МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЖИТОМИРСЬКИЙ ДЕРЖАВНИЙ ТЕХНОЛОГІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ

І.А. Оринчак

МЕТОДИЧНІ ВКАЗІВКИ ДЛЯ САМОСТІЙНОЇ РОБОТИ

з курсу «Node.JS»

для студентів напряму підготовки 6.050103
«Програмна інженерія»
денна та заочна форма навчання

Житомир

2016

ЗМІСТ

JavaScript и Node.js

JavaScript и Вы

До того как мы поговорим о технических вещах, позвольте занять некоторое время и поговорить о вас и ваших отношениях с JavaScript. Эта глава позволит вам понять, имеет ли смысл читать дальше.

Скорее всего, как и в моем случае, вы начали свой путь в веб-разработке с написания простых статических HTML-документов. Вместе с этим, вы познакомились с веселой штукой, называемой JavaScript, но использовали его исключительно в простых случаях, добавляя интерактивности на ваши веб-странички.

Что вы хотели узнать — так это действительно полезные вещи; вы хотели знать, как создать сложный сайт. Для этого вы изучали PHP, Ruby, Java и начинали писать backend-код.
Тем не менее, вы постоянно следили за JavaScript, вы видели, что с появлениям JQuery, Prototype и других фреймворков этот язык стал больше, чем просто *window.open()*.
Однако, это всё ещё относилось к frontend-разработке. Конечно, jQuery — очень мощный инструмент, но всякий раз, когда вы приправляли ваш сайт разными jQuery-«фишками», в лучшем случае, вы были JavaScript-*пользователем* нежели JavaScript-*разработчиком*.
А потом пришел Node.js. JavaScript на сервере: насколько это хорошо?
И вы решили, что пора проверить старый новый JavaScript. Подождите. Написать Node.js приложение — одно дело, а понять, почему оно должно быть написано таким образом, для этого нужно понимать JavaScript. И на этот раз — по-настоящему.
В этом — как раз и проблема. JavaScript живёт двумя, может даже тремя разными жизнями: весёлый маленький DHMTL-помощник из середины 90-х годов, более серьезный frontend-инструмент в лице jQuery и наконец серверный (server-side, backend) JavaScript. По этой причине не так просто найти информацию, которая поможет вам познать правильный JavaScript, пригодный для написания Node.js приложения в манере, дающий ощущение, что вы не просто использовали JavaScript, а действительно разрабатывали на JavaScript.
Это — наиболее правильный подход. Вы — уже опытный разработчик, вы не хотите изучать новые технологии поверхностно, просто валяя дурака. Вы хотите быть уверенным, что вы подходите к проблеме под правильным углом.
Конечно, существует отличная документация по Node.js, но её зачастую недостаточно. Нужно руководство.
Моя цель заключается в обеспечении вас руководством.

Предупреждение

Существуют действительно отличные специалисты в области JavaScript. Я не из их числа.
Я — действительно, тот парень, о котором написано в предыдущем параграфе. Я знаю кое-что о разработке backend веб-приложений, но я всё ещё новичок в «реальном» JavaScript и всё ещё новичок в Node.js. Я узнал некоторые продвинутые аспекты JavaScript совсем недавно. Я неопытен.
Вот почему эта книга не из разряда «от новичка к эксперту», а скорее «от новичка к продвинутому новичку».
Если всё удастся, то этот документ станет тем руководством, которое я хотел бы иметь, когда начинал в Node.js.

Server-side JavaScript

Первая инкарнация JavaScript жила в теле браузера. Но это всего лишь контекст. Он определяет, что вы можете делать с языком, но не говорит о том, что язык сам по себе может сделать. JavaScript это «полноценный» язык: вы можете использовать его в различных контекстах и достичь всего того, что можете достичь с другими «полноценными» языками.

Node.js — действительно, просто другой контекст: он позволяет вам запускать JavaScript-код вне браузера.
Чтобы ваш JavaScript код выполнился на *вычислительной машине вне браузера* (на **backend**), он должен быть интерпретирован и, конечно же, выполнен. Именно это и делает Node.js. Для этого он использует движок V8 VM от Google — ту же самую среду исполнения для JavaScript, которую использует браузер Google Chrome.
Кроме того, Node.js поставляется со множеством полезных модулей, так что вам не придется писать всё с нуля, как, например, вывод строки в консоль.
Таким образом, Node.js состоит из 2 вещей: среды исполнения и полезных библиотек.
Для того чтобы их использовать, вам необходимо установить Node.js. Вместо повторения всего процесса установки здесь, я просто приглашу вас посетить официальную инструкцию по инсталляции. Пожалуйста, вернитесь обратно после успешной установки.

«Hello world»

Хорошо, давайте пойдём сразу с места в карьер и напишем наше первое Node.js-приложение: «Hello world».
Откройте ваш любимый редактор и создайте файл под названием*helloworld.js*.
Мы хотим вывести строку «Hello world» в консоль, для этого пишем следующий код:

```
console.log("Hello World");
```

Сохраняем файл и выполняем его посредством Node.js:

```
node helloworld.js
```

Это должно вывести *Hello World* на наш терминал.
Ладно, всё это скучно, правда? Давайте напишем что-нибудь полезное.

