<html>
<head>
    <script src="./js/jquery-1.12.0.min.js"></script>
    <script src="./js/main.js"></script>
</head>
<body>
    <p>Example of processing long-running node requests.</p>
    <button id="go" type="button">Run</button>
    <br />
    <p>Log:</p>
    <textarea id="log" rows="20" cols="50"></textarea>
</body>
</html>
```

main.js:

```
$(document).on('ready', function(){
    $('#go').on('click', function(e){
        // Очистка логов
        $("#log").val('');
        $.post("/long-running-request", {some_params: 'params' })
            .done(function(task){
                $("#log").val( $("#log").val() + '\n' + task.status);

                // Функция для отслеживания статуса задачи
                function updateStatus(){
                    $.post("/is-ready", {id: task._id })
```

```
                        .done(function(response){
                            $("#log").val( $("#log").val() + '\n' +
response.status);
                            if(response.status != 'Done!'){
                                checkTaskTimeout =
setTimeout(updateStatus, 500);
                            }
                        });
                }

                // Начало проверки задачи
                var checkTaskTimeout = setTimeout(updateStatus, 100);
            });
    });
});
```

package.json:

```
{
    "name": "nodeProcessor",
    "dependencies": {
        "body-parser": "^1.15.2",
        "express": "^4.14.0",
        "html": "0.0.10",
        "mongoose": "^4.5.5"
    }
}
```

Обратите внимание: этот пример предназначен для объяснения основной идеи. Для использования в производственной среде он требует доработок.

Глава 83.
Интеграция с PostgreSQL

Примеры

Подключение к PostgreSQL

Рассмотрим использование NPM-модуля PostgreSQL.

Установите зависимость:

```
npm install pg –save
```

Теперь нужно создать подключение к PostgreSQL, которое вы сможете использовать для выполнения запросов.

Предположим, что у вас Database_Name = students, Host = localhost и DB_User = postgres.

```
var pg = require("pg");
var connectionString = "pg://postgres:postgres@localhost:5432/
students";
var client = new pg.Client(connectionString);
client.connect();
```

Запрос с использованием объекта подключения

Если вы хотите использовать объект подключения для запроса к базе данных, вы можете использовать этот пример кода:

```
var queryString = "SELECT name, age FROM students";
var query = client.query(queryString);
query.on("row", (row, result) => {
    result.addRow(row);
});
query.on("end", function (result) {
    // ЛОГИКА
});
```

Глава 84.
Структура проекта

Введение

Структура проекта на Node.js зависит от личных предпочтений, архитектуры проекта и используемой стратегии инъекции модулей. Также она влияет на событийно-ориентированную архитектуру, которая использует механизм динамического инстанцирования модулей. Чтобы создать структуру MVC, необходимо разделить серверный и клиентский код, так как клиентский код, вероятно, будет минимизирован и отправлен в браузер и является публичным по своей природе. Серверная часть или бэкенд будет предоставлять API для выполнения операций CRUD.

Примечания

Проект, описанный выше, использует модули browserify и `vue.js` в качестве основы для представления и библиотек минимизации. Поэтому структура проекта может незначительно измениться в зависимости от того, какой MVC-фреймворк вы используете. Например, директория `build` в папке `public` должна содержать весь код MVC. Вы можете настроить задачу, которая будет это делать за вас.

Примеры

Простое Node.js-приложение с использованием MVC и API

- Основное различие заключается в разделении динамически генерируемых директорий, которые будут использоваться для хостинга, и исходных директорий.
- Исходные директории будут содержать файл или папку конфигурации в зависимости от объема конфигурации, который может включать конфигурацию окружения и конфигурацию бизнес-логики, которую можно поместить в директорию `config`.

```
|-- Config
    |-- config.json
    |-- appConfig
        |-- pets.config
        |-- payment.config
```

- Теперь рассмотрим важнейшие директории, в которых различаются серверная (бэкенд) и клиентская части проекта. Две директории `server` и `webapp` представляют собой соответственно бэкенд и фронтенд, которые можно разместить в исходной директории, например `src`.

  Вы можете использовать другие названия в зависимости от личных предпочтений для `server` или `webapp`, в зависимости от того, что вам кажется более логичным. Однако не следует выбирать слишком длинные или сложные названия, так как это внутренняя структура проекта.

- Внутри директории `server` вы можете разместить контроллеры, а также файл `App.js/index.js`, который будет основным файлом Node.js и точкой входа. Директория `server` также может содержать директорию `dto`, в которой будут находиться все объекты передачи данных, используемые контроллерами API:

```
|-- server
    |-- dto
        |-- pet.js
        |-- payment.js
    |-- controller
        |-- PetsController.js
        |-- PaymentController.js
    |-- App.js
```

Директория `webapp` может быть разделена на две основные части: `public` и `mvc`. Это снова зависит от выбранной стратегии сборки. Мы используем browserify для сборки части MVC в веб-приложении и минимизации содержимого из директории `mvc`:

```
|-- webapp
    |-- public
        |-- build // будет содержать минимизированные скрипты (mvc)
        |-- images
            |-- mouse.jpg
            |-- cat.jpg
        |-- styles
            |-- style.css
        |-- views
```

```
        |-- petStore.html
        |-- paymentGateway.html
        |-- header.html
        |-- footer.html
        |-- index.html
```

- Директория mvc будет содержать фронтенд-логику, включая модели, контроллеры представлений и любые другие утилитарные модули, которые могут понадобиться как часть интерфейса. Также файл index.js или shell.js, в зависимости от предпочтений, будет частью этой директории:

```
|-- mvc
    |-- controllers
        |-- Dashborad.js
        |-- Help.js
        |-- Login.js
    |-- utils
        |-- index.js
```

Итак, в конце вся структура проекта будет выглядеть следующим образом. Простая задача сборки, например, с использованием gulp browserify, будет минимизировать скрипты MVC и публиковать их в директории public. Затем мы можем предоставить эту директорию как статический ресурс через express.use(static('public')) API:

```
|-- node_modules
|-- src
    |-- server
        |-- controller
        |-- App.js // node app
    |-- webapp
        |-- public
            |-- styles
            |-- images
            |-- index.html
        |-- mvc
            |-- controller
                |-- shell.js // mvc shell
|-- config
|-- Readme.md
|-- .gitignore
|-- package.json
```

Глава 85.
Push-уведомления

Введение

Если вы хотите создать уведомления для веб-приложения, мы рекомендуем использовать фреймворк Push.js или OneSignal для веб/мобильных приложений.

Push.js — это самый быстрый способ начать работать с уведомлениями на JavaScript. Довольно новое дополнение к официальной спецификации, API уведомлений позволяет современным браузерам, таким как Chrome, Safari, Firefox и IE 9+, отправлять уведомления на рабочий стол пользователя.

Вам необходимо будет использовать Socket.io и какой-либо backend-фреймворк. В этом примере мы задействуем Express.

Параметры

Модуль/Фреймворк	Описание
node.js/express	Простой backend-фреймворк для приложений на Node.js, очень простой в использовании и чрезвычайно мощный.
Socket.io	Socket.io обеспечивает двунаправленную событийную коммуникацию в реальном времени. Работает на всех платформах, браузерах и устройствах, уделяя равное внимание надежности и скорости.
Push.js	Самый универсальный фреймворк для уведомлений на рабочий стол.
OneSignal	Еще одна форма push-уведомлений для устройств Apple.
Firebase	Мобильная платформа Google, которая помогает быстро разрабатывать высококачественные приложения.

Примеры

Веб-уведомление

Сначала вам нужно установить модуль Push.js:

```
$ npm install push.js –save
```

...или импортируйте его в ваше фронтенд-приложение через CDN:

```
<script src="./push.min.js"></script> <!-- CDN ссылка -->
```

После этого вы можете начать работать. Вот как это может выглядеть, если вы хотите сделать простое уведомление:

```
Push.create('Hello World!')
```

Мы предполагаем, что вы знаете, как настроить Socket.io в вашем приложении. Вот пример кода для нашего backend-приложения на Express:

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server);
server.listen(80);

app.get('/', function (req, res) {
    res.sendfile(__dirname + '/index.html');
});

io.on('connection', function (socket) {
    socket.emit('pushNotification', { success: true, msg: 'hello' });
});
```

После того как ваш сервер настроен, можно переходить к фронтенд-части. Теперь все, что нужно сделать — это импортировать Socket.io CDN и добавить этот код в файл `index.html`:

```
<script src="../socket.io.js"></script> <!-- CDN ссылка -->
<script>
    var socket = io.connect('http://localhost');
    socket.on('pushNotification', function (data) {
        console.log(data);
        Push.create("Hello world!", {
            body: data.msg, // это должно вывести "hello"
            icon: '/icon.png',
            timeout: 4000,
            onClick: function () {
                window.focus();
```

```
                this.close();
            }
        });
    });
</script>
```

Теперь вы должны отобразить ваше уведомление. Это также работает на любом устройстве с Android. Если вы хотите использовать Firebase Cloud Messaging, вы можете использовать его с этим модулем. Вот ссылка на пример, написанный Ником (создателем Push.js): https://github.com/Nickersoft/push-fcm-plugin.

Apple

Имейте в виду, что это не будет работать на устройствах Apple, но если вы хотите создать push-уведомления, проверьте плагин OneSignal. Ссылка на документацию OneSignal: https://onesignal.com/.

Глава 86.
Readline

Синтаксис

- `const readline = require('readline')`
- `readline.close()`
- `readline.pause()`
- `readline.prompt([preserveCursor])`
- `readline.question(query, callback)`
- `readline.resume()`
- `readline.setPrompt(prompt)`
- `readline.write(data[, key])`
- `readline.clearLine(stream, dir)`
- `readline.clearScreenDown(stream)`
- `readline.createInterface(options)`
- `readline.cursorTo(stream, x, y)`
- `readline.emitKeypressEvents(stream[, interface])`
- `readline.moveCursor(stream, dx, dy)`

Примеры

Построчное чтение файла

```
const fs = require('fs');
const readline = require('readline');

const rl = readline.createInterface({
    input: fs.createReadStream('text.txt')
});
```

```
// Каждая новая строка вызывает событие – каждый раз, когда поток по-
лучает \r, \n или \r\n
rl.on('line', (line) => {
    console.log(line);
});

rl.on('close', () => {
    console.log('Done reading file'); // Файл прочитан
});
```

Запрос ввода от пользователя через CLI

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

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

rl.question('What is your name?', (name) => {
    console.log(`Hello ${name}!`);

    rl.close();
});
```

Глава 87.
Удаленная отладка в Node.js

Примеры

Конфигурация запуска Node.js

Чтобы настроить удаленную отладку Node.js, просто запустите процесс node с флагом --debug. Вы можете указать порт, на котором должен работать отладчик, используя --debug=<port>.

Когда ваш процесс Node.js запустится, вы увидите сообщение:

```
Debugger listening on port <port>
```

Это означает, что все настроено правильно.

Затем настройте цель удаленной отладки в вашем конкретном IDE.

Конфигурация IntelliJ/WebStorm

1. Убедитесь, что плагин Node.js включен.
2. Выберите конфигурации запуска (скриншот).

3. Выберите + > Node.js Remote Debug.

4. Убедитесь, что вы указали выбранный выше порт, а также правильный хост.

Когда все будет настроено, запустите цель отладки, как обычно, и она остановится на ваших точках останова.

Использование прокси для отладки через порт в Linux

Если вы запускаете ваше приложение на Linux, используйте прокси для отладки через порт, например:

```
socat TCP-LISTEN:9958,fork TCP:127.0.0.1:5858 &
```

Используйте порт 9958 для удаленной отладки.

Глава 88.
Require()

Введение

В данной главе рассматривается использование оператора `require()`, который включен в Node.js.

`Require` представляет собой импорт определенных файлов или пакетов, используемых с модулями Node.js. Он применяется для улучшения структуры кода и его использования. `require()` применяется к файлам, которые установлены локально, с прямым маршрутом от файла, который использует `require()`.

Синтаксис

- `module.exports = {testFunction: testFunction};`
- `var test_file = require('./testFile.js');` // Предположим, у нас есть файл с именем `testFile`
- `test_file.testFunction(our_data);` // Пусть в `testFile` есть функция `testFunction`

Примечания

Использование `require()` позволяет структурировать код аналогично использованию классов и публичных методов в Java. Если функция экспортируется с помощью `.exports`, ее можно использовать в другом файле через `require()`. Если файл не экспортируется, его нельзя использовать в другом файле.

Примеры

Начало использования require() с функцией и файлом

`Require` — это оператор, который Node интерпретирует в некотором смысле как функцию получения (getter). Например, предположим, что у вас есть файл с именем `analysis.js`, и внутри этого файла содержится следующий код:

```
function analyzeWeather(weather_data) {
    console.log('Weather information for ' + weather_data.time + ': ');
    console.log('Rainfall: ' + weather_data.precip);
    console.log('Temperature: ' + weather_data.temp);
    // Дополнительный анализ/вывод weather_data...
}
```

Этот файл содержит только метод `analyzeWeather(weather_data)`. Если мы хотим использовать данную функцию, ее необходимо либо использовать внутри этого файла, либо скопировать в файл, в котором она должна быть использована. Однако Node.js предоставляет очень полезный инструмент для организации кода и файлов — модули.

Чтобы использовать нашу функцию, мы должны сначала экспортировать ее с помощью оператора в начале файла. Наш новый файл выглядит так:

```
module.exports = {
    analyzeWeather: analyzeWeather
}
function analyzeWeather(weather_data) {
    console.log('Weather information for ' + weather_data.time + ': ');
    console.log('Rainfall: ' + weather_data.precip);
    console.log('Temperature: ' + weather_data.temp);
    // Дополнительный анализ/вывод weather_data...
}
```

С этим небольшим оператором `module.exports` наша функция теперь готова к использованию за пределами файла. Осталось только использовать `require()`.

Когда используется `require()` для функции или файла, синтаксис очень схож. Обычно это делается в начале файла и присваивается переменным или константам для использования в течение всего файла. Например, у нас есть другой файл (на том же уровне, что и `analyze.js`) с именем `handleWeather.js`, который выглядит так:

```
const analysis = require('./analysis.js');
weather_data = {
    time: '01/01/2001',
    precip: 0.75,
    temp: 78,
    // Дополнительные данные о погоде...
};
analysis.analyzeWeather(weather_data);
```

В этом файле мы используем `require()` для захвата нашего файла `analysis.js`. При использовании мы просто вызываем переменную или константу, присвоенную этому `require`, и используем любую функцию внутри, которая была экспортирована.

Начало использования require() с NPM-пакетом

Require также очень полезен при использовании вместе с NPM-пакетом. Предположим, например, что вы хотите использовать NPM-пакет в файле с именем getWeather.js. После установки вашего пакета через командную строку (npm install request) вы сможете его использовать. Ваш файл getWeather.js может выглядеть так:

```
var https = require('request');
// Сконструируйте переменную url...
https.get(url, function(error, response, body) {
    if (error) {
        console.log(error);
    } else {
        console.log('Response => ' + response);
        console.log('Body => ' + body);
    }
});
```

Когда этот файл выполняется, сначала используется require() (импортируется) пакет, который вы только что установили, под названием request. Внутри файла request есть множество функций, к которым теперь у вас есть доступ, одна из которых называется get. В следующих нескольких строках эта функция используется для выполнения HTTP GET-запроса.

Глава 89.
RESTful API: лучшие практики

Примеры

Обработка ошибок: GET всех ресурсов

Как обрабатывать ошибки, а не просто логировать их в консоль?

Плохой способ:

```
Router.route('/')
    .get((req, res) => {
        Request.find((err, r) => {
            if(err){
                console.log(err)
            } else {
                res.json(r)
            }
        })
    })

    .post((req, res) => {
        const request = new Request({
            type: req.body.type,
            info: req.body.info
        });
        request.info.user = req.user._id;
        console.log("ABOUT TO SAVE REQUEST", request);
        request.save((err, r) => {
            if (err) {
                res.json({ message: 'there was an error saving your r'});

            } else {
                res.json(r);
```

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

Лучший способ:

```
Router.route('/')
    .get((req, res, next) => {
        Request.find((err, r) => {
            if(err){
                console.log(err)
            } else {
                return next(err)
            }
        })
    })
    .post((req, res, next) => {
        const request = new Request({
            type: req.body.type,
            info: req.body.info
        });
        request.info.user = req.user._id;
        console.log("ABOUT TO SAVE REQUEST", request);
        request.save((err, r) => {
            if (err) {
                return next(err)
            } else {
                res.json(r);
            }
        });
    });
```

Глава 90.
Структура Route-Controller-Service для Express.js

Примеры

Структура директорий
Models Routes-Controllers-Services

```
├──models
|  ├──user.model.js
├──routes
|  ├──user.route.js
├──services
|  ├──user.service.js
├──controllers
|  ├──user.controller.js
```

Для модульной структуры кода логика должна быть разделена на следующие директории и файлы.

- `Models` — определение схемы модели.
- `Routes` — API-маршруты, которые «мапятся» на контроллеры.
- `Controllers` — контроллеры, которые обрабатывают всю логику, связанную с валидацией параметров запроса, отправкой ответов с правильными кодами.
- `Services` — сервисы, которые содержат запросы к базе данных, возврат объектов или выброс ошибок.

Такой подход требует написания большего объема кода. Но в конечном итоге код будет гораздо более поддерживаемым и разделенным.

Пример структуры кода Model-Routes-Controllers-Services

user.model.js

```
var mongoose = require('mongoose')
const UserSchema = new mongoose.Schema({
    name: String
})
const User = mongoose.model('User', UserSchema)
module.exports = User;
```

user.routes.js

```
var express = require('express');
var router = express.Router();
var UserController = require('../controllers/user.controller')

router.get('/', UserController.getUsers)

module.exports = router;
```

user.controllers.js

```
var UserService = require('../services/user.service')

exports.getUsers = async function (req, res, next) {
    // Validate request parameters, queries using express-validator
    var page = req.params.page ? req.params.page : 1;
    var limit = req.params.limit ? req.params.limit : 10;

    try {
        var users = await UserService.getUsers({}, page, limit)
      return res.status(200).json({ status: 200, data: users, message:
"Succesfully Users Retrieved" });
    } catch (e) {
        return res.status(400).json({ status: 400, message: e.message
});
    }
}
```

user.services.js

```
var User = require('../models/user.model')
exports.getUsers = async function (query, page, limit) {
```

```
    try {
        var users = await User.find(query)
        return users;
    } catch (e) {
        // Log Errors
        throw Error('Error while Paginating Users')
    }
}
```

Глава 91. Маршрутизация ajax-запросов с помощью Express.js

Примеры

Простая реализация AJAX

Для начала вам нужно использовать базовый шаблон express-generator.

В файле app.js добавьте (можно добавить в любом месте после var app = express.app()):

```
app.post(function(req, res, next){
    next();
});
```

Теперь в вашем файле index.js (или его соответствии) добавьте:

```
router.get('/ajax', function(req, res){
     res.render('ajax', {title: 'An Ajax Example', quote: "AJAX is
great!"});
});
router.post('/ajax', function(req, res){
     res.render('ajax', {title: 'An Ajax Example', quote: req.body.
quote});
});
```

Создайте файл ajax.jade / ajax.pug или ajax.ejs в каталоге /views, добавьте:

Для Jade/PugJS:

```
extends layout
script(src="http://code.jquery.com/jquery-3.1.0.min.js")
script(src="/magic.js")
h1 Quote: !{quote}
form(method="post" id="changeQuote")
```

```
   input(type='text', placeholder='Set quote of the day', name='quote')
    input(type="submit", value="Save")
```

Для EJS:

```
<script src="http://code.jquery.com/jquery-3.1.0.min.js"></script>
<script src="/magic.js"></script>
<h1>Quote: <%=quote%> </h1>
<form method="post" id="changeQuote">
   <input type="text" placeholder="Set quote of the day" name="quote"/>
    <input type="submit" value="Save">
</form>
```

Теперь создайте файл в каталоге /public с названием magic.js:

```
$(document).ready(function(){
    $("form#changeQuote").on('submit', function(e){
        e.preventDefault();
        var data = $('input[name=quote]').val();
        $.ajax({
            type: 'post',
            url: '/ajax',
            data: data,
            dataType: 'text'
        })
        .done(function(data){
            $('h1').html(data.quote);
        });
    });
});
```

Когда вы нажмете «Save», цитата изменится!

Глава 92.
Запуск Node.js как сервиса

Введение

В отличие от многих веб-серверов, Node не установлен как служба по умолчанию. Но в рабочей среде лучше запускать его как daemon, управляемый системой инициализации.

Примеры

Node.js как daemon systemd

`systemd` является стандартной системой инициализации в большинстве дистрибутивов Linux. После настройки Node для работы с `systemd` можно использовать команду `service` для управления им.

Прежде всего необходим конфигурационный файл, давайте создадим его. Для дистрибутивов на базе Debian он будет находиться в

```
/etc/systemd/system/node.service

[Unit]
Description=My super nodejs app

[Service]
# установить рабочий каталог для согласованности относительных путей
WorkingDirectory=/var/www/app
# запуск файла сервера (файл относительно WorkingDirectory)
ExecStart=/usr/bin/node serverCluster.js
# если процесс аварийно завершится, всегда пытаться перезапустить
Restart=always
# задержка в 500 мс между аварийным завершением и перезапуском
RestartSec=500ms
# отправка лога в syslog (не конфликтует с другими настройками логов
в самом приложении)
```

```
StandardOutput=syslog
StandardError=syslog
# имя процесса nodejs в syslog
SyslogIdentifier=nodejs
# пользователь и группа, запускающие приложение
User=www-data
Group=www-data
# установка окружения (dev, prod…)
Environment=NODE_ENV=production

[Install]
# запуск node на уровне мультипользовательской системы (= sysVinit
runlevel 3)
WantedBy=multi-user.target
```

Теперь можно, соответственно, запускать, останавливать и перезапускать приложение с помощью:

```
service node start
service node stop
service node restart
```

Чтобы указать systemd автоматически запускать Node при загрузке, просто введите: `systemctl enable node`.

Теперь Node запускается как daemon.

Глава 93.
Обеспечение безопасности приложений Node.js

Примеры

Предотвращение межсайтовой подделки запроса (CSRF)

CSRF — это атака, при которой конечный пользователь вынужден выполнять нежелательные действия в веб-приложении, в котором он в данный момент аутентифицирован.

Это может произойти из-за того, что cookie отправляются с каждым запросом к веб-сайту, даже если эти запросы поступают с другого сайта.

Мы можем использовать модуль `csurf` для создания CSRF-токена и его валидации.

Пример

```
var express = require('express')
var cookieParser = require('cookie-parser') // для разбора cookie
var csrf = require('csurf') // модуль csrf
var bodyParser = require('body-parser') // для разбора тела запроса

// настройка промежуточных маршрутов
var csrfProtection = csrf({ cookie: true })
var parseForm = bodyParser.urlencoded({ extended: false })

// создание приложения express
var app = express()

// разбор cookie
app.use(cookieParser())

app.get('/form', csrfProtection, function(req, res) {
    // генерация и передача csrfToken в представление
```

```
    res.render('send', { csrfToken: req.csrfToken() })
})
app.post('/process', parseForm, csrfProtection, function(req, res) {
    res.send('data is being processed')
})
```

Таким образом, когда мы обращаемся к `GET /form`, это передаст CSRF-токен `csrfToken` в представление.

Теперь, внутри представления установите значение `csrfToken` как значение скрытого поля ввода с именем `_csrf`.

Пример для шаблонов Handlebars:

```
<form action="/process" method="POST">
    <input type="hidden" name="_csrf" value="{{csrfToken}}">
    Name: <input type="text" name="name">
    <button type="submit">Submit</button>
</form>
```

Пример для шаблонов Jade:

```
form(action="/process" method="post")
    input(type="hidden", name="_csrf", value=csrfToken)
    span Name:
    input(type="text", name="name", required=true)
    br
    input(type="submit")
```

Пример для шаблонов EJS:

```
<form action="/process" method="POST">
    <input type="hidden" name="_csrf" value="<%= csrfToken %>">
    Name: <input type="text" name="name">
    <button type="submit">Submit</button>
</form>
```

SSL/TLS в Node.js

Если вы хотите иметь возможность управлять SSL/TLS в вашем приложении на Node.js, примите во внимание, что вы также несете ответственность за предотвращение атак SSL/TLS на этом этапе. В многих архитектурах «клиент — сервер» SSL/TLS завершается на обратном прокси-сервере, чтобы уменьшить сложность приложения и снизить объем настроек безопасности.

Если ваше приложение на Node.js должно обрабатывать SSL/TLS, его можно обезопасить, загрузив файлы ключей и сертификатов.

Если ваш провайдер сертификатов требует цепочку сертификатов удостоверяющего центра (CA), она может быть добавлена в параметр CA как массив. Цепочка с несколькими записями в одном файле должна быть разделена на несколько файлов и введена в том же порядке в массиве, поскольку Node.js в настоящее время не поддерживает несколько записей CA в одном файле. Пример приведен в коде ниже для файлов 1_ca.crt и 2_ca.crt. Если массив CA требуется и не настроен правильно, браузеры клиентов могут отображать сообщения о невозможности проверки подлинности сертификата.

Пример

```
const https = require('https');
const fs = require('fs');
const options = {
    key: fs.readFileSync('privatekey.pem'),
    cert: fs.readFileSync('certificate.pem'),
    ca: [fs.readFileSync('1_ca.crt'), fs.readFileSync('2_ca.crt')]
};
https.createServer(options, (req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
}).listen(8000);
```

Использование HTTPS

Минимальная настройка HTTPS-сервера в Node.js выглядит следующим образом:

```
const https = require('https');
const fs = require('fs');
const httpsOptions = {
    key: fs.readFileSync('path/to/server-key.pem'),
    cert: fs.readFileSync('path/to/server-crt.pem')
};
const app = function (req, res) {
    res.writeHead(200);
    res.end("hello world\n");
}
https.createServer(httpsOptions, app).listen(4433);
```

Если вы также хотите поддерживать HTTP-запросы, вам нужно внести небольшое изменение:

```
const http = require('http');
const https = require('https');
const fs = require('fs');
```

```
const httpsOptions = {
    key: fs.readFileSync('path/to/server-key.pem'),
    cert: fs.readFileSync('path/to/server-crt.pem')
};
const app = function (req, res) {
    res.writeHead(200);
    res.end("hello world\n");
}
http.createServer(app).listen(8888);
https.createServer(httpsOptions, app).listen(4433);
```

Настройка HTTPS-сервера

Как только вы установите Node.js, выполните следующую процедуру, чтобы запустить базовый веб-сервер с поддержкой как HTTP, так и HTTPS.

Шаг 1. Создайте центр сертификации

Создайте папку, где вы будете хранить ключ и сертификат:

```
mkdir conf
```

Перейдите в этот каталог:

```
cd conf
```

Скачайте файл `ca.cnf` для использования в качестве конфигурационного шаблона:

```
wget     https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/ca.cnf
```

Создайте новый центр сертификации с использованием этой конфигурации:

```
openssl req -new -x509 -days 9999 -config ca.cnf -keyout ca-key.pem -out ca-cert.pem
```

Теперь, когда у нас есть центр сертификации в файлах `ca-key.pem` и `ca-cert.pem`, давайте создадим приватный ключ для сервера:

```
openssl genrsa -out key.pem 4096
```

Скачайте файл `server.cnf` для использования в качестве конфигурационного шаблона:

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/server.cnf
```

Создайте запрос на подпись сертификата, используя эту конфигурацию:

```
openssl req -new -config server.cnf -key key.pem -out csr.pem
```

Подпишите запрос:

```
openssl x509 -req -extfile server.cnf -days 999 -passin "pass:password"
-in csr.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out
cert.pem
```

Шаг 2. Установите свой сертификат в качестве корневого

Скопируйте ваш сертификат в папку с корневыми сертификатами:

```
sudo cp ca-cert.pem /usr/local/share/ca-certificates/ca-cert.pem
```

Обновите хранилище сертификатов CA:

```
sudo update-ca-certificates
```

Защита приложения на express.js 3

Конфигурация для создания защищенного соединения с использованием express.js (с версии 3):

```
var fs = require('fs');
var http = require('http');
var https = require('https');
var privateKey = fs.readFileSync('sslcert/server.key', 'utf8');
var certificate = fs.readFileSync('sslcert/server.crt', 'utf8');
// Определите ваш ключ и сертификат
var credentials = {key: privateKey, cert: certificate};
var express = require('express');
var app = express();
// ваша конфигурация express здесь
var httpServer = http.createServer(app);
var httpsServer = https.createServer(credentials, app);
// Использование порта 8080 для http и 8443 для https
httpServer.listen(8080);
httpsServer.listen(8443);
```

Таким образом вы обеспечиваете работу middleware express на нативных серверах http/https.

Если вы хотите, чтобы ваше приложение работало на портах ниже 1024, вам нужно использовать команду sudo (не рекомендуется) или обратный прокси-сервер (например, nginx, haproxy).

Глава 94.
Отправка веб-уведомлений

Примеры

Отправка веб-уведомления с использованием GCM (Google Cloud Messaging System)

Такой пример широко известен среди PWA (прогрессивных веб-приложений), и в этом примере мы собираемся отправить простое уведомление с помощью Node.js и ES6.