Полномасштабное веб-приложение с Node.js

Что должно делать наше приложение

Возьмём что-нибудь попроще, но приближенное к реальности:

- Пользователь должен иметь возможность использовать наше приложение с браузером;
- Пользователь должен видеть страницу приветствия по адресу http://*domain*/start;
- Когда запрашивается http://*domain*/upload, пользователь должен иметь возможность загрузить картинку со своего компьютера и просмотреть её в своем браузере.

Вполне достаточно. Конечно, вы могли бы достичь этой цели, немного погуглив и поговнокодив. Но это не то, что нам нужно.
Кроме того, мы не хотим писать только простой код для достижения цели, каким бы он элегантным и корректным ни был. Мы будем интенсивно наращивать больше абстракции, чем это необходимо, чтобы понять как создавать более сложные Node.js-приложения.

Задачи

Давайте проанализируем наше приложение. Что нужно, чтобы его реализовать:

- У нас — онлайн веб-приложение, поэтому нам нужен **HTTP-сервер;**
- Нашему серверу необходимо обслуживать различные запросы в зависимости от URL, по которому был сделан запрос. Для этого нам нужен какой-нибудь роутер (маршрутизатор), чтобы иметь возможность направлять запросы определенным обработчикам;
- Для выполнения запросов, пришедших на сервер и направляемые роутером, нам нужны действующие **обработчики запросов;**
- Роутер, вероятно, должен иметь дело с разными входящими POST-данными и передавать их обработчикам запросов в удобной форме. Для этого нам нужен какой-нибудь **обработчик входных данных;**
- Мы хотим не только обрабатывать запросы, но и показывать пользователю контент по запрошенным URL-адресам, поэтому нам нужна некая **логика отображения** для обработчиков запросов, чтобы иметь возможность отправлять контент пользовательскому браузеру;
- Последнее, но не менее важное — пользователь сможет загружать картинки, поэтому нам нужен какой-нибудь **обработчик загрузки,** который возьмёт на себя заботу о деталях.

Давайте подумаем о том, как бы мы реализовали это на PHP. Скорее всего, типичное решение будет на HTTP-сервере Apache с установленным mod_php5. Это относится к первому пункту наших задач, то есть, «принимать HTTP-запросы и отправлять готовые веб-странички пользователю» — вещи, которые PHP сам не делает.

С Node.js — немного иначе. Потому что в Node.js мы не только создаем наше приложение, мы также реализуем полноценный HTTP-сервер. Действительно, наше веб-приложение и веб-сервер — в сущности, одно и тоже.

Может показаться, что это приведет к лишней работе, но сейчас вы увидите, что с Node.js это не так.

Давайте просто начнём реализовывать нашу первую задачу — HTTP-сервер.

Реализация приложения

Простой HTTP-сервер

Когда я подошел к моменту создания своего первого «реального» Node.js-приложения, я задался вопросом, как организовать мой код. Я должен делать всё в одном файле? Большинство учебных пособий в интернете учат как создавать простой HTTP-сервер в Node.js, сохраняя всю логику в одном месте. Что, если я хочу быть уверенным, что мой код останется читабельным по мере реализации всё большего функционала.

На самом деле, достаточно легко отыскивать проблемные участки вашего кода, который разделён на модули.

Это позволяет вам иметь чистый главный файл, который вы исполняете в Node.js и чистые модули, которые могут использоваться главным файлом и друг другом.

Так, давайте создадим главный файл, который мы будем использовать для запуска нашего приложения, и файл модуля, в котором будет находиться наш HTTP-сервер.

Я думаю, это более-менее традиционно назвать главным файлом *index.js*. А код нашего сервера имеет смысл поместить в файл под названием *server.js*.
Давайте начнём с модуля сервера. Создайте файл *server.js* в корневой директории вашего проекта и поместите туда следующий код:

```
var http = require("http");

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

И всё! Вы написали работающий HTTP-сервер. Давайте проверим его, запустив и протестировав. Во-первых, выполните ваш скрипт в Node.js:

```
node server.js
```

Теперь откройте ваш браузер и перейдите по адресу http://localhost:8888/. Должна вывестись веб-страница со строкой «Hello world».
Правда, это довольно интересно? Как насчёт того, чтобы поговорить о том, что здесь происходит и оставить на потом вопрос о том, как организовать наш проект? Я обещаю, мы вернемся к нему.

Анализ нашего HTTP-сервера

Хорошо, тогда давайте проанализируем, что здесь действительно происходит.
Первая строчка подключает http-модуль, который поставляется вместе с Node.js и делает его доступным через переменную *http*.
Далее, мы вызываем одну из функций http-модуля *createServer*. Эта функция возвращает объект, имеющий метод *listen*, принимающий числовое значение порта нашего HTTP-сервера, который необходимо прослушивать.
Пожалуйста, проигнорируйте функцию, которая определяется внутри скобок *http.createServer*.
Мы могли бы написать код, который запускает наш сервер, прослушивающий порт 8888, так:

```
var http = require("http");

var server = http.createServer();
server.listen(8888);
```

Это запустило бы HTTP-сервер прослушивающего порт 8888, который больше ничего не делает (даже не отвечает на входящие запросы).
Действительно интересная (и, если вы привыкли к более консервативным языкам как PHP, довольно странная) часть — это определение функции там, где вы бы ожидали увидеть первый параметр для *createServer()*.

Оказывается, эта определяемая функции и есть первый (и только) параметр, который мы передаём в *createServer()* при вызове. Потому что в JavaScript функции могут быть переданы как параметр в другую функцию.

Передача функций в качестве параметра

Вы можете в качестве примера сделать что-то подобное:

```
function say(word) {
  console.log(word);
}

function execute(someFunction, value) {
  someFunction(value);
}

execute(say, "Hello");
```

Разберите пример внимательно! Здесь мы передаём функцию *say* как первый параметр функции *execute*. Не значение, которое возвращает функция *say*, а саму функцию *say*!

Таким образом, *say* становится локальной переменной *someFunction*внутри *execute* и *execute* может вызвать функцию в этой переменной вот так:*someFunction()* (то есть, добавив скобки).

Конечно же, так как *say* принимает один параметр (word), *execute* может передать какое-либо значение в качестве этого параметра, когда вызываетс*omeFunction*.

Мы можем, что мы и сделали, передать функцию как параметр в другую функцию. Но мы не обязаны применять этот косвенный способ, когда сначала определяется функция, а потом передаётся как параметр. Мы можем определить и передать функцию как параметр в другую функцию прямо на месте:

```
function execute(someFunction, value) {
  someFunction(value);
}

execute(function(word){ console.log(word) }, "Hello");
```

Мы определяем функцию, которую хотим передать в *execute*, прямо там, где у *execute* должен быть первый параметр.

Из-за того, что нам даже не надо давать имя этой функции, её называют*анонимная функция*.

Это первый проблеск, который я называю «продвинутый» JavaScript, но давайте всё по порядку. А сейчас давайте просто примем то, что в JavaScript мы можем передать функцию как параметр, когда вызываем другую функцию. Мы можем сделать это путём присвоения нашей функции переменной, которую му передаем, или путём определения функции для передачи на месте.

Как анонимная функция делает наш HTTP-сервер рабочим

С этими знаниями давайте вернемся назад к нашему минималистичному HTTP-серверу:

```
var http = require("http");

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

Сейчас должно быть ясно, что мы здесь делаем: передаём в функцию*createServer* анонимную функцию.

Мы можем добиться того же самого через рефакторинг нашего кода:

```
var http = require("http");

function onRequest(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}

http.createServer(onRequest).listen(8888);
```

Может сейчас самое время спросить: Почему мы это делаем так?

Событийно-ориентированные обратные вызовы

Ответ на вопрос a) не так легко дать (по крайней мере для меня), и b) кроется в самой природе работы Node.js — это событийно-ориентированность, то, благодаря чему он работает так быстро.

Возможно, вы захотите занять немного своего времени и почитать отличный пост Felix Geisendörfer Понимание node.js, чтобы прояснить этот момент.

Все сводится к тому факту, что Node.js работает событийно-ориентированно. Ах да, я тоже до конца не понимаю, что это значит. Но я постараюсь объяснить, почему это так важно для тех, кто хочет писать веб-приложения в Node.js.

Когда вызываем метод *http.createServer*, мы, конечно, не только хотим иметь сервер, слушающий какой-то порт. Мы также хотим что-нибудь сделать, когда приходит HTTP-запрос на этот сервер.

Проблема состоит в асинхронности: запрос происходит в любой момент времени, в то время как у нас только один процесс, в котором запущен наш сервер.

Когда пишем PHP-приложения, мы не беспокоимся обо всем этом: всякий раз, когда приходит HTTP-запрос, веб-сервер (обычно Apache) ответвляет новый процесс специально для этого запроса и запускает соответствующий PHP-скрипт с нуля, который выполняется от начала до конца.

Когда приходит новый запрос на порт 8888, относительно потоков управления, мы находимся в середине нашей Node.js-программы. Как это понять, чтоб не помешаться?
Это как раз то, где событийно-ориентированный дизайн Node.js/JavaScript на самом деле помогает. Нам надо узнать некоторые новые понятия, чтобы досконально понять всё это.
Мы создаем сервер и передаём функцию в созданный им метод. Всякий раз, когда наш сервер получает запрос, переданная нами функция будет вызываться. Мы не знаем, когда это произойдет, но у нас теперь есть место, где можем обрабатывать входящий запрос. Это наша переданная функция и не имеет значения, определили ли мы её сначала или передали анонимно.
Этот принцип называется *обратный вызов* или *callback*. Мы передаём в некоторый метод функцию и этот метод исполняет её, когда происходит связанное с методом событие.
По крайней мере для меня, это заняло некоторое время, чтобы понять. Просто почитайте блог Felix Geisendörfer снова, если вы всё ещё не уверены.
Давайте немного поиграем с этим новым понятием. Можем ли мы доказать, что наш код продолжает работать после создания сервера, даже если нет HTTP-запроса и callback-функция, переданная нами, не вызывается? Давайте попробуем:

```
var http = require("http");

function onRequest(request, response) {
  console.log("Request received.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}

http.createServer(onRequest).listen(8888);

console.log("Server has started.");
```