1. Установите модуль Node-GCM:

```
npm install node-gcm
```

2. Установите Socket.io:

```
npm install socket.io
```

3. Создайте приложение с поддержкой GCM, используя Google Console.
4. Получите ваш GCM Application Id (он понадобится позже).
5. Получите ваш секретный код приложения GCM.
6. Откройте ваш любимый редактор кода и добавьте следующий код:

```
'use strict';
const express = require('express');
const app = express();
const gcm = require('node-gcm');
app.io = require('socket.io')();
// [*] Настраиваем наш GCM-канал.
const sender = new gcm.Sender('Project Secret');
const regTokens = [];
let message = new gcm.Message({
    data: {
        key1: 'msg1'
```

```
    }
});
// [*] Настраиваем статические файлы.
app.use(express.static('public/'));
// [*] Настраиваем маршруты.
app.get('/', (req, res) => {
    res.sendFile(__dirname + '/public/index.html');
});
// [*] Настраиваем соединение через Socket.
app.io.on('connection', socket => {
    console.log('we have a new connection ...');
    socket.on('new_user', (reg_id) => {
        // [*] Добавляем регистрационный токен уведомлений пользовате-
ля в наш список, обычно
        // скрытый в безопасном месте.
        if (regTokens.indexOf(reg_id) === -1) {
            regTokens.push(reg_id);
            // [*] Отправляем push-уведомления
            sender.send(message, {
                registrationTokens: regTokens
            }, (err, response) => {
                if (err) console.error('err', err);
                else console.log(response);
            });
        }
    })
});
module.exports = app;
```

PS: Здесь мы использовали специальный хак, чтобы `Socket.io` работал с Express, потому что просто так он не работает.

Теперь создайте .json-файл и назовите его `Manifest.json`, откройте его и вставьте следующее:

```
{
    "name": "Application Name",
    "gcm_sender_id": "GCM Project ID"
}
```

Закройте файл и сохраните его в корневом каталоге вашего приложения.

PS: Файл `Manifest.json` должен находиться в корневом каталоге, иначе он не будет работать.

В приведенном выше коде мы сделали следующее:

1. Настроили и отправили обычную страницу `index.html`, которая также будет использовать `socket.io`. Слушали событие подключения, которое запускается с фронтенда, то есть с нашей страницы `index.html` (оно будет запущено, как только новый клиент успешно подключится к нашему предопределенному адресу).

2. Отправили специальный токен, известный как регистрационный токен, с нашей страницы `index.html` через событие `socket.io new_user`. Этот токен будет уникальным кодом доступа пользователя, и каждый код обычно генерируется поддерживающим браузером для API веб-уведомлений (читайте больше здесь: https://developer.mozilla.org/en-US/docs/Web/API/Notification).

3. Использовали модуль `node-gcm` для отправки нашего уведомления, которое будет обработано и показано позже с использованием Service Workers.

PS: Полный рабочий пример вы можете найти здесь: https://github.com/houssem-yahiaoui/progressive-web-app-starter-kit.

Глава 95.
Отправка файлового потока клиенту

Примеры

Использование fs и pipe для потоковой передачи статических файлов с сервера

Хороший сервис VOD (Video On Demand) должен начинаться с основ. Допустим, у вас есть каталог на сервере, который не доступен публично, но через некоторый портал или платный доступ вы хотите разрешить пользователям доступ к вашим медиафайлам:

```
var movie = path.resolve('./public/' + req.params.filename);
fs.stat(movie, function (err, stats) {
    var range = req.headers.range;
    if (!range) {
        return res.sendStatus(416);
    }
    // Логика для фрагментов (чанков)
    var positions = range.replace(/bytes=/, "").split("-");
    var start = parseInt(positions[0], 10);
    var total = stats.size;
    var end = positions[1] ? parseInt(positions[1], 10) : total - 1;
    var chunksize = (end - start) + 1;
    res.writeHead(206, {
        'Transfer-Encoding': 'chunked',
        "Content-Range": "bytes " + start + "-" + end + "/" + total,
        "Accept-Ranges": "bytes",
        "Content-Length": chunksize,
        "Content-Type": mime.lookup(req.params.filename)
    });
    var stream = fs.createReadStream(movie, { start: start, end: end,
autoClose: true })
        .on('end', function () {
```

```
            console.log('Stream Done');
        })
        .on("error", function (err) {
            res.end(err);
        })
        .pipe(res, { end: true });
});
```

Приведенный выше пример — это базовый план, как можно передавать видео клиенту. Логика для фрагментов зависит от множества факторов, включая сетевой трафик и задержки. Важно сбалансировать размер фрагментов и их количество.

Наконец, вызов .pipe дает понять Node.js, что необходимо поддерживать соединение с сервером открытым и отправлять дополнительные фрагменты по мере необходимости.

Потоковая передача с использованием fluent-ffmpeg

Вы также можете использовать fluent-ffmpeg для конвертации файлов .mp4 в .flv или другие форматы:

```
res.contentType('flv');
var pathToMovie = './public/' + req.params.filename;
var proc = ffmpeg(pathToMovie)
    .preset('flashvideo')
    .on('end', function () {
        console.log('Stream Done');
    })
    .on('error', function (err) {
        console.log('an error happened: ' + err.message);
        res.send(err.message);
    })
    .pipe(res, { end: true });
```

Глава 96. Sequelize.js

Примеры

Установка

Убедитесь, что у вас установлены Node.js и NPM. Затем установите `sequelize.js` с помощью NPM :

```
npm install --save sequelize
```

Также вам нужно установить модули Node.js для поддерживаемых баз данных. Вам следует установить только тот модуль, который вы используете.

Для MySQL и MariaDB:

```
npm install --save mysql
```

Для PostgreSQL:

```
npm install --save pg pg-hstore
```

Для SQLite:

```
npm install --save sqlite
```

Для MSSQL:

```
npm install --save tedious
```

Как только вы установили все необходимое, вы можете подключить и создать новый экземпляр Sequelize следующим образом.

Синтаксис ES5:

```
var Sequelize = require('sequelize');
var sequelize = new Sequelize('database', 'username', 'password');
```

Синтаксис ES6 с Babel stage-0:

```
import Sequelize from 'sequelize';
```

```
const sequelize = new Sequelize('database', 'username', 'password');
```

Теперь у вас есть экземпляр Sequelize, доступный для использования. Вы можете назвать его по-другому, если хотите, например:

```
var db = new Sequelize('database', 'username', 'password');
```

или

```
var database = new Sequelize('database', 'username', 'password');
```

Эта часть — на ваше усмотрение. После установки вы можете использовать его в своем приложении в соответствии с документацией API (https://sequelize.org/v3/api/sequelize/).

Следующий шаг после установки — создание вашей собственной модели. Подробнее об этом можно узнать, перейдя по ссылке: http://docs.sequelizejs.com/en/v3/docs/getting-started/#your-first-model.

Определение моделей

Существует два способа определения моделей в Sequelize: с помощью `sequelize.define(...)` или `sequelize.import(...)`. Обе функции возвращают объект модели Sequelize.

1. sequelize.define(modelName, attributes, [options])

Этот способ подходит, если вы хотите определить все свои модели в одном файле или иметь дополнительный контроль над определением модели.

```
/* Инициализация Sequelize */
const config = {
    username: "database username",
    password: "database password",
    database: "database name",
    host: "database's host URL",
     dialect: "mysql" // Другие варианты: postgres, sqlite, mariadb,
mssql.
}
var Sequelize = require("sequelize");
var sequelize = new Sequelize(config);

/* Определение моделей */
sequelize.define("MyModel", {
    name: Sequelize.STRING,
    comment: Sequelize.TEXT,
    date: {
        type: Sequelize.DATE,
        allowNull: false
    }
});
```

Дополнительную информацию и другие примеры можно найти в документации по doclets: https://doclets.io/

или в документации sequelize.com: https://sequelize.org/v3/docs/models-definition/.

2. sequelize.import(path)

Если ваши определения моделей разбиты на отдельные файлы, то метод `import` будет вам чрезвычайно полезен. В файле, где вы инициализируете `Sequelize`, вызовите `import` следующим образом:

```
/* Инициализация Sequelize */
// См. предыдущий пример кода для инициализации

/* Определение моделей */
sequelize.import("./models/my_model.js"); // Путь может быть относительным или абсолютным
```

В ваших файлах определения модели код будет выглядеть следующим образом:

```
module.exports = function(sequelize, DataTypes) {
    return sequelize.define("MyModel", {
        name: DataTypes.STRING,
        comment: DataTypes.TEXT,
        date: {
            type: DataTypes.DATE,
            allowNull: false
        }
    });
};
```

Глава 97.
Простой CRUD API на основе REST

Примеры

REST API для CRUD в Express 3+

```
var express = require("express"),
    bodyParser = require("body-parser"),
    server = express();

// body parser для парсинга тела запроса

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

// временное хранилище для `item` в памяти
var itemStore = [];

// Получение всех элементов
server.get('/item', function (req, res) {
    res.json(itemStore);
});

// Получение элемента с указанным id
server.get('/item/:id', function (req, res) {
    res.json(itemStore[req.params.id]);
});

// Добавление нового элемента
server.post('/item', function (req, res) {
    itemStore.push(req.body);
    res.json(req.body);
});
```

```
// Обновление элемента по указанному id
server.put('/item/:id', function (req, res) {
    itemStore[req.params.id] = req.body
    res.json(req.body);
});

// Удаление элемента с указанным id
server.delete('/item/:id', function (req, res) {
    itemStore.splice(req.params.id, 1)
    res.json(req.body);
});

// Запуск сервера
server.listen(3000, function () {
    console.log("Server running");
})
```

Глава 98.
Связь через Socket.io

Примеры

«Hello world!» с сообщениями через сокеты

Установка модулей Node.js:

```
npm install express
npm install socket.io
```

Сервер на Node.js

```
const express = require('express');
const app = express();
const server = app.listen(3000, console.log("Socket.io Hello World
server started!"));
const io = require('socket.io')(server);

io.on('connection', (socket) => {
    // console.log("Client connected!");
    socket.on('message-from-client-to-server', (msg) => {
        console.log(msg);
    })
    socket.emit('message-from-server-to-client', 'Hello World!');
});
```

Браузерный клиент

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello World with Socket.io</title>
</head>
<body>
```

```
    <script src="https://cdn.socket.io/socket.io-1.4.5.js"></script>
    <script>
        var socket = io("http://localhost:3000");
        socket.on("message-from-server-to-client", function(msg) {
            document.getElementById('message').innerHTML = msg;
        });
        socket.emit('message-from-client-to-server', 'Hello World!');
    </script>
    <p>Socket.io Hello World client started!</p>
    <p id="message"></p>
</body>
</html>
```

Глава 99.
Синхронное и асинхронное программирование в Node.js

Примеры

Использование async

Пакет async предоставляет функции для асинхронного кода.

С помощью функции auto можно определить асинхронные отношения между двумя или более функциями:

```
var async = require('async');

async.auto({
    get_data: function(callback) {
        console.log('in get_data');
        // асинхронный код для получения данных
        callback(null, 'data', 'converted to array');
    },
    make_folder: function(callback) {
        console.log('in make_folder');
        // асинхронный код для создания директории для хранения файла
        // выполняется одновременно с получением данных
        callback(null, 'folder');
    },
   write_file: ['get_data', 'make_folder', function(results, callback)
{
        console.log('in write_file', JSON.stringify(results));
       // как только данные будут получены и директория будет создана,
        // записываем данные в файл в директории
        callback(null, 'filename');
    }],
    email_link: ['write_file', function(results, callback) {
        console.log('in email_link', JSON.stringify(results));
```

```
        // после того как файл записан, отправим ссылку на него
по электронной почте...
        // results.write_file содержит имя файла, возвращаемое функ-
цией write_file.
        callback(null, {'file':results.write_file, 'email':'user@
example.com'});
    }]
}, function(err, results) {
    console.log('err = ', err);
    console.log('results = ', results);
});
```

Этот код мог бы быть выполнен синхронно, просто путем вызова `get_data`, `make_folder`, `write_file` и `email_link` в правильном порядке. Async отслеживает результаты за вас, и если произошла ошибка (первый параметр `callback` не равен `null`), выполнение остальных функций прекращается.

Глава 100.
TCP-сокеты

Примеры

Простой TCP-сервер

```
// Подключение модуля net Node.js
const Net = require('net');

// Порт, на котором сервер прослушивает.
const port = 8080;

// Используйте net.createServer() в вашем коде. Это только для примера.
// Создание нового TCP-сервера.
const server = new Net.Server();

// Сервер слушает сокет для получения запроса на подключение от клиента.
// Сокет можно представить как конечную точку.
server.listen(port, function() {
     console.log(`Server listening for connection requests on socket
localhost:${port}.`);
});

// Когда клиент запрашивает соединение с сервером, сервер создает новый
// сокет, предназначенный для этого клиента.
server.on('connection', function(socket) {
    console.log('A new connection has been established.');

     // Теперь, когда TCP-соединение установлено, сервер может отпра-
вить данные
    // клиенту, записав их в его сокет.
    socket.write('Hello, client.');
```

```
    // Сервер также может получать данные от клиента, читая из его сокета.
    socket.on('data', function(chunk) {
        console.log(`Data received from client: ${chunk.toString()}.`);
    });
    // Когда клиент запрашивает завершение TCP-соединения с сервером,
сервер
    // завершает соединение.
    socket.on('end', function() {
        console.log('Closing connection with the client');
    });

    // Не забудьте обработать ошибки для вашего же блага.
    socket.on('error', function(err) {
        console.log(`Error: ${err}`);
    });
});
```

Простой TCP-клиент

```
// Подключение модуля net Node.js
const Net = require('net');

// Номер порта и имя хоста сервера.
const port = 8080;
const host = 'localhost';

// Создание нового TCP-клиента.
const client = new Net.Socket();

// Отправка запроса на соединение серверу.
client.connect({ port: port, host: host }, function() {
    // Если ошибки нет, сервер принял запрос и создал новый
    // сокет, предназначенный для нас.
    console.log('TCP connection established with the server.');

    // Клиент теперь может отправить данные серверу, записав их в свой
сокет.
    client.write('Hello, server.');
});

// Клиент также может получать данные от сервера, читая из своего сокета.
```

```
client.on('data', function(chunk) {
   console.log(`Data received from the server: ${chunk.toString()}.`);
    // Запрос завершения соединения после получения данных.
    client.end();
});

client.on('end', function() {
    console.log('Requested an end to the TCP connection');
});
```

Глава 101.
Шаблонные фреймворки

Примеры

Nunjucks

Серверный движок с поддержкой наследования блоков, автоматическим экранированием, макросами, асинхронным управлением и многим другим. Его создатели вдохновлялись jinja2, и он очень похож на Twig (php).

Документация — http://mozilla.github.io/nunjucks/.

Установка — `npm i nunjucks`.

Ниже приведен пример базового использования с Express.

app.js

```
var express = require('express');
var nunjucks = require('nunjucks');
var app = express();
app.use(express.static('/public'));

// Применяем nunjucks и добавляем пользовательский фильтр и функцию (например).
```

```
var env = nunjucks.configure(['views/'], { // устанавливаем папки с ша-
блонами
    autoescape: true,
    express: app
});

env.addFilter('myFilter', function(obj, arg1, arg2) {
    console.log('myFilter', obj, arg1, arg2);
    // Делать что-то с obj
    return obj;
});

env.addGlobal('myFunc', function(obj, arg1) {
    console.log('myFunc', obj, arg1);
    // Делать что-то с obj
    return obj;
});

app.get('/', function(req, res){
    res.render('index.html', {title: 'Main page'});
});

app.get('/foo', function(req, res){
    res.locals.smthVar = 'This is Sparta!';
    res.render('foo.html', {title: 'Foo page'});
});

app.listen(3000, function() {
    console.log('Example app listening on port 3000...');
});
```

/views/index.html

```
<html>
<head>
    <title>Nunjucks example</title>
</head>
<body>
    {% block content %}
        {{title}}
    {% endblock %}
</body>
</html>
```

/views/foo.html

```
{% extends "index.html" %}

{# Это комментарий #}
{% block content %}
    <h1>{{title}}</h1>
    {# применяем пользовательскую функцию и затем встроенные и пользо-
вательские фильтры #}
    {{ myFunc(smthVar) | lower | myFilter(5, 'abc') }}
{% endblock %}
```

Глава 102.
Удаление Node.js

Примеры

Полное удаление Node.js на Mac OSX

В терминале на операционной системе Mac введите следующие две команды:

```
lsbom -f -l -s -pf /var/db/receipts/org.nodejs.pkg.bom | while read
f; do sudo rm /usr/local/${f}; done
sudo rm -rf /usr/local/lib/node /usr/local/lib/node_modules /var/db/
receipts/org.nodejs.*
```

Удаление Node.js на Windows

Чтобы удалить Node.js на Windows, используйте «Установка и удаление программ» следующим образом:

1. Откройте «Установка и удаление программ» через меню «Пуск».
2. Найдите Node.js.

Windows 10:

3. Нажмите на Node.js.
4. Нажмите «Удалить».
5. Нажмите новую кнопку «Удалить».

Windows 7-8.1:

6. Нажмите кнопку «Удалить» под Node.js.

Глава 103.
Фреймворки для модульного тестирования

Примеры

Mocha синхронный

```
describe('Suite Name', function() {
    describe('#method()', function() {
        it('should run without an error', function() {
            expect([ 1, 2, 3 ].length).to.be.equal(3)
        })
    })
})
```

Mocha асинхронный (callback)

```
var expect = require("chai").expect;
describe('Suite Name', function() {
    describe('#method()', function() {
        it('should run without an error', function(done) {
            testSomething(err => {
                expect(err).to.not.be.equal(null)
                done()
            })
        })
    })
})
```

Mocha асинхронный (Promise)

```
describe('Suite Name', function() {
    describe('#method()', function() {
```

```
        it('should run without an error', function() {
            return doSomething().then(result => {
                expect(result).to.be.equal('hello world')
            })
        })
    })
})
```

Mocha асинхронный (async/await)

```
const { expect } = require('chai')
describe('Suite Name', function() {
    describe('#method()', function() {
        it('should run without an error', async function() {
            const result = await answerToTheUltimateQuestion()
            expect(result).to.be.equal(42)
        })
    })
})
```

Глава 104.
Сценарии использования Node.js

Примеры

HTTP-сервер

```
const http = require('http');
console.log('Starting server...');
var config = {
    port: 80,
    contentType: 'application/json; charset=utf-8'
};
// JSON-API сервер на порту 80
var server = http.createServer();
server.listen(config.port);
server.on('error', (err) => {
    if (err.code == 'EADDRINUSE') console.error('Port ' + config.port
+ ' is already in use');
    else console.error(err.message);
});
server.on('request', (request, res) => {
    var remoteAddress = request.headers['x-forwarded-for'] ||
        request.connection.remoteAddress; // Адрес клиента
    console.log(remoteAddress + ' ' + request.method + ' ' + request.
url);
    var out = {};
    // Здесь можно изменить вывод в зависимости от `request.url`
    out.test = request.url;
    res.writeHead(200, {
        'Content-Type': config.contentType
    });
    res.end(JSON.stringify(out));
});
```

```
server.on('listening', () => {
    c.info('Server is available: http://localhost:' + config.port);
});
```

Консоль с командной строкой

```
const process = require('process');
const rl = require('readline').createInterface(process.stdin, process.
stdout);
rl.pause();
console.log('Something long is happening here...');
var cliConfig = {
    promptPrefix: ' > '
}

/*
 Распознавание команд
 НАЧАЛО
*/
var commands = {
    eval: function(arg) { // Попробуйте ввести в консоли: eval 2 * 10
^ 3 + 2 ^ 4
        arg = arg.join(' ');
        try { console.log(eval(arg)); }
        catch (e) { console.log(e); }
    },
    exit: function(arg) {
        process.exit();
    }
};

rl.on('line', (str) => {
    rl.pause();
    var arg = str.trim().match(/([^"]+)|("(?:[^"\\]|\\.)+")/g);
// Применение регулярного выражения для удаления всех пробелов,
кроме тех, что между двойными кавычками: http://stackoverflow.
com/a/14540319/2396907
    if (arg) {
        for (let n in arg) {
            arg[n] = arg[n].replace(/^\"|\"$/g, '');
        }
        var commandName = arg[0];
        var command = commands[commandName];
        if (command) {
```

```
            arg.shift();
            command(arg);
        }
        else console.log('Command "' + commandName + '" doesn\'t exist');
    }
    rl.prompt();
});

/*
 КОНЕЦ
 распознавания команд
*/

rl.setPrompt(cliConfig.promptPrefix);
rl.prompt();
```

Глава 105.
Использование Browserify для устранения ошибки ‘required’ в браузерах

Примеры

Пример — file.js

В этом примере у нас есть файл под названием `file.js`.

Предположим, вам нужно разобрать URL с использованием модуля `querystring` в JavaScript и Node.js. Для этого достаточно вставить в файл следующий оператор:

```
const querystring = require('querystring');
var ref = querystring.parse("foo=bar&abc=xyz&abc=123");
```

Что делает этот фрагмент кода?

Во-первых, мы создаем модуль `querystring`, который предоставляет утилиты для разбора и форматирования строк запроса URL. Доступ к нему можно получить с помощью:

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

Затем мы разбираем URL с использованием метода `.parse()`. Этот метод разбирает строку запроса URL (str) в коллекцию пар «ключ-значение».

Например, строка запроса ‘`foo=bar&abc=xyz&abc=123`’ разбирается в

```
{ foo: 'bar', abc: ['xyz', '123'] }
```

К сожалению, в браузерах метод `require` не определен, но он доступен в Node.js.

Установка Browserify

С помощью Browserify вы можете писать код, который использует `require` так же, как это делается в Node. Как решить проблему? Это просто.

1. Сначала установите Node.js, который поставляется с NPM. Затем выполните:

```
npm install -g browserify
```

2. Перейдите в каталог, в котором находится ваш `file.js`, и установите модуль `querystring` с помощью `npm`:

```
npm install querystring
```

Примечание: если вы не перейдете в конкретный каталог, команда не выполнится, потому что не удастся найти файл, содержащий модуль.

3. Теперь рекурсивно объедините все необходимые модули, начиная с file.js, в один файл под названием `bundle.js` (или любое другое название) с помощью команды browserify:

```
browserify file.js -o bundle.js
```

Browserify анализирует абстрактное синтаксическое дерево для вызовов `require()` и проходит по всему графу зависимостей вашего проекта.

4. Наконец, добавьте единственный тег в ваш HTML, и все готово!

```
<script src="bundle.js"></script>
```

Что происходит? Вы получаете комбинацию вашего старого `.js`-файла (то есть `file.js`) и вашего вновь созданного файла `bundle.js`. Эти два файла объединяются в один.

Важно

Имейте в виду, что если вы хотите внести какие-либо изменения в `file.js`, это не повлияет на поведение вашей программы. Ваши изменения вступят в силу только в том случае, если вы отредактируете вновь созданный `bundle.js`.

Что это значит?

Это означает, что если по какой-либо причине вы хотите отредактировать `file.js`, изменения не будут иметь эффекта. Вам действительно придется редактировать `bundle.js`, так как он представляет собой объединение `bundle.js` и `file.js`.

Глава 106. Использование IISNode для размещения веб-приложений Node.js в IIS

Примечания

Проблема с виртуальным каталогом / вложенным приложением с представлениями

Если вы собираетесь использовать Express для рендеринга представлений с помощью движка представлений, вам нужно передать значение `virtualDirPath` в ваши представления:

```
res.render('index', { virtualDirPath: virtualDirPath });
```

Суть этого заключается в том, чтобы ваши гиперссылки на другие представления, размещенные вашим приложением, и пути к статическим ресурсам знали, где размещен сайт, без необходимости изменять все представления после развертывания. Это одна из наиболее неприятных и трудоемких проблем при использовании виртуальных каталогов с IISNode.

Версии

Все приведенные выше примеры работают с:

- Express v4.x
- IIS 7.x/8.x
- Socket.io v1.3.x или выше

Примеры

Начало работы

IISNode позволяет размещать веб-приложения Node.js на IIS 7/8 так же, как и приложение .NET. Конечно, вы можете самостоятельно разместить процесс

node.exe на Windows, но зачем это делать, если можно просто запустить ваше приложение в IIS?

IISNode будет управлять масштабированием на несколько ядер, процессом node.exe и автоматической перезагрузкой вашего IIS-приложения при обновлении вашего приложения, и это лишь некоторые из его преимуществ.

Требования

IISNode предъявит вам несколько требований перед тем, как вы сможете разместить ваше Node.js-приложение в IIS.

1. Node.js должен быть установлен на хосте IIS 32-битной или 64-битной версии — обе поддерживаются.

2. IISNode должен быть установлен x86 или x64, это должно соответствовать разрядности вашего хоста IIS.

3. Модуль Microsoft URL-Rewrite для IIS должен быть установлен на вашем хосте IIS.

Это важно, иначе запросы к вашему приложению Node.js не будут работать должным образом.

4. Файл `Web.config` должен находиться в корневой папке вашего приложения Node.js.

5. Конфигурация IISNode через файл `iisnode.yml` или элемент `<iisnode>` в вашем `Web.config`.

Простой пример Hello World с использованием Express

Чтобы этот пример работал, вам нужно создать приложение IIS 7/8 на вашем хосте IIS и добавить каталог, содержащий веб-приложение Node.js, как физический каталог. Убедитесь, что учетная запись Application/Application Pool Identity имеет доступ к установленному Node.js. В этом примере используется 64-битная установка Node.js.

Структура проекта

Это базовая структура проекта веб-приложения IISNode/Node.js. Она почти идентична любой веб-программе, не использующей IISNode, за исключением добавления файла `Web.config`.

```
-/app_root
-package.json
-server.js
-Web.config
```

server.js-приложение на Express

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

// Нам нужно получить порт, который IISNode передает нам,
// используя переменную окружения PORT, если она не установлена, ис-
пользуйте значение по умолчанию
const port = process.env.PORT || 3000;

// Настройка маршрута в корне нашего приложения
server.get('/', (req, res) => {
    return res.status(200).send('Hello World');
});
server.listen(port, () => {
    console.log(`Listening on ${port}`);
});
```

Конфигурация и Web.config

Файл `Web.config` точно такой же, как и любой другой `Web.config` для IIS, за исключением того, что должны присутствовать следующие два элемента: URL <rewrite><rules> и IISNode <handler>. Оба этих элемента являются дочерними для <system.webServer>.

Конфигурация

Вы можете настроить IISNode с помощью файла `iisnode.yml` или добавив элемент `<iisnode>` как дочерний элемент `<system.webServer>` в вашем `Web.config`. Оба этих конфигурационных файла можно использовать одновременно. Однако в этом случае `Web.config` должен указывать на файл `iisnode.yml`, и любые конфликты конфигураций будут взяты из файла `iisnode.yml`. Такое переопределение конфигурации не может происходить в обратном порядке.

Обработчик IISNode

Чтобы IIS знал, что файл `server.js` содержит наше веб-приложение Node.js, нам нужно явно указать это. Мы можем сделать это, добавив обработчик `IISNode` `<handler>` в элемент `<handlers>`.

```
<handlers>
    <add name="iisnode" path="server.js" verb="*" modules="iisnode"/>
</handlers>
```

Правила URL-Rewrite

Последний этап конфигурации заключается в том, чтобы убедиться, что трафик, предназначенный для нашего Node.js-приложения, поступающий в экземпляр IIS, перенаправляется на IISNode. Без правил переписывания URL нам пришлось бы обращаться к нашему приложению по адресу `http://<host>/server.js`, и что еще хуже, при попытке запроса ресурса, предоставляемого `server.js`, вы получите ошибку 404. Именно поэтому переписывание URL необходимо для веб-приложений IISNode.

```
<rewrite>
    <rules>
        <!-- Сначала проверяем, соответствует ли входящий URL физическому файлу в папке /public -->
        <rule name="StaticContent" patternSyntax="Wildcard">
     <action type="Rewrite" url="public/{R:0}" logRewrittenUrl="true"/>
            <conditions>
                  <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true"/>
            </conditions>
            <match url="*.*"/>
        </rule>

        <!-- Все остальные URL перенаправляются на точку входа приложения Node.js -->
        <rule name="DynamicContent">
            <conditions>
                  <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="True"/>
            </conditions>
            <action type="Rewrite" url="server.js"/>
        </rule>
    </rules>
</rewrite>
```

Это рабочий файл `Web.config` для данного примера, настроенный для 64-битной установки Node.js. Вот и все, теперь откройте свой сайт IIS и убедитесь, что ваше приложение Node.js работает.

Использование виртуального каталога IIS или вложенного приложения

Использование виртуального каталога или вложенного приложения в IIS — это обычный сценарий, и, скорее всего, вы захотите применить его, используя IISNode.

IISNode не предоставляет прямую поддержку виртуальных каталогов или вложенных приложений через конфигурацию, поэтому для достижения этого мы должны воспользоваться функцией IISNode, которая не является частью конфигурации и гораздо менее известна. Все дочерние элементы элемента <appSettings> в Web.config добавляются в объект process.env в качестве свойств с использованием ключа appSetting.

Создадим виртуальный каталог в наших <appSettings>:

```
<appSettings>
    <add key="virtualDirPath" value="/foo" />
</appSettings>
```

В нашем приложении Node.js мы можем получить доступ к настройке virtualDirPath:

```
console.log(process.env.virtualDirPath); // выводит /foo
```

Теперь, когда мы можем использовать элемент <appSettings> для конфигурации, давайте воспользуемся этим в нашем серверном коде:

```
// Доступ к настройкам virtualDirPath и присвоение значения по умол-
чанию '/'
var virtualDirPath = process.env.virtualDirPath || '/';

// Также мы хотим убедиться, что virtualDirPath
// всегда начинается с косой черты
if (!virtualDirPath.startsWith('/', 0))
    virtualDirPath = '/' + virtualDirPath;

// Настройка маршрута в корне нашего приложения
server.get(virtualDirPath, (req, res) => {
    return res.status(200).send('Hello World');
});
```

Мы можем использовать virtualDirPath и для наших статических ресурсов

```
// Публичный каталог
server.use(express.static(path.join(virtualDirPath, 'public')));

// Bower
server.use('/bower_components',                    express.static(path.
join(virtualDirPath, 'bower_components')));
```

Соединим все это вместе:

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

```
const port = process.env.PORT || 3000;
// Доступ к настройкам virtualDirPath и присвоение значения по умол-
чанию '/'
// в случае, если оно не существует или не установлено
var virtualDirPath = process.env.virtualDirPath || '/';

// Также мы хотим убедиться, что virtualDirPath
// всегда начинается с косой черты
if (!virtualDirPath.startsWith('/', 0))
    virtualDirPath = '/' + virtualDirPath;

// Публичный каталог
server.use(express.static(path.join(virtualDirPath, 'public')));

// Bower
server.use('/bower_components',                    express.static(path.
join(virtualDirPath, 'bower_components')));

// Настройка маршрута в корне нашего приложения
server.get(virtualDirPath, (req, res) => {
    return res.status(200).send('Hello World');
});

server.listen(port, () => {
    console.log(`Listening on ${port}`);
});
```

Здесь вы найдете рабочий файл `Web.config` для данного примера, настроенный для установки 64-разрядной версии Node.js: https://gist.github.com/pbaio/f63918181d8d7f8ee1d2.