Обратите внимание, что я использую *console.log* для вывода текста «Request received.», когда срабатывает функция *onRequest* (наш callback), а текст «Server has started.» — сразу *после* запуска HTTP-сервера.
Когда мы запустим этот код (как обычно, *node server.js*), он тут же выведет в командной строке «Server has started.». Всякий раз, когда мы делаем запрос нашему серверу (через переход по адресу http://localhost:8888/ в нашем браузере), в командной строке выводится сообщение «Request received.».
Объектно-ориентированный асинхронный серверный JavaScript с callback-ми в действии :-)
(Обратите внимание, что наш сервер, возможно, будет выводить «Request received.» в консоль 2 раза при открытии страницы в браузере. Это происходит

из-за того, что большинство браузеров будут пытаться загрузить фавикон по адресу http://localhost:8888/favicon.ico при запросе http://localhost:8888/)

Как наш сервер обрабатывает запросы

Хорошо, давайте быстро проанализируем остальной код сервера внутри тела нашей callback-функции *onRequest()*.

Когда callback запускается и наша функция *onRequest()* срабатывает, в неё передаются 2 параметра: *request* и *response*.

Они являются объектами и вы можете использовать их методы для обработки пришедшего HTTP-запроса и ответа на запрос (то есть, просто что-то посылать по проводам обратно в браузер, который запрашивал ваш сервер).

И наш код делает именно это: Всякий раз, когда запрос получен, он использует функцию *response.writeHead()* для отправки HTTP-статуса 200 и Content-Type в заголовке HTTP-ответа, а функцию *Response.Write()* для отправки текста «Hello World» в теле HTTP-ответа.

И последнее, мы вызываем *response.end()* чтобы завершить наш ответ.

На данный момент, мы не заботимся о деталях запроса, поэтому мы не используем объект *request* полностью.

Выбор места для нашего серверного модуля

Я обещал, что мы вернёмся к организации нашего приложения. У нас есть код очень простого HTTP-сервера в файле *server.js* и я упоминал, что общепринято иметь главный файл с названием *index.js*, который используется для начальной загрузки и запуска нашего приложения, путём использования других модулей приложения (таких как наш модуль HTTP-сервера в *server.js*).

Давайте поговорим о том, как сделать server.js настоящим Node.js-модулем, чтобы его можно было использовать в нашем главном файле *index.js*.

Как вы могли заметить, мы уже использовали модули в нашем коде:

```
var http = require("http");

...

http.createServer(...);
```

Где-то внутри Node.js живёт модуль под названием «http» и мы можем использовать его в нашем коде, путём подключения и присвоения его результата локальной переменной.

Это делает нашу локальную переменную объектом, содержащим в себе все публичные методы модуля *http*.

Общепринятая практика — использовать имя модуля для имени локальной переменной, но мы свободны в своём выборе делать, как нам нравится:

```
var foo = require("http");

...
```

```
foo.createServer(...);
```

Теперь понятно, как использовать внутренние модули Node.js. А как создать свой собственный модуль и как его использовать?
Давайте выясним это, превратив наш скрипт *server.js* в настоящий модуль.
Оказывается, нам не надо менять слишком многое. Создание модуля означает, что нам нужно *экспортировать* какую-либо функциональность этого модуля в скрипт, который его вызывает.
Сейчас функционал нашего HTTP-сервера надо экспортировать, что довольно просто: скрипты, подключающие наш модуль сервера, просто запускают сервер.
Чтобы сделать это возможным, поместим код нашего сервера в функцию под название *start* и будем экспортировать эту функцию:

```
var http = require("http");

function start() {
  function onRequest(request, response) {
    console.log("Request received.");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

Теперь мы можем создать наш основной файл *index.js*, и запускать наш HTTP-сервер там, хотя код для сервера находится всё ещё в файле *server.js*.
Создаём файл *index.js* со следующим содержимым:

```
var server = require("./server");

server.start();
```

Как вы могли видеть, мы можем использовать модуль сервера просто как внешний модуль: вызвав этот файл и определив для него переменную, экспортированные функции становятся доступны нам.
Вот и всё. Сейчас мы можем запустить наше приложение через главный скрипт и он будет делать всё то же самое, что и раньше:

```
node index.js
```

Великолепно — сейчас мы можем помещать различные части нашего приложения в разные файлы и связывать их вместе, посредством превращения их в модули.

До сих пор мы работали только с первой частью нашего приложения: получение HTTP-запроса. Но нам надо что-нибудь с ним делать. В зависимости от URL, запрошенного браузером у нашего сервера, мы должны реагировать по-разному.

В очень простом приложении мы могли бы делать это напрямую внутри callback-функции *onRequest()*. Но, как я говорил, давайте добавим немного больше абстракции, чтобы сделать наш пример интереснее.

Задание соответствия между разными HTTP-запросами и разными частями нашего кода называется «маршрутизация» («routing», роутинг). Давайте тогда создадим модуль под названием *router*.

Что необходимо для «роутера»?

Нам нужно иметь возможность скармливать запрошенный URL и возможные добавочные GET- и POST-параметры нашему роутеру и, с учётом этого, роутер должен определять, какой код выполнять (этот код есть третья составляющая нашего приложения: коллекция обработчиков запросов, делающие необходимую работу по определённому запросу).

Итак, нам надо рассматривать HTTP-запрос и извлекать запрошенный URL, а также GET/POST-параметры. Можно поспорить, должен ли этот код быть частью роутера или сервера (или даже своего собственного модуля), но давайте сейчас пока просто сделаем его частью сервера.

Вся необходимая нам информация доступна через объект *request*, который передается в качестве первого параметра нашей callback-функции*onRequest()*.

Чтобы интерпретировать эту информацию, нам необходимо добавить кое-какие Node.js-модули, а именно *url* и *querystring*.

Модуль *url* поддерживает методы, которые позволяют нам извлекать различные части URL (такие как запрошенный путь (URL path) и строка параметров запроса (query string)), а *querystring* в свою очередь, используется для парсинга строки параметров запроса (query string):

```
                           url.parse(string).query
                                      |
           url.parse(string).pathname |
                      |               |
                      |               |
                     ------ -------------------
http://localhost:8888/start?foo=bar&hello=world
                           ---       -----
                            |         |
                            |         |
                querystring(string)["foo"]    |
```

```
                    |
querystring(string)["hello"]
```

Конечно, мы также можем использовать *querystring* для парсинга тела POST-запроса, как мы увидим далее.
Давайте сейчас добавим в нашу функцию *onRequest()* логику, необходимую для извлечения пути URL (pathname), запрошенного браузером:

```
var http = require("http");
var url = require("url");

function start() {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

Замечательно. Теперь наше приложение может различать запросы на основе запрошенного пути URL. Это позволяет нам направлять запросы нашим обработчикам запросов в зависимости от пути URL, используя наш роутер. Таким образом, мы можем строить наше приложение RESTful-путём, потому что теперь можем реализовать интерфейс, следующий принципам *Идентификации ресурсов* (смотри статью в википедии REST для справки).
В контексте нашего приложения, это означает, что мы сможем обрабатывать запросы с URL */start* и */upload* разными частями нашего кода. Скоро мы увидим, как всё соединяется вместе.
Теперь самое время написать наш роутер. Создаём новый файл под названием *router.js* со следующим содержимым:

```
function route(pathname) {
  console.log("About to route a request for " + pathname);
}

exports.route = route;
```

Конечно этот код ничего не делает, но сейчас этого достаточно. Давайте сначала посмотрим, как скрепить этот роутер с нашим сервером до того как поместим больше логики в роутер.
Нашему HTTP-серверу необходимо знать о роутере и использовать его. Мы могли бы жёстко прописать эти зависимости в нашем сервере, но, так как мы знаем только сложные способы из нашего опыта в других языках программирования, мы сделаем слабосвязанную зависимость сервера и роутера через внедрение этих зависимостей (можете почитать отличный пост Мартина Фаулера по внедрениям зависимости на английском языкеили статью в Википедии на русском языке для дополнительной информации).
Для начала, расширим нашу серверную функцию *start()*, чтобы дать нам возможность передавать функцию *route()* как параметр:

```
var http = require("http");
var url = require("url");

function start(route) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    route(pathname);

    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

Теперь расширим наш *index.js* соответственно, то есть внедрим функцию*route()* нашего роутера в сервер:

```
var server = require("./server");
var router = require("./router");

server.start(router.route);
```

Мы опять передаём функцию, которая не является чем-то новым для нас.
Если мы сейчас запустим наше приложение (*node index.js, как обычно*) и запросим какой-нибудь URL, вы сможете увидеть в консоли, что наш HTTP-сервер использует наш роутер и передает ему запрошенный pathname:

```
bash$ node index.js
Request for /foo received.
About to route a request for /foo
```

(Я опустил слегка надоедливый вывод для запроса /favicon.ico)

Исполнение королевских постановлений в царстве глаголов

Позвольте мне ещё раз побродить вокруг и около и снова поговорить о функциональном программировании.

Передача функций связана не только с техническими соображениями. Относительно разработки программного обеспечения это — почти философия. Просто подумайте: в нашем index-файле мы могли бы передавать объект *router* в наш сервер и сервер мог бы вызывать функцию*route* этого объекта.

Этим способом мы бы передавали *нечто* и сервер использовал бы это нечто, чтобы *сделать* что-то. Эй, роутер, не могли бы вы показать мне маршрут?

Но серверу не нужно нечто. Ему нужно только получить что-то *сделанное*, а чтоб получить уже что-то сделанное, вам не нужно нечто совсем, вам необходимо *действие*. Вам не нужно *существительное*, вам нужен *глагол*.

Понимание фундаментальных умозаключений, которые лежат в основе этой идеи, позволило мне действительно понять функциональное программирование.

Я понял это, когда читал шедевр Стива Йегге Execution in the Kingdom of Nouns (частичный перевод на русский Исполнение королевских постановлений в царстве существительных). Почитайте это обязательно. Это одно из лучших произведений о программировании, которое я когда-либо имел удовольствие встречать.

Роутинг реальных обработчиков запроса

Вернёмся к делу. Наш HTTP-сервер и наш роутер запросов сейчас — лучшие друзья, и общаются друг с другом так, как мы хотели.

Конечно, этого недостаточно. «Роутинг» подразумевает, что мы хотим обрабатывать запросы на разные URL по-разному. Мы хотели бы иметь «бизнес-логику» для запросов к */start* в одной функции, а для запросов к*/upload* в другой.

Прямо сейчас мы закончим с нашим роутером. Роутер не место на самом деле, чтобы делать что-то с запросами, потому что будет плохо масштабироваться, а наше приложение будет становиться сложнее.

Давайте эти функции, в которые направляются запросы, назовём*обработчиками запросов*. И давайте возьмёмся за них сейчас, потому что делать что-либо с роутером сейчас пока не имеет смысла.

Новая часть приложения, новый модуль — здесь никаких сюрпризов. Создадим модуль под названием requestHandlers, добавим болванки функций для каждого обработчика запроса и экспортируем их как методы модуля:

```
function start() {
  console.log("Request handler 'start' was called.");
}

function upload() {
  console.log("Request handler 'upload' was called.");
}

exports.start = start;
exports.upload = upload;
```

Это позволяет нам связать обработчики запросов с роутером, давая нашему роутеру что-нибудь маршрутизировать.

В этот момент мы должны принять решение: захардкодить использование модуля requestHandlers в роутере или мы хотим ещё немного внедрения зависимостей? Хотя внедрение зависимостей, как и любой другой паттерн, не должен использоваться только ради того, чтобы быть использованным, в нашем случае имеет смысл сделать слабосвязанную пару роутера и обработчиков запроса и, таким образом, сделать роутер действительно многоразовым.

Это означает, что нам нужно передавать обработчики запросов из нашего сервера в наш роутер, но это немного неправильно, поэтому мы должны пройти весь путь и передать их в сервер из нашего главного файла, а также — оттуда передавать в роутер.

Как мы собираемся передать их? Сейчас у нас есть два обработчика, но в реальном приложении это число будет увеличиваться и меняться. И мы уверены, что не хотим возиться с роутером каждый раз, когда добавляется новый URL + обработчик запроса. И какие-нибудь *if запрос == x then вызвать обработчик y* в роутере будут более чем убоги.

Переменное число элементов и каждому соответствует строка (запрашиваемый URL)? Так, похоже на ассоциативный массив, это наиболее подходящее.

Это решение немного разочаровывает тем фактом, что JavaScript не поддерживает ассоциативные массивы. Или нет? Оказывается в действительности, если нам нужны ассоциативные массивы, мы должны использовать объекты!

Об этом есть хорошее введение http://msdn.microsoft.com/en-us/magazine/cc163419.aspx. Позвольте мне процитировать подходящую часть:

В C++ или C#, когда мы говорим об объектах, мы ссылаемся на экземпляры классов или структуры. Объекты имеют разные свойства и методы, в зависимости от шаблонов (классов), экземплярами которых они являются. Но не в случае с JavaScript-объектами. В JavaScript, объекты — это просто коллекция пар имя/значение — JavaScript-объект — это как словарь со строковыми ключами.

Если JavaScript-объекты это просто коллекции пар имя/значение, как тогда у них могут быть методы? Итак, значения могут быть строками, числами и т.д. или функциями!

Хорошо, наконец-то возвращаемся к нашему коду. Мы решили, что мы хотим передать список из requestHandlers как объект и, для того, чтобы достичь слабое связывание, мы хотим внедрить этот объект в *route()*.
Начнём с добавления объекта в наш главный файл *index.js*:

```
var server = require("./server");
var router = require("./router");
var requestHandlers = require("./requestHandlers");

var handle = {}
handle["/"] = requestHandlers.start;
handle["/start"] = requestHandlers.start;
handle["/upload"] = requestHandlers.upload;

server.start(router.route, handle);
```

Хотя *handle* — это больше из разряда «нечто» (коллекция обработчиков запроса), я, всё-таки, предлагаю называть его глаголом, потому что в результате это будет функциональное выражение в нашем роутере, как вы скоро увидите.
Как вы можете видеть, это действительно просто — назначать различные URL соответствующему обработчику запроса: просто добавляя пару ключ/значение из *«/»* и *requestHandlers.start*, мы можем выразить красивым и аккуратным способом, что не только запросы к *«/start»*, но также и запросы к *«/»* должны быть обработаны обработчиком *start*.
После определения объекта мы передали его в сервер как дополнительный параметр. Изменим наш *server.js*, чтобы использовать его:

```
var http = require("http");
var url = require("url");

function start(route, handle) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    route(handle, pathname);

    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}
```

```
exports.start = start;
```

Мы добавили параметр *handle* в функцию *start()* и передаём объект handle в callback-функцию *route()* в качестве перового параметра.
Соответственно, изменим функцию *route()* в нашем файле *router.js*:

```
function route(handle, pathname) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname]();
  } else {
    console.log("No request handler found for " + pathname);
  }
}

exports.route = route;
```

Что мы здесь делаем — мы проверяем, существует ли обработчик запроса для данного пути, и если существует, просто вызываем соответствующую функцию. Из-за того, что мы имеем доступ к нашим функциям обработчиков запроса из нашего объекта просто, как если бы имели доступ к элементу ассоциативного массива, у нас есть это прекрасное выражение*handle[pathname]();*, о котором говорилось ранее: «Пожалуйста, *handle* этот*pathname*».
Хорошо, это всё, что нужно, чтобы связать сервер, роутер и обработчики запроса вместе! При запуске нашего приложения и запросаhttp://localhost:8888/start в браузере, мы можем убедиться, что надлежащий обработчик запроса действительно был вызван:

```
Server has started.
Request for /start received.
About to route a request for /start
Request handler 'start' was called.
```

Так же открываем http://localhost:8888/ в нашем браузере и убеждаемся, что эти запросы в самом деле обрабатываются обработчиком запросов*start*:

```
Request for / received.
About to route a request for /
Request handler 'start' was called.
```

Создание ответа обработчиков запроса

Замечательно. Вот только если бы обработчики запроса могли отправлять что-нибудь назад браузеру, было бы ещё лучше, правильно?

Вспомните, «Hello World», который выводит ваш браузер в запрошенной странице, всё ещё исходит от функции *onRequest* в нашем файле *server.js*.
«Обработка запроса» подразумевает «ответ на запросы» в конце концов, поэтому необходимо дать возможность нашим обработчикам запроса общаться с браузером так же, как это делает функция *onRequest*.

Как делать не надо

Прямой подход, который мы захотим использовать как разработчики с опытом в PHP или Ruby, на самом деле ложный: он может прекрасно работать, иметь большой смысл, а потом, когда мы этого не ждём, неожиданно всё развалится.
Под «прямым подходом» я подразумеваю использование в обработчиках запроса *return ""* для контента, который надо показать пользователю, и отправлять этот ответ в функцию *onRequest* назад пользователю.
Давайте просто сделаем это и тогда увидим, почему это не такая уж и хорошая идея.
Мы начнём с обработчиков запроса и заставим их возвращать то, что хотели бы показать в браузере. Нам надо изменить *requestHandlers.js* вот так:

```
function start() {
  console.log("Request handler 'start' was called.");
  return "Hello Start";
}

function upload() {
  console.log("Request handler 'upload' was called.");
  return "Hello Upload";
}

exports.start = start;
exports.upload = upload;
```

Хорошо. Также, роутер должен вернуть серверу то, что обработчики запроса вернули ему. Поэтому надо отредактировать *router.js* так:

```
function route(handle, pathname) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    return handle[pathname]();
  } else {
    console.log("No request handler found for " + pathname);
    return "404 Not found";
  }
}

exports.route = route;
```

Как видим, возвращается некоторый текст «404 Not found», если запрос не может быть маршрутизирован.
И самое последнее, но не менее важное, нам нужен рефакторинг нашего сервера, чтобы заставить его отвечать браузеру с контентом обработчиков запроса, возвращаемых через роутер. Трансформируем *server.js* в:

```
var http = require("http");
var url = require("url");

function start(route, handle) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    response.writeHead(200, {"Content-Type": "text/plain"});
    var content = route(handle, pathname)
    response.write(content);
    response.end();
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

Если запустим наше написаное приложение, всё будет работать замечательно: запрос http://localhost:8888/start выдаст в браузере результат «Hello Start», запрос http://localhost:8888/upload даст нам «Hello Upload», а http://localhost:8888/foo выведет «404 Not found».
OK, тогда в чём проблема? Короткий ответ: потому что мы столкнемся с проблемами, если один из обработчиков запроса захочет использовать неблокирующую операцию в будущем.
Подробный ответ займёт немного больше времени.

Блокирование и неблокирование

Как было сказано, проблемы будут возникать, когда мы добавим неблокирующие операции в обработчики запроса. Давайте сначала поговорим о блокирующих, а потом уже о неблокирующих операциях.
Вместо того, чтобы объяснять, что такое «блокирование» и «неблокирование», давайте продемонстрируем себе, что произойдёт, если мы добавим блокирующую операцию в наши обработчики запроса.
Для этого модифицируем обработчик запроса *start* так, чтобы он ждал 10 секунд до того как вернёт свою строку «Hello Start». В JavaScript нет такой штуки как *sleep()*, поэтому мы будем использовать хитрый хак.
Пожалуйста, измените *requestHandlers.js* как описано далее:

```
function start() {
  console.log("Request handler 'start' was called.");

  function sleep(milliSeconds) {
    var startTime = new Date().getTime();
    while (new Date().getTime() < startTime + milliSeconds);
  }

  sleep(10000);
  return "Hello Start";
}

function upload() {
  console.log("Request handler 'upload' was called.");
  return "Hello Upload";
}

exports.start = start;
exports.upload = upload;
```