Вот и все, теперь посетите свой сайт IIS и посмотрите, как работает ваше Node.js-приложение.

Использование Socket.io с IISNode

Для работы Socket.io с IISNode единственные изменения, необходимые при отсутствии использования виртуального каталога/вложенного приложения, заключаются в файле `Web.config`.

Поскольку Socket.io отправляет запросы, начинающиеся с `/socket.io`, IISNode должен сообщить IIS, что эти запросы также должны обрабатываться IISNode, а не просто как запросы на статические файлы или другой трафик. Для этого требуется другой `<handler>`, чем для стандартных приложений IISNode.

```
<handlers>
    <add name="iisnode-socketio" path="server.js" verb="*"
modules="iisnode" />
</handlers>
```

Помимо изменений в `<handlers>` нам также нужно добавить дополнительное правило переписывания URL. Это правило отправляет весь трафик `/socket.io` в наш серверный файл, где запущен сервер Socket.io.

```
<rule name="SocketIO" patternSyntax="ECMAScript">
    <match url="socket.io.+"/>
    <action type="Rewrite" url="server.js"/>
</rule>
```

Если вы используете IIS 8, вам нужно отключить настройку `webSockets` в вашем `Web.config` в дополнение к добавлению указанного выше обработчика и правил переписывания. Это не требуется в IIS 7, так как там нет поддержки `webSocket`.

```
<webSocket enabled="false" />
```

Глава 107.
Использование потоков

Параметры

Параметр	Определение
Readable	Тип потока, из которого можно считывать данные.
Writable	Тип потока, в который можно записывать данные.
Duplex	Тип потока, который является как читаемым, так и записываемым.
Transform	Тип дуплексного потока, который может преобразовывать данные во время их считывания и последующей записи.

Примеры

Считывание данных из текстового файла с использованием потоков

Ввод/вывод (I/O) в Node.js является асинхронным, поэтому взаимодействие с диском и сетью включает передачу обратных вызовов (callback) функциям. Вы можете написать код, который обслуживает файл с диска следующим образом:

```
var http = require('http');
var fs = require('fs');
var server = http.createServer(function (req, res) {
    fs.readFile(__dirname + '/data.txt', function (err, data) {
        res.end(data);
    });
});
server.listen(8000);
```

Этот код работает, но он громоздкий и буферизует весь файл `data.txt` в память для каждого запроса перед тем, как отправить результат обратно клиентам. Если файл `data.txt` очень большой, ваша программа может начать потреблять много памяти, обслуживая множество пользователей одновременно, особенно пользователей с медленным подключением. Пользователям нужно будет ждать, пока весь файл не будет загружен в память на вашем сервере, прежде чем они начнут получать его содержимое.

К счастью, оба аргумента (`req`, `res`) являются потоками, а это означает, что данный код можно написать гораздо лучше, используя `fs.createReadStream()` вместо `fs.readFile()`:

```
var http = require('http');
var fs = require('fs');
var server = http.createServer(function (req, res) {
    var stream = fs.createReadStream(__dirname + '/data.txt');
    stream.pipe(res);
});
server.listen(8000);
```

Здесь метод `.pipe()` сам заботится о прослушивании событий 'data' и 'end' от `fs.createReadStream()`. Этот код не только более чистый, но и теперь файл `data.txt` будет отправляться клиентам по частям (чанкам) сразу же, как только эти части будут считаны с диска.

Перенаправление потоков (Piping streams)

Читаемые потоки могут быть перенаправлены (piped) или соединены с записываемыми потоками. Это позволяет данным перемещаться от исходного потока к целевому без особых усилий:

```
var fs = require('fs')
var readable = fs.createReadStream('file1.txt')
var writable = fs.createWriteStream('file2.txt')
readable.pipe(writable) // возвращает writable
```

Когда записываемые потоки также являются читаемыми, то есть когда они являются дуплексными потоками, вы можете продолжать перенаправлять их на другие записываемые потоки:

```
var zlib = require('zlib')
fs.createReadStream('style.css')
    .pipe(zlib.createGzip()) // Возвращаемый объект, zlib.Gzip, является дуплексным потоком.
    .pipe(fs.createWriteStream('style.css.gz'))
```

Читаемые потоки также могут быть перенаправлены в несколько потоков.

```
var readable = fs.createReadStream('source.css')
readable.pipe(zlib.createGzip()).pipe(fs.createWriteStream('output.
css.gz'))
readable.pipe(fs.createWriteStream('output.css'))
```

Обратите внимание, что вы должны перенаправлять на выходные потоки синхронно (одновременно) до того, как любые данные начнут «течь». Несоблюдение этого правила может привести к неполному потоку данных.

Также обратите внимание, что объекты потоков могут генерировать события ошибок; убедитесь, что вы ответственно обрабатываете эти события для каждого потока по мере необходимости:

```
var readable = fs.createReadStream('file3.txt')
var writable = fs.createWriteStream('file4.txt')
readable.pipe(writable)
readable.on('error', console.error)
writable.on('error', console.error)
```

Создание собственного читаемого/записываемого потока

Мы часто видим, как объекты потоков возвращаются модулями, такими как fs и т. д., но что если мы хотим создать свой собственный объект потока?

Для создания объекта потока необходимо использовать модуль stream, предоставляемый Node.js:

```
var fs = require("fs");
var stream = require("stream").Writable;

/*
 * Реализация функции записи в классе записываемого потока.
 * Это функция, которая будет использоваться, когда другой поток будет
перенаправлен в этот
 * записываемый поток.
 */

stream.prototype._write = function(chunk, data) {
    console.log(data);
}

var customStream = new stream();
fs.createReadStream("am1.js").pipe(customStream);
```

Это даст нам наш собственный пользовательский записываемый поток. Мы можем реализовать что угодно в пределах функции _write. Вышеописан-

ный метод работает в Node.js версии 4.x.x, но в Node.js 6.x были введены классы ES6, поэтому синтаксис изменился. Ниже приведен код для версии Node.js 6.x:

```
const Writable = require('stream').Writable;
class MyWritable extends Writable {
    constructor(options) {
        super(options);
    }

    _write(chunk, encoding, callback) {
        console.log(chunk);
    }
}
```

Зачем нужны потоки?

Рассмотрим следующие два примера чтения содержимого файла.

Первый пример использует асинхронный метод чтения файла и предоставляет функцию обратного вызова, которая вызывается, как только файл полностью загружается в память:

```
fs.readFile(`${__dirname}/utils.js`, (err, data) => {
    if (err) {
        handleError(err);
    } else {
        console.log(data.toString());
    }
})
```

И второй пример, который использует потоки для чтения содержимого файла по частям:

```
var fileStream = fs.createReadStream(`${__dirname}/file`);
var fileContent = '';

fileStream.on('data', data => {
    fileContent += data.toString();
})

fileStream.on('end', () => {
    console.log(fileContent);
})
fileStream.on('error', err => {
    handleError(err)
})
```

Стоит упомянуть, что оба примера выполняют одно и то же действие. В чем же тогда разница?

- Первый пример короче и выглядит элегантнее.
- Второй позволяет вам обрабатывать файл, пока он читается (!).

Когда файлы, с которыми вы работаете, малы, то использование потоков не оказывает значительного эффекта, но что происходит, когда файл большой (настолько большой, что его чтение в память занимает 10 секунд)?

Без потоков вам придется ждать, ничего не делая (если только ваш процесс не выполняет другие задачи), пока пройдут эти 10 секунд и файл полностью загрузится в память, и только после этого вы сможете начать его обработку.

С потоками вы получаете содержимое файла по частям, как только они становятся доступны, и это позволяет вам обрабатывать файл, пока он читается.

Пример выше не показывает, как потоки могут быть использованы для задач, которые сложно выполнить, используя функции обратного вызова, поэтому давайте рассмотрим другой пример.

Допустим, вам необходимо скачать файл в формате gzip, распаковать его и сохранить содержимое на диск. Имея URL файла, нужно сделать следующее:

- Скачать файл
- Распаковать файл
- Сохранить его на диск

Предположим, что у нас имеется небольшой файл, который хранится в S3-хранилище. Следующий код выполняет описанную выше последовательность действий с использованием функций обратного вызова:

```
var startTime = Date.now()
s3.getObject({Bucket: 'some-bucket', Key: 'tweets.gz'}, (err, data)
=> {
    // здесь загружается весь файл
    zlib.gunzip(data.Body, (err, data) => {
        // здесь весь файл распакован
        fs.writeFile(`${__dirname}/tweets.json`, data, err => {
            if (err) console.error(err)
            // здесь весь файл записан на диск
            var endTime = Date.now()
            console.log(`${endTime - startTime} milliseconds`)
// 1339 milliseconds
        })
    })
})
// 1339 milliseconds
```

Вот как это выглядит при использовании `streams`:

```
s3.getObject({Bucket: 'some-bucket', Key: 'tweets.gz'}).createReadStream()
    .pipe(zlib.createGunzip())
    .pipe(fs.createWriteStream(`${__dirname}/tweets.json`));
// 1204 milliseconds
```

Да, это не быстрее при работе с небольшими файлами — тестируемый файл «весит» 80 КВ. Тестирование на большем файле, 71 MB в формате `gzip` (382 MB в распакованном виде), показывает, что версия с потоками значительно быстрее.

- С использованием функций обратного вызова это заняло 20 925 миллисекунд для загрузки 71 MB, распаковки и записи 382 MB на диск.
- Для сравнения, с использованием потоков это заняло 13 434 миллисекунды (на 35% быстрее).

Глава 108.
Использование WebSocket с Node.js

Примеры

Установка WebSocket

Есть несколько способов установки WebSocket в ваш проект. Вот некоторые примеры:

```
npm install --save ws
```

или внутри вашего package.json, используя:

```
"dependencies": {
 "ws": "*"
}
```

Добавление WebSocket в ваши файлы

Чтобы добавить WebSocket в ваши файлы, просто используйте:

```
var ws = require('ws');
```

Использование WebSocket и WebSocket Server

Чтобы открыть новый WebSocket, просто добавьте такой код:

```
var WebSocket = require("ws");
var ws = new WebSocket("ws://host:8080/OptionalPathName");
// Продолжайте с вашим кодом
```

Или, чтобы открыть сервер, используйте:

```
var WebSocketServer = require("ws").Server;
var ws = new WebSocketServer({port: 8080, path: "OptionalPathName"});
```

Простой пример WebSocket-сервера

```
var WebSocketServer = require('ws').Server,
    wss = new WebSocketServer({ port: 8080 }); // Если хотите добавить
путь, используйте path: "PathName"

wss.on('connection', function connection(ws) {
    ws.on('message', function incoming(message) {
        console.log('received: %s', message);
    });
    ws.send('something');
});
```

Глава 109.
Веб-приложения с Express

Введение

Express — это минималистичный и гибкий фреймворк для веб-приложений на Node.js, предоставляющий мощный набор возможностей для создания веб-приложений.

Официальный сайт Express — https://expressjs.com/.

Исходный код можно найти на GitHub, перейдя по ссылке https://github.com/expressjs/express.

Синтаксис

- `app.get(path [, middleware], callback[, callback...])`
- `app.put(path [, middleware], callback[, callback...])`
- `app.post(path [, middleware], callback[, callback...])`
- `app['delete'](path [, middleware], callback[, callback...])`
- `app.use(path [, middleware], callback[, callback...])`
- `app.use(callback)`

Параметры

Параметр	Детали
path	Указывает часть пути или URL, который будет обрабатываться указанным callback.
middleware	Одна или несколько функций, которые будут вызваны перед callback. Полезно для более специфичной обработки, например, для авторизации или обработки ошибок.
callback	Функция, которая будет использоваться для обработки запросов к указанному пути. Она будет вызываться как `callback (request, response, next)`.
request	Объект, инкапсулирующий детали HTTP-запроса, который необходимо обработать.
response	Объект, который используется для указания того, как сервер должен ответить на запрос.
next	Функция, которая передает управление следующему подходящему маршруту. Принимает необязательный объект ошибки.

Примеры

Начало работы

Сначала вам нужно создать директорию, перейти в нее через оболочку и установить Express с помощью npm, выполнив команду:

```
npm install express -save
```

Создайте файл и назовите его `app.js`, добавив в него следующий код, который создает новый сервер Express и добавляет одну конечную точку (`/ping`) с использованием метода `app.get`:

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

app.get('/ping', (request, response) => {
    response.send('pong');
});
app.listen(8080, 'localhost');
```

Чтобы запустить ваш скрипт, используйте следующую команду в оболочке:

```
> node app.js
```

Ваше приложение будет принимать подключения на `localhost` порта 8080. Если аргумент `hostname` в `app.listen` опущен, то сервер будет принимать подключения как на IP-адрес машины, так и на `localhost`. Если значение порта равно 0, операционная система назначит доступный порт.

После запуска скрипта вы можете протестировать его в оболочке, чтобы убедиться, что вы получаете ожидаемый ответ «pong» от сервера:

```
> curl http://localhost:8080/ping
pong
```

Вы также можете открыть веб-браузер и перейти по URL-адресу http://localhost:8080/ping, чтобы увидеть вывод.

Основное маршрутизирование

Сначала создайте приложение Express:

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

Затем вы можете определить маршруты следующим образом:

```
app.get('/someUri', function (req, res, next) {})
```

Такая структура работает для всех HTTP-методов и ожидает путь в качестве первого аргумента и обработчик для этого пути, который получает объекты `request` и `response`. Таким образом, для основных HTTP-методов маршруты будут следующими:

```
// GET www.domain.com/myPath
app.get('/myPath', function (req, res, next) {})

// POST www.domain.com/myPath
app.post('/myPath', function (req, res, next) {})
```

```
// PUT www.domain.com/myPath
app.put('/myPath', function (req, res, next) {})
// DELETE www.domain.com/myPath
app.delete('/myPath', function (req, res, next) {})
```

Вы можете проверить полный список поддерживаемых глаголов здесь: https://expressjs.com/en/4x/api.html#app.METHOD.

Если вы хотите определить одинаковое поведение для маршрута и всех HTTP-методов, вы можете использовать:

```
app.all('/myPath', function (req, res, next) {})
```

или

```
app.use('/myPath', function (req, res, next) {})
```

или

```
app.use('*', function (req, res, next) {})
// * wildcard будет маршрутизировать для всех путей
```

Вы можете цеплять определения маршрутов для одного пути:

```
app.route('/myPath')
    .get(function (req, res, next) {})
    .post(function (req, res, next) {})
    .put(function (req, res, next) {})
```

Вы также можете добавлять функции к любому HTTP-методу. Они будут выполняться перед финальным `callback` и принимать параметры `req`, `res`, `next` в качестве аргументов.

```
// GET www.domain.com/myPath
app.get('/myPath', myFunction, function (req, res, next) {})
```

Ваши финальные callback-функции могут быть сохранены во внешнем файле, чтобы избежать перенасыщения кода в одном файле:

```
// other.js
exports.doSomething = function(req, res, next) {/* выполняем что-то
*/};
```

...а затем в файле, содержащем ваши маршруты:

```
const other = require('./other.js');
app.get('/someUri', myFunction, other.doSomething);
```

Это сделает ваш код гораздо более удобным для чтения.

Получение информации из запроса

Допустим, вам необходимо получить информацию из URL-запроса (обратите внимание, что `req` — это объект запроса в обработчике маршрутов). Рассмотрим это определение маршрута `/settings/:user_id` и следующий пример: `/settings/32135?field=name`:

```
// получение полного пути
req.originalUrl // => /settings/32135?field=name

// получение параметра user_id
req.params.user_id // => 32135

// получение значения query параметра field
req.query.field // => 'name'
```

Вы также можете получить заголовки запроса следующим образом:

```
req.get('Content-Type')
// "text/plain"
```

Для упрощения получения другой информации можно использовать `middlewares` (промежуточные обработчики). Например, чтобы получить информацию из тела запроса, вы можете использовать `middleware body-parser`, который преобразует «сырое» тело запроса в удобный формат:

```
var app = require('express')();
var bodyParser = require('body-parser');

app.use(bodyParser.json()); // для парсинга application/json
app.use(bodyParser.urlencoded({ extended: true })); // для парсинга
application/x-www-form-urlencoded
```

Теперь предположим, что запрос выглядит так:

```
PUT /settings/32135
{
 "name": "Peter"
}
```

Вы можете получить доступ к переданному имени следующим образом:

```
req.body.name
// "Peter"
```

Аналогично вы можете получить доступ к cookies из запроса. Для этого также нужен `middleware`, такой как `cookie-parser`:

```
req.cookies.name
```

Модульное приложение на Express

Чтобы сделать веб-приложение на Express модульным, используйте фабрики роутеров

Модуль:

```
// greet.js

const express = require('express');

module.exports = function(options = {}) { // Фабрика роутера
    const router = express.Router();
    router.get('/greet', (req, res, next) => {
        res.end(options.greeting);
    });
    return router;
};
```

Приложение:

```
// app.js
const express = require('express');
const greetMiddleware = require('./greet.js');
express()
    .use('/api/v1/', greetMiddleware({ greeting:'Hello world' }))
    .listen(8080);
```

Это сделает ваше приложение модульным, настраиваемым, а ваш код — переиспользуемым.

При обращении по адресу `http://<hostname>:8080/api/v1/greet` вывод будет «`Hello world`».

Более сложный пример

Рассмотрим пример с сервисами, который демонстрирует преимущества middleware-фабрики.

Модуль:

```
// greet.js
const express = require('express');
module.exports = function(options = {}) { // Фабрика роутера
    const router = express.Router();
    // Получаем контроллер
    const {service} = options;
    router.get('/greet', (req, res, next) => {
        res.end(
            service.createGreeting(req.query.name || 'Stranger')
        );
    });
    return router;
};
```

Приложение:

```
// app.js
const express = require('express');
const greetMiddleware = require('./greet.js');
class GreetingService {
    constructor(greeting = 'Hello') {
        this.greeting = greeting;
    }
    createGreeting(name) {
        return `${this.greeting}, ${name}!`;
    }
}

express()
    .use('/api/v1/service1', greetMiddleware({
        service: new GreetingService('Hello'),
    }))
    .use('/api/v1/service2', greetMiddleware({
        service: new GreetingService('Hi'),
    }))
    .listen(8080);
```

При обращении по адресу `http://<hostname>:8080/api/v1/service1/greet?name=World` вывод будет «Hello, World», а при обращении по адресу `http://<hostname>:8080/api/v1/service2/greet?name=World` вывод будет «Hi, World».

Использование шаблонизатора

Следующий код настроит Jade в качестве шаблонизатора (**Примечание:** с декабря 2015 года Jade был переименован в Pug).

```
const express = require('express'); // Импортируем модуль express
const app = express(); // Создаем экземпляр модуля express
const PORT = 3000; // Случайно выбранный порт

app.set('view engine', 'jade'); // Устанавливаем jade в качестве шаблонизатора
app.set('views', 'src/views'); // Устанавливаем каталог, где хранятся все представления (.jade-файлы)

// Создаем корневой маршрут
app.get('/', function(req, res) {
    res.render('index'); // рендерим файл index.jade в html и возвращаем как ответ.
    // Функция render опционально принимает данные для передачи в представление.
});
// Запускаем сервер Express с обратным вызовом
app.listen(PORT, function(err) {
    if (!err) {
        console.log('Server is running at port', PORT);
    } else {
        console.log(JSON.stringify(err));
    }
});
```

Аналогичным образом могут использоваться и другие шаблонизаторы, такие как `Handlebars (hbs)` или EJS. Не забудьте установить соответствующий пакет с помощью `npm`. Для `Handlebars` используйте пакет `hbs`, для `Jade` — пакет `jade`, а для EJS — пакет `ejs`.

Пример с шаблоном EJS

С помощью EJS (как и других шаблонизаторов Express) вы можете выполнять серверный код и получать доступ к переменным сервера из HTML.

В EJS это делается с использованием «<%» в качестве начального тега и «%>» в качестве конечного тега, переменные, переданные как параметры рендера, можно получить с помощью <%=var_name%>.

Например, если у вас есть массив supplies в серверном коде, вы можете итерировать по нему следующим образом:

```
<h1><%= title %></h1>

<ul>
    <% for(var i = 0; i < supplies.length; i++) { %>
        <li>
            <a href='supplies/<%= supplies[i] %>'>
                <%= supplies[i] %>
            </a>
        </li>
    <% } %>
</ul>
```

Как видно из примера, каждый раз, когда вы переключаетесь между серверным кодом и HTML, вам нужно закрыть текущий тег EJS и позже открыть новый. В этом примере мы хотели создать li внутри цикла for, поэтому нам нужно было закрыть тег EJS в конце for и создать новый тег только для фигурных скобок.

Еще один пример: если мы хотим установить значение по умолчанию для элемента input в качестве переменной с сервера, мы используем <%=:

```
Message:<br>
<input type="text" value="<%= message %>" name="message" required>
```

Здесь переменная message, переданная с сервера, будет значением по умолчанию для вашего input. Обратите внимание, что если вы не передадите переменную message с сервера, EJS вызовет исключение. Вы можете передать параметры с помощью res.render('index', {message: message}); (для файла EJS под названием index.ejs).

В тегах EJS вы также можете использовать if, while или любую другую команду JavaScript.

JSON API с Express.js

```
var express = require('express');
var cors = require('cors'); // Используем модуль cors для включения
Cross-origin resource sharing
var app = express();
app.use(cors()); // для всех маршрутов
```

```
var port = process.env.PORT || 8080;
app.get('/', function(req, res) {
    var info = {
        'string_value': 'StackOverflow',
        'number_value': 8476
    }
    res.json(info);
    // или
    /* res.send(JSON.stringify({
        string_value: 'StackOverflow',
        number_value: 8476
    })) */
    // можно добавить статус-код к json-ответу
    /* res.status(200).json(info) */
});

app.listen(port, function() {
    console.log('Node.js listening on port ' + port);
});
```

При обращении по адресу http://localhost:8080/ выводится следующий объект:

```
{
    "string_value": "StackOverflow",
    "number_value": 8476
}
```

Обслуживание статических файлов

При создании веб-сервера с помощью Express часто требуется обслуживать сочетание динамического контента и статических файлов.

Например, у вас могут быть файлы `index.html` и `script.js`, которые являются статическими и хранятся в файловой системе. Обычно для хранения статических файлов используется папка с именем «public». В этом случае структура папок может выглядеть следующим образом:

```
project root
├— server.js
├— package.json
└— public
    ├— index.html
    └— script.js
```

Настроить Express для обслуживания статических файлов можно следующим образом:

```
const express = require('express');
const app = express();
app.use(express.static('public'));
```

Примечание: после настройки папки `index.html` `script.js` и все файлы в папке `public` будут доступны по корневому пути (не нужно указывать `/public/` в URL). Это связано с тем, что Express ищет файлы относительно настроенной статической папки. Вы можете указать виртуальный префикс пути, как показано ниже:

```
app.use('/static', express.static('public'));
```

Это сделает ресурсы доступными по префиксу `/static/`.

Несколько папок

Можно определить несколько папок одновременно:

```
app.use(express.static('public'));
app.use(express.static('images'));
app.use(express.static('files'));
```

При обслуживании ресурсов Express будет просматривать папки в порядке их определения. В случае наличия файлов с одинаковыми именами будет обслуживаться файл из первой совпавшей папки.

Именованные маршруты в стиле Django

Одной из основных проблем является то, что поддержка именованных маршрутов не реализована в Express «из коробки». Решением является установка стороннего пакета, например `express-reverse`:

```
npm install express-reverse
```

Подключите его в ваш проект:

```
var app = require('express')();

require('express-reverse')(app);
```

Затем используйте его следующим образом:

```
app.get('test', '/hello', function(req, res) {
    res.end('hello');
});
```

Недостатком этого подхода является то, что вы не можете использовать модуль маршрутов Express. Обходным решением может стать передача вашего приложения в качестве параметра фабрике роутера:

```
require('./middlewares/routing')(app);
```

...и использование его следующим образом:

```
module.exports = (app) => {
    app.get('test', '/hello', function(req, res) {
        res.end('hello');
    });
};
```

Дальше вы сможете самостоятельно разобраться, как определять функции для их объединения с заданными пользовательскими пространствами имен и указывать на соответствующие контроллеры.

Обработка ошибок

Основная обработка ошибок

По умолчанию Express будет искать представление 'error' в директории /views для рендеринга. Просто создайте представление 'error' и поместите его в директорию views для обработки ошибок. Ошибки записываются с сообщением об ошибке, статусом и трассировкой стека, например:

```
views/error.pug

html
    body

        h1= message
        h2= error.status
        p= error.stack
```

Расширенная обработка ошибок

Определите свои функции промежуточной обработки ошибок в самом конце стека функций промежуточной обработки. Эти функции принимают четыре аргумента вместо трех (err, req, res, next), например:

```
app.js

// Обработка ошибки 404 и передача ее обработчику ошибок
app.use(function(req, res, next) {
    var err = new Error('Not Found');
```

```
    err.status = 404;
    // передаем ошибку на следующий соответствующий маршрут
    next(err);
});
// Обработка ошибок и вывод трассировки стека
app.use(function(err, req, res, next) {
    res.status(err.status || 500);
    res.render('error', {
        message: err.message,
        error: err
    });
});
```

Вы можете определить несколько функций промежуточной обработки ошибок, так же, как и с обычными функциями промежуточной обработки.

Использование промежуточного ПО и обратного вызова next

Express передает обратный вызов `next` каждому обработчику маршрутов и функции промежуточной обработки, который может быть использован для разделения логики одного маршрута на несколько обработчиков. Вызов `next()` без аргументов указывает Express продолжить выполнение следующей соответствующей функции промежуточной обработки или обработчика маршрутов. Вызов `next(err)` с ошибкой приведет к запуску любого промежуточного ПО для обработки ошибок. Вызов `next('route')` пропустит любое последующее промежуточное ПО на текущем маршруте и перейдет к следующему соответствующему маршруту. Это позволяет разделить доменную логику на переиспользуемые компоненты, которые являются автономными, проще тестируются и легче поддерживаются и изменяются.

Множественные соответствующие маршруты

Запросы к `/api/foo` или `/api/bar` будут запускать начальный обработчик для поиска участника, а затем передавать управление фактическому обработчику для каждого маршрута:

```
app.get('/api', function(req, res, next) {
    // И /api/foo, и /api/bar будут запускать это
    lookupMember(function(err, member) {
        if (err) return next(err);
        req.member = member;
        next();
    });
});
```

```
app.get('/api/foo', function(req, res, next) {
    // Только /api/foo будет запускать это
    doSomethingWithMember(req.member);
});
app.get('/api/bar', function(req, res, next) {
    // Только /api/bar будет запускать это
    doSomethingDifferentWithMember(req.member);
});
```

Обработчик ошибок

Обработчики ошибок являются функциями промежуточной обработки с сигнатурой `function(err, req, res, next)`. Они могут быть установлены для конкретного маршрута (например, `app.get('/foo', function(err, req, res, next))`, но, как правило, достаточно одного обработчика ошибок, который рендерит страницу ошибки:

```
app.get('/foo', function(req, res, next) {
    doSomethingAsync(function(err, data) {
        if (err) return next(err);
        renderPage(data);
    });
});
// В случае если doSomethingAsync возвращает ошибку, этот специальный
// обработчик ошибок промежуточного ПО будет вызван с ошибкой в каче-
стве
// первого параметра.
app.use(function(err, req, res, next) {
    renderErrorPage(err);
});
```

Промежуточное ПО

Каждая из вышеуказанных функций фактически является функцией промежуточного ПО, которая запускается всякий раз, когда запрос соответствует определенному маршруту, но любое количество функций промежуточного ПО может быть определено на одном маршруте. Это позволяет определить промежуточное ПО в отдельных файлах и использовать общую логику повторно в нескольких маршрутах:

```
app.get('/bananas', function(req, res, next) {
    getMember(function(err, member) {
        if (err) return next(err);
        // Если участника нет, не пытайтесь искать данные.
        // Просто перейдите к рендерингу страницы.
        if (!member) return next('route');
       // В противном случае вызовите следующее промежуточное ПО и по-
лучите
```

```
        // данные участника.
        req.member = member;
        next();
    });
}, function(req, res, next) {
    getMemberData(req.member, function(err, data) {
        if (err) return next(err);
        // Если у этого участника нет данных, не утруждайтесь
        // их разбором. Просто перейдите к рендерингу страницы.
        if (!data) return next('route');
       // В противном случае вызовите следующее промежуточное ПО и раз-
берите
        // данные участника. Затем отрендерьте страницу.
        req.member.data = data;
        next();
    });
}, function(req, res, next) {
    req.member.parsedData = parseMemberData(req.member.data);
    next();
});

app.get('/bananas', function(req, res, next) {
    renderBananas(req.member);
});
```

В этом примере каждая функция промежуточного ПО будет находиться либо в своем собственном файле, либо в переменной в другом месте файла, чтобы ее можно было использовать повторно в других маршрутах.

Обработка ошибок

Основную документацию можно найти здесь: https://expressjs.com/en/guide/error-handling.html.

```
app.get('/path/:id(\\d+)', function (req, res, next) { // обратите
внимание: "next" передается
    if (req.params.id == 0) // проверка параметра
        return next(new Error('Id is 0')); // перейти к первому обра-
ботчику ошибок, см. ниже
    // Обработка ошибок при синхронной операции
    var data;
    try {
        data = JSON.parse('/file.json');
    } catch (err) {
        return next(err);
    }

    // Если возникла критическая ошибка, остановите приложение
    if (!data)

        throw new Error('Smth wrong');
     // Если необходимо передать дополнительную информацию в обработ-
чик ошибок,
    // отправьте пользовательскую ошибку (см. Приложение B)

    if (smth)
        next(new MyError('smth wrong', arg1, arg2));

    // Завершите запрос с помощью res.render или res.end
    res.status(200).end('OK');
});
```

Будьте уверены: порядок использования app.use имеет значение.

```
// Обработчик ошибок
app.use(function(err, req, res, next) {

    if (smth-check, e.g. req.url != 'POST')
        return next(err); // перейти к обработчику ошибок 2.
    console.log(req.url, err.message);

     if (req.xhr) // если запрос через ajax, то отправить json, иначе
отрендерьте страницу ошибки
        res.json(err);

    else
        res.render('error.html', {error: err.message});
});
```

```
// Обработчик ошибок 2
app.use(function(err, req, res, next) {
      // выполните что-нибудь здесь, например, проверьте, является
ли ошибка MyError

    if (err instanceof MyError) {
        console.log(err.message, err.arg1, err.arg2);
    }
    ...

    res.end();
});
```

Приложение A

```
// "В Express ответы с кодом 404 не являются результатом ошибки,
// поэтому промежуточное ПО для обработки ошибок их не захватит."
// Вы можете изменить это.
app.use(function(req, res, next) {
    next(new Error(404));
});
```

Приложение B

```
// Как определить пользовательскую ошибку
var util = require('util');
...

function MyError(message, arg1, arg2) {
    this.message = message;
    this.arg1 = arg1;
    this.arg2 = arg2;
    Error.captureStackTrace(this, MyError);
}

util.inherits(MyError, Error);
MyError.prototype.name = 'MyError';
```

Как выполнить код перед любым запросом и после любого ответа

app.use() и промежуточное ПО (middleware) могут использоваться для выполнения кода «перед», а комбинация событий `close` и `finish` может использоваться для выполнения кода «после».

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

```
        res.removeListener('finish', afterResponse);
        res.removeListener('close', afterResponse);
        // действия после ответа
    }

    res.on('finish', afterResponse);
    res.on('close', afterResponse);
    // действия перед запросом
    // в конце вызов `next()`
    next();
});

app.use(app.router);
```

Примером этого является промежуточное ПО logger, которое по умолчанию добавляет запись в лог после ответа.

Просто убедитесь, что это промежуточное ПО используется перед `app.router`, так как порядок имеет значение.

Оригинальный пост находится здесь: https://stackoverflow.com/questions/20175806/before-and-after-hooks-for-a-request-in-express-to-be-executed-before-any-req-a.

Обработка POST-запросов

Точно так же, как вы обрабатываете GET-запросы в Express с помощью метода `app.get`, вы можете использовать метод `app.post` для обработки POST-запросов.

Но прежде чем вы сможете обрабатывать POST-запросы, вам нужно будет использовать промежуточное ПО `body-parser`. Оно просто разбирает тело POST, PUT, DELETE и других запросов.

Промежуточное ПО `body-parser` разбирает тело запроса и превращает его в объект, доступный в `req.body`:

```
var bodyParser = require('body-parser');

const express = require('express');
const app = express();

// Разбирает тело для POST, PUT, DELETE и т.д.
app.use(bodyParser.json());

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

app.post('/post-data-here', function(req, res, next){
    console.log(req.body); // req.body содержит разобранное тело за-
проса.
});

app.listen(8080, 'localhost');
```

Установка cookies с использованием cookie-parser

Ниже приведен пример установки и чтения cookies с использованием модуля cookie-parser:

```
var express = require('express');

var cookieParser = require('cookie-parser'); // модуль для разбора
cookies
var app = express();

app.use(cookieParser());

app.get('/setcookie', function(req, res) {
    // установка cookies
   res.cookie('username', 'john doe', { maxAge: 900000, httpOnly: true
});
    return res.send('Cookie has been set');
});

app.get('/getcookie', function(req, res) {
    var username = req.cookies['username'];

    if (username) {
        return res.send(username);
    }
```

```
    return res.send('No cookie found');
});

app.listen(3000);
```

Пользовательское middleware (промежуточное ПО) в Express

В Express вы можете определять middleware, которое может использоваться для проверки запросов или установки некоторых заголовков в ответе:

```
app.use(function(req, res, next){ }); // сигнатура
```

Пример

Следующий код добавляет пользователя в объект запроса и передает управление следующему соответствующему маршруту:

```
var express = require('express');
var app = express();

// каждый запрос пройдет через это middleware
app.use(function(req, res, next){
    req.user = 'testuser';

   next(); // передаст управление следующему соответствующему маршруту
});
app.get('/', function(req, res){
    var user = req.user;
    console.log(user); // testuser
    return res.send(user);
});

app.listen(3000);
```

Обработка ошибок в Express

В Express вы можете определить единый обработчик ошибок для обработки ошибок, возникших в приложении. Определите этот обработчик в конце всех маршрутов и логического кода.

Пример

```
var express = require('express');

var app = express();
// GET /names/john
```

```
app.get('/names/:name', function(req, res, next) {
    if (req.params.name == 'john'){
        return res.send('Valid Name');

    } else{
         next(new Error('Not valid name')); // передать в обработчик
ошибок
    }

});
// обработчик ошибок
app.use(function(err, req, res, next) {

    console.log(err.stack); // например, Not valid name
    return res.status(500).send('Internal Server Occured');
});

app.listen(3000);
```

Добавление промежуточного ПО (middleware)

Функции middleware — это функции, которые имеют доступ к объекту запроса (`req`), объекту ответа (`res`) и следующей функции middleware в цикле «запрос-ответ» приложения. Функции middleware могут выполнять любой код, изменять объекты `res` и `req`, завершать цикл ответа и вызывать следующее middleware.

Распространенным примером middleware является модуль `cors`. Чтобы добавить поддержку CORS, просто установите его, подключите и добавьте эту строку:

```
app.use(cors());
```

перед любыми роутерами или функциями маршрутизации.

Hello World

Здесь мы создаем простой сервер `Hello World` с использованием Express. Маршруты:

- `'/'`
- `'/wiki'`

Для остальных маршрутов будет возвращено «404», т.е. «страница не найдена».

```
'use strict';
const port = process.env.PORT || 3000;
var app = require('express')();

app.listen(port);

app.get('/', (req, res) => res.send('HelloWorld!'));
app.get('/wiki', (req, res) => res.send('This is wiki page.'));
app.use((req, res) => res.send('404-PageNotFound'));
```

Примечание: мы разместили маршрут 404 в качестве последнего, так как Express обрабатывает маршруты по порядку и последовательно обрабатывает их для каждого запроса.

Глава 110.
Windows-аутентификация в Node.js

Примечания

Существуют и другие API для Active Directory, такие как `activedirectory2` и `adldap`.

Примеры

Использование activedirectory

Пример ниже взят из полной документации, доступной на GitHub: https://github.com/gheeres/node-activedirectory

...или NPM: https://npmdoc.github.io/node-npmdoc-activedirectory/build/apidoc.html.

Установка

```
npm install --save activedirectory
```

Использование

```
// Инициализация
var ActiveDirectory = require('activedirectory');
var config = {
    url: 'ldap://dc.domain.com',
    baseDN: 'dc=domain,dc=com'
};
var ad = new ActiveDirectory(config);
var username = 'john.smith@domain.com';
var password = 'password';

// Аутентификация
ad.authenticate(username, password, function(err, auth) {
    if (err) {
        console.log('ERROR: '+JSON.stringify(err));
        return;
    }
    if (auth) {
        console.log('Authenticated!');
    }
    else {
        console.log('Authentication failed!');
    }
});
```

Глава 111.
Менеджер пакетов Yarn

Введение

Yarn — это менеджер пакетов для Node.js, аналогичный NPM. Несмотря на сходство, между Yarn и NPM есть некоторые ключевые различия.

Установка Yarn

В этом примере объясняются различные методы установки Yarn для вашей операционной системы.

macOS

Homebrew

```
brew update
brew install yarn
```

MacPorts

```
sudo port install yarn
```

Добавление Yarn в PATH

Добавьте следующую строку в профиль вашей оболочки (например, `.profile`, `.bashrc`, `.zshrc` и т. д.):

```
export PATH="$PATH:`yarn global bin`"
```

Windows

Установщик

Сначала установите Node.js, если он еще не установлен. Затем скачайте установщик Yarn в формате .msi с сайта Yarn: https://classic.yarnpkg.com/en/docs/install#windows-stable.

Chocolatey

```
choco install yarn
```

Linux

Debian / Ubuntu

Убедитесь, что Node.js установлен для вашей дистрибуции, или выполните следующую команду:

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
sudo apt-get install -y nodejs
```

Настройка репозитория YarnPkg:

```
curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -
echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee /
etc/apt/sources.list.d/yarn.list
```

Установка Yarn:

```
sudo apt-get update && sudo apt-get install yarn
```

CentOS / Fedora / RHEL

Установите Node.js, если он еще не установлен:

```
curl --silent --location https://rpm.nodesource.com/setup_6.x | bash -
```

Установка Yarn:

```
sudo wget https://dl.yarnpkg.com/rpm/yarn.repo -O /etc/yum.repos.d/
yarn.repo
sudo yum install yarn
```

Arch

Установите Yarn через AUR. Пример с использованием `yaourt`:

```
yaourt -S yarn
```

Solus

```
sudo eopkg install yarn
```

Все дистрибутивы

Добавьте следующую строку в профиль вашей оболочки (например, `.profile`, `.bashrc`, `.zshrc` и т. д.):

```
export PATH="$PATH:`yarn global bin`"
```

Альтернативный метод установки

Скрипт оболочки

```
curl -o- -L https://yarnpkg.com/install.sh | bash
```

или укажите версию для установки:

```
curl -o- -L https://yarnpkg.com/install.sh | bash -s -- --version
[version]
```

Tarball

```
cd /opt
wget https://yarnpkg.com/latest.tar.gz
tar zvxf latest.tar.gz
```

NPM

Если у вас уже установлен NPM, просто выполните:

```
npm install -g yarn
```

После установки

Проверьте установленную версию Yarn, выполнив:

```
yarn –version
```

Создание базового пакета

Команда `yarn init` проведет вас через процесс создания файла `package.json`, чтобы настроить информацию о вашем пакете. Это похоже на команду `npm init` в NPM.

Создайте новую директорию для вашего пакета и перейдите в нее, затем выполните `yarn init`:

```
mkdir my-package && cd my-package
yarn init
```

Ответьте на вопросы в интерфейсе командной строки:

```
question name (my-package): my-package
question version (1.0.0):
question description: A test package
question entry point (index.js):
question repository url:
question author: StackOverflow Documentation
question license (MIT):
success Saved package.json
Done in 27.31s.
```

Это создаст файл `package.json`, похожий на следующий:

```
{
    "name": "my-package",
    "version": "1.0.0",
    "description": "A test package",
    "main": "index.js",
    "author": "StackOverflow Documentation",
    "license": "MIT"
}
```

Теперь попробуем добавить зависимость. Базовый синтаксис для этого — `yarn add [package-name]`.

Для начала добавим ExpressJS.

Выполните для этого следующую команду:

```
yarn add express
```

При помощи этого вы добавите раздел `dependencies` в ваш `package.json` и добавите ExpressJS:

```
"dependencies": {
    "express": "^4.15.2"
}
```

Установка пакета с помощью Yarn

Yarn использует тот же реестр, что и NPM. Это означает, что каждый пакет, доступный в NPM, доступен и в Yarn.

Чтобы установить пакет, выполните:

```
yarn add package
```

Если вам нужна конкретная версия пакета, используйте:

```
yarn add package@version
```

Если версия, которую вы хотите установить, была помечена тегом, используйте:

```
yarn add package@tag.
```

Благодарности

№	Название главы	Участники
1	Начало работы с Node.js	4444, Abdelaziz Mokhnache, Abhishek Jain, Adam, Aeolingamenfel, Alessandro Trinca Tornidor, Aljoscha Meyer, Amila Sampath, Ankit Gomkale, Ankur Anand, ...
2	Взаимодействие Arduino с Node.js	sBanda
3	async.js	David Knipe, devnull69, DrakaSAN, F. Kauder, jerry, lsampaio, Shriganesh Kolhe, Sky, walid
4	Async/Await	Cami Rodriguez, Cody G., cyanbeam, Dave, David Xu, Dom Vinyard, m_callens, Manuel, nomanbinhussein, Toni Villena
5	Асинхронное программирование	Ala Eddine JEBALI, cyanbeam, Florian Hämmerle, H. Pauwelyn, John, Marek Skiba, Native Coder, omgimanerd, slowdeath007
6	Автоматическая перезагрузка при изменениях	ch4nd4n, Dean Rather, Jonas S, Joshua Kleveter, Nivesh, Sanketh Katta, zurfyx
7	Избегайте ада функций обратного вызова (callback hell)	tyehia
8	Bluebird Promises (промисы)	David Xu

9	Преобразование функций обратного вызова	Clement JACOB, Michael Buen, Sanketh Katta
10	Интеграция с Cassandra	Vsevolod Goloviznin
11	Интерфейс командной строки (CLI)	Ze Rubeus
12	Взаимодействие клиент-сервер	Zoltán Schmidt
13	Модуль Cluster	Benjamin, Florian Hämmerle, Kid Binary, MayorMonty, Mukesh Sharma, riyadhalnur, Vsevolod Goloviznin
14	Подключение к MongoDB	FabianCook, Nainesh Raval, Shriganesh Kolhe
15	Создание библиотеки Node.js, поддерживающей как Promises (промисы), так и функции обратного вызова с первым аргументом-ошибкой	Dave
16	Создание API с помощью Node.js	Mukesh Sharma
17	Парсер CSV в Node.js	aisflat439
18	Работа с базой данных (MongoDB с Mongoose)	zurfyx
19	Отладка Node.js приложения	4444, Alister Norris, Ankur Anand, H. Pauwelyn, Matthew Shanley
20	Доставка HTML или других типов файлов	Himani Agrawal, RamenChef, user2314737
21	Внедрение зависимостей	Niroshan Ranapathi
22	Развертывание приложения Node.js без простоя	gentlejo

23	Развертывание приложений Node.js в продакшене	Apidcloud, Brett Jackson, Community, Cristian Boariu, duncanhall, Florian Hämmerle, guleria, haykam, KlwntSingh, Mad Scientist, MatthieuLemoine, Mukesh Sharma, ...
24	ECMAScript 2015 (ES6) с Node.js	David Xu, Florian Hämmerle, Osama Bari
25	Окружение	Chris, Freddie Coleman, KlwntSingh, Louis Barranqueiro, Mikhail, sBanda
26	Источники событий	DrakaSAN, Duly Kinsky, Florian Hämmerle, jamescostian, MindlessRanger, Mothman
27	Eventloop (цикл событий)	Kelum Senanayake
28	Обработка исключений	KlwntSingh, Nivesh, riyadhalnur, sBanda, sjmarshy, topheman
29	Выполнение файлов или команд с помощью дочерних процессов	guleria, hexacyanide, iSkore
30	Экспорт и использование модулей	Aminadav, Craig Ayre, cyanbeam, devnull69, DrakaSAN, Fenton, Florian Hämmerle, hexacyanide, Jason, jdrydn, Loufylouf, Louis Barranqueiro, ...
31	Экспорт и импорт модулей в Node.js	AndrewLeonardi, Bharat, commonSenseCode, James Billingham, Oliver, sharif.io, Shog9
32	Загрузка файлов	Aikon Mogwai, Iceman, Mikhail, walid
33	Файловый ввод/вывод (Filesystem I/O)	4444, Accepted Answer, Aeolingamenfel, Christophe Marois, Craig Ayre, DrakaSAN, Duly Kinsky, Florian Hämmerle, gnerkus, Harshal Bhamare, hexacyanide, ...
34	Начало работы с профилированием Node.js	damitj07

35	Хороший стиль кодирования	Ajitej Kaushik, RamenChef
36	Graceful Shutdown	RamenChef, Sathish
37	grunt	Naeem Shaikh, Waterscroll
38	Hack	—
39	Обработка POST-запросов в Node.js	Manas Jayanth
40	Как загружаются модули	RamenChef, umesh
41	HTTP	Ahmed Metwally, Alister Norris, Aminadav, Anh Cao, asherbar, Batsu, Buzut, Chance Snow, Chezzwizz, Dmitriy Borisov, Florian Hämmerle, GilZ, guleria, ...
42	Установка Node.js	Hashim, Siddharth Srivastva, Sveratum, tandrewnichols, user2314737, user6939352, V1P3R, victorkohl
43	Взаимодействие с консолью	ScientiaEtVeritas
44	Постоянное выполнение Node.js приложения	Alex Logan, Bearington, cyanbeam, Himani Agrawal, Mikhail, mscdex, optimus, pietrovismara, RamenChef, Sameer Srivastava, somebody, Taylor Swanson
45	Фреймворк Koa v2	David Xu
46	Lodash	M1kstur
47	Loopback — REST соединитель	Roopesh
48	metalsmith	RamenChef, vsjn3290ckjnaoij2jikndckjb
49	Интеграция с MongoDB	cyanbeam, FabianCook, midnightsyntax

50	Интеграция MongoDB для Node.js/Express.js	William Carron
51	Библиотека Mongoose	Alex Logan, manuerumx, Mikhail, Naeem Shaikh, Qiong Wu, Simplans, Will
52	Интеграция MSSQL	damitj07
53	Многопоточность	arcs
54	Пул соединений MySQL	KlwntSingh
55	Интеграция MySQL	Aminadav, Andrés Encarnación, Florian Hämmerle, Ivan Schwarz, jdrydn, JohnnyCoder, Kapil Vats, KlwntSingh, Marek Skiba, Rafael Gadotti Bachovas, ...
56	N-API	Parham Alvani
57	Локализация Node.js	Osama Bari
58	Сервер Node без фреймворка	Hasan A Yousef, Taylor Ackley
59	Node.js (express.js) с Angular.js	sigfried
60	Node.js и MongoDB	midnightsyntax, RamenChef, Satyam S
61	Архитектура и внутренняя работа Node.js	Ivan Hristov
62	Код Node.js для STDIN и STDOUT без использования библиотек	Syam Pradeep
63	Основы проектирования Node.js	Ankur Anand, pietrovismara
64	Управление ошибками в Node.js	Karlen
65	Производительность Node.js	Florian Hämmerle, Inanc Gumus

66	Новые функции и улучшения Node.js v6	creyD, DominicValenciana, KlwntSingh
67	Node.js с CORS	Buzut
68	Node.js с ES6	Inanc Gumus, xam, ymz, zurfyx
69	Node.js с Oracle	oliolioli
70	Руководство для начинающих по Node.js	Niroshan Ranapathi
71	Фреймворки Node.js	dthree
72	История Node.js	Kelum Senanayake
73	Маршрутизация в Node.js	parlad neupane
74	Node.js с Redis	evalsocket, Abhishek Jain, AJS, Amreesh Tyagi, Ankur Anand, Asaf Manassen, Ates Goral, ccnokes, CD.., Cristian Cavalli, David G., DrakaSAN, Eric Fortin, ...
75	npm	Sorangwala Abbasali, still_learning, subbu, the12, tlo, Un3qual, uzaif, VladNeacsu, Vsevolod Goloviznin, Wasabi Fan, Yerko Palma
76	nvm — Node Version Manager	cyanbeam, guleria, John Vincent Jardin, Luis González, pranspach, Shog9, Tushar Gupta
77	OAuth 2.0	tyehia
78	package.json	Ankur Anand, Asaf Manassen, Chance Snow, efeder, Eric Smekens, Florian Hämmerle, Jaylem Chaudhari, Kornel, lauriys, mezzode, OzW, RamenChef, Robbie, ...
79	Разбор аргументов командной строки	yrtimiD
80	Интеграция Passport	Ankit Rana, Community, Léo Martin, M. A. Cordeiro, Rupali Pemare, shikhar bansal
81	passport.js	Red

82	Проблемы производительности	Antenka, SteveLacy
83	Интеграция с PostgreSQL	Niroshan Ranapathi
84	Структура проекта	damitj07
85	Push-уведомления	Mario Rozic
86	Readline	4444, Craig Ayre, Florian Hämmerle, peteb
87	Удаленная отладка в Node.js	Rick, VooVoo
88	Require()	Philip Cornelius Glover
89	Дизайн RESTful API: Лучшие практики	fresh5447, nilakantha singh deo
90	Структура Route-Controller-Service для Express.js	nomanbinhussein
91	Маршрутизация ajax-запросов с помощью Express.js	RamenChef, SynapseTech
92	Запуск Node.js как сервиса	Buzut
93	Обеспечение безопасности приложений Node.js	akinjide, devnull69, Florian Hämmerle, John Slegers, Mukesh Sharma, Pauly Garcia, Peter G, pranspach, RamenChef, Simplans
94	Отправка веб-уведомлений	Houssem Yahiaoui
95	Отправка файлового потока клиенту	Beshoy Hanna
96	Sequelize.js	Fikra, Niroshan Ranapathi, xam
97	Простой CRUD API на основе REST	Iceman
98	Связь через Socket.io	Forivin, N.J.Dawson
99	Синхронное и асинхронное программирование в Node.js	Craig Ayre, Veger
100	TCP-сокеты	B Thuy

101	Шаблонные фреймворки	Aikon Mogwai
102	Удаление Node.js	John Vincent Jardin, RamenChef, snuggles08, Trevor Clarke
103	Фреймворки для модульного тестирования	David Xu, Florian Hämmerle, skiilaa
104	Сценарии использования Node.js	vintproykt
105	Использование Browserify для устранения ошибки ‘required’ в браузерах	Big Dude
106	Использование IISNode для размещения веб-приложений Node.js в IIS	peteb
107	Использование потоков	cyanbeam, Duly Kinsky, efeder, johni, KlwntSingh, Max, Ze Rubeus
108	Использование WebSocket с Node.js	Rowan Harley
109	Веб-приложения с Express	Aikon Mogwai, Alex Logan, alexi2, Andres C. Viesca, Aph, Asaf Manassen, Batsu, bekce, brianmearns, Community, Craig Ayre, Daniel Verem, devnull69, Everettss, ...
110	Аутентификация Windows в Node.js	CJ Harries
111	Менеджер пакетов Yarn	Andrew Brooke, skiilaa

16+

Справочное издание
Анықтамалық басылым

Серия «Быстрый старт в программирование»

NODE.JS
САМОЕ ПОЛНОЕ РУКОВОДСТВО ДЛЯ ВЕБ-РАЗРАБОТЧИКОВ В ПРИМЕРАХ ОТ СООБЩЕСТВА STACK OVERFLOW

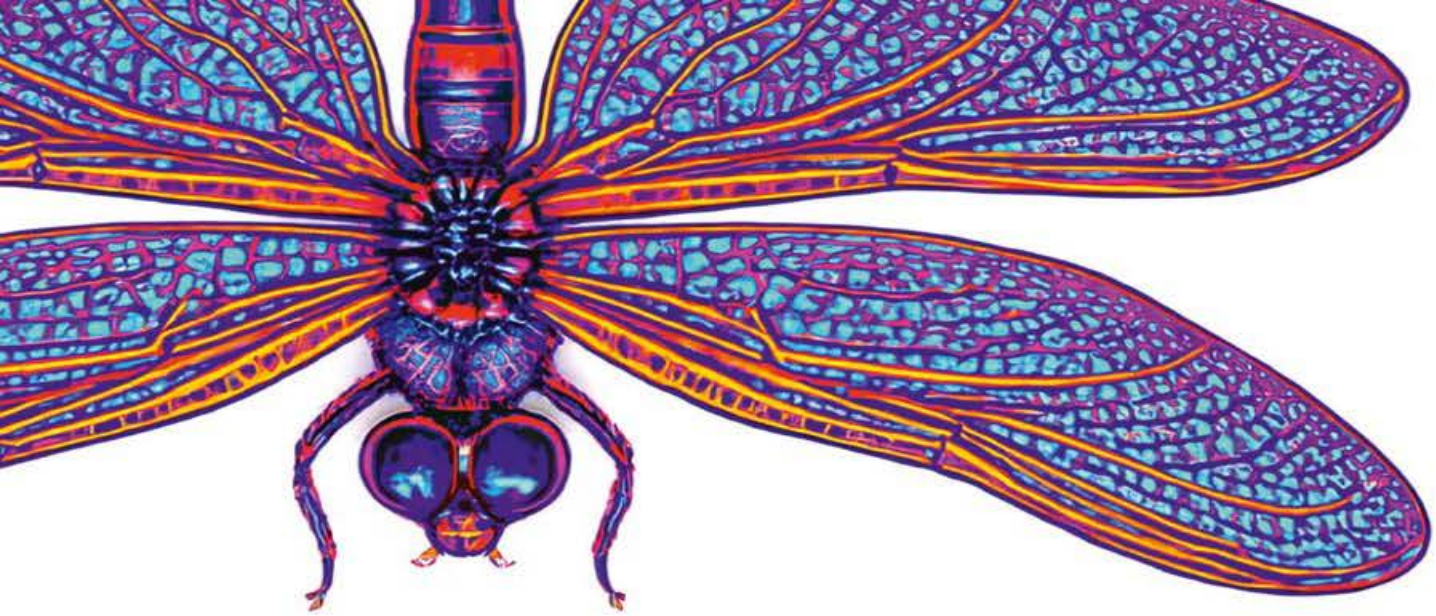
Ответственный за выпуск: *И. Резько*
Ответственный редактор: *А. Ходякова*
Оформление обложки: *С. Дарсич*

Подписано в печать 13.10.2025.
Формат 70x100$^{1}/_{16}$. Бумага офсетная. Печать офсетная. Гарнитура Noto Serif SemiCondensed.
Усл. печ. л. 39. Тираж 1500 экз. Заказ №

Общероссийский классификатор продукции ОК-034-2014 (КПЕС 2008);
58.11.1 — книги, брошюры печатные.

Изготовлено в 2026 г.
Произведено в Российской Федерации.
Изготовитель: ООО «Издательство АСТ».
129085, Российская Федерация, г. Москва, Звёздный бульвар, дом 21,
строение 1, комната 705, пом. I, 7 этаж.
Адрес места осуществления деятельности по изготовлению продукции:
123112, Российская Федерация, г. Москва, Пресненская набережная, д. 6, стр. 2,
Деловой комплекс «Империя», 14, 15 этаж.
Наш электронный адрес: ask@ast.ru
Наш сайт: www.ast.ru
Интернет-магазин: www.book24.ru

Өндіруші: «Издательство АСТ» ЖШҚ 129085, Ресей Федерациясы,
Мәскеу, Звёздный бульвары, 21-үй, 1-құрылыс, 705-бөлме, I үй-жай, 7-қабат.
Өнім өндіру қызметін жүзеге асыру мекенжайы: 123112, Ресей Федерациясы, Мәскеу,
Пресненская жағ., 6-үй, 2-құр., «Империя» іскерлік кешені, 14, 15-қабат.
Біздің электрондық мекенжаймыз: www.ast.ru E-mail: ask@ast.ru
Интернет-магазин: www.book24.ru Интернет-дүкен: www.book24.kz
Импортер в Республику Казахстан, дистрибьютор и представитель по приёму претензий
на продукцию в Республике Казахстан: ТОО «РДЦ-Алматы».
г. Алматы, ул. Домбровского, 3«а», литер Б, офис 1.
Қазақстан Республикасына импорттаушы дистрибьютор және Қазақстан Республикасында өнімге
шағымдар қабылдау жөніндегі өкіл: «РДЦ-Алматы» ЖШС.
Алматы қ., Домбровский көш., 3«а», Б литері, офис 1.
Тел.: 8(727) 2 51 59 90,91, факс: 8 (727) 251 59 92 ішкі 107;
E-mail: RDC-Almaty@eksmo.kz, www.book24.kz
Өндірілген жылы: 2026. Өнімнің жарамдылық мерзімі шектелмеген.
Сертификаттауға жатпайды.
Ресей Федерациясында өндірілген.



Перед вами нечто гораздо большее, чем просто пособие по веб-разработке. Данное издание — это практическое руководство по работе в широко распространенной кроссплатформенной среде выполнения Node.js, в основе которой лежит язык программирования JavaScript. Благодаря ему начинающие программисты смогут быстро поднять свой уровень профессиональных знаний в области создания серверов, API, разработки веб-приложений, работы с базами данных и менеджерами пакетов NPM и Yarn, а также использования WebSocket.

Node.js — это универсальный и уникальный инструмент как frontend-, так и backend-разработки, позволяющий реализовывать масштабные веб-порталы с использованием единого стека технологий и применяемый крупнейшими мировыми IT-компаниями. А поскольку в книге, основанной на практических примерах от экспертов международного сообщества Stack Overflow, собрано буквально все — от основ работы с Node.js до последних профессиональных инноваций, — она будет полезна как начинающим, так и опытным программистам, которые стремятся наиболее продуктивно реализовать свой потенциал в сфере веб-разработки.

КНИГИ ДЛЯ ЛЮБОГО НАСТРОЕНИЯ ЗДЕСЬ:
www.ast.ru / www.book24.ru
vk.com/izdatelstvoast
ok.ru/izdatelstvoast

ИЗДАТЕЛЬСКАЯ ГРУППА АСТ

ISBN 978-5-17-162197-1

9 785171 621971