Просто объясню, что этот код делает: когда функция *start()* вызвана, Node.js ожидает 10 секунд и только тогда возвращает «Hello Start». Когда вызывается *upload()*, она выполняется немедленно, как и раньше.
(Конечно, вы уже поняли, вместо засыпания на 10 секунд, в *start()* могут быть реальные блокирующие операции, такие как сложные длительные вычисления.)
Давайте посмотрим, что поменялось.
Как обычно, нам надо перезапустить сервер. На этот раз я попрошу вас следовать немного более сложному «протоколу», чтобы увидеть, что произошло: во-первых, откройте браузер или таб. В первом окне браузера, введите, пожалуйста, http://localhost:8888/start в адресную строку, но не переходите пока по этому адресу!
В адресную строку второго окна браузера введитеhttp://localhost:8888/upload и снова не переходите по адресу.
Теперь сделайте, как описано далее: нажмите клавишу Enter в первом окне («/start»), а затем быстро переключитесь на второе окно («/upload») и нажмите тоже Enter.
Что вы будете наблюдать: URL /start потребуется 10 секунд для загрузки, как мы и ожидали. Но URL /upload *так же* потребуется 10 секунд на загрузку, хотя в соответствующем обработчике запроса нет *sleep()*!
Почему? Потому что *start()* содержит блокирующую операцию. Like in "it's blocking everything else from working".
И в этом проблема, потому что, как говорят: *«В node всё работает параллельно, за исключением вашего кода»*.
Это значит, что Node.js может обрабатывать одновременно множество вещей, но при этом не разделяет всё на отдельные потоки — Node.js однопоточный. Он

делает это, запуская цикл событий, а мы, разработчики, можем использовать это — мы должны избегать блокирующих операций, где это возможно, и использовать неблокирующие операции вместо них.
Но для этого нам надо использовать обратные вызовы, передавая функции внутри тех функций, которые могут сделать то, что занимает некоторое время (как например sleep() на 10 секунд или запрос к базе данных или какое-то дорогостоящее вычисление.)
Таким образом, мы как бы говорим: *«Эй, возможноДолгаяФункция(), пожалуйста, сделай вот это, но я, однопотоковый Node.js, не собираюсь ждать здесь, пока ты закончишь, я продолжу выполнение строчек кода ниже тебя, а ты возьми пока вот эту функцию callbackFunction() и вызови её, когда всё сделаешь. Спасибо!»*
(Если хотите почитать об этом более подробно, пожалуйста посмотрите пост Mixu на Understanding the node.js event loop.)
И мы сейчас увидим, почему способ, которым мы создали «обработчик запроса обрабатывающий ответ» в нашем приложении не позволит правильно использовать неблокирующие операции.
Ещё раз давайте попробуем испытать проблему на своей шкуре, модифицировав наше приложение.
Мы снова используем наш обработчик запроса *start*. Пожалуйста, измените его следующим образом (файл *requestHandlers.js*)

```
var exec = require("child_process").exec;

function start() {
  console.log("Request handler 'start' was called.");
  var content = "empty";

  exec("ls -lah", function (error, stdout, stderr) {
    content = stdout;
  });

  return content;
}

function upload() {
  console.log("Request handler 'upload' was called.");
  return "Hello Upload";
}

exports.start = start;
exports.upload = upload;
```

Как можно видеть, мы просто внедрили новый модуль Node.js*child_process*. Мы сделали так, потому что это позволит нам использовать очень простую, но полезную неблокирующую операцию: *exec()*.

Что делает *exec()* — она выполняет shell-команду внутри Node.js. В этом примере мы собираемся использовать её, чтобы получить список всех файлов в текущей директории ("ls -lah"), позволяя нам отобразить этот список в браузере пользователя, запросившего URL */start*.
Что делает этот код: создает новую переменную *content* (с начальным значением "empty"), выполняет "ls -lah", заполняет переменную результатом и возвращает её.
Как обычно, запустим наше приложение и посетимhttp://localhost:8888/start.
Которая загрузит нам красивую страничку со строкой "empty". Что тут не так?
Ну, как вы уже догадались, *exec()* делает свою магию в неблокирующий манере. Это хорошая штука, потому что таким образом мы можем выполнять очень дорогостоящие shell-операции (как, например, копирование больших файлов или что-то подобное), не заставляя наше приложение полностью останавливаться, пока блокирующая *sleep*-операция не выполнится.
Если хотите удостовериться, замените "ls -lah" на более дорогостоящую операцию "find /").
Но мы не совсем довольны своей элегантной неблокирующей операцией, когда наш браузер не отображает её результат, не так ли?
Давайте тогда пофиксим это. Давайте попытаемся понять, почему текущая архитектура не работает.
Проблемой является то, что *exec()*, чтобы работать без блокирования, использует callback-функцию.
В нашем примере это анонимная функция, которая передаётся как второй параметр в функцию *exec()*:

```
function (error, stdout, stderr) {
  content = stdout;
}
```

И здесь лежит корень нашей проблемы: наш собственный код исполняется синхронно, что означает, что сразу после вызова *exec()*, Node.js продолжит выполнять *return content;*. К этому моменту *content* ещё "empty", из-за того, что callback-функция, переданная в *exec()*, до сих пор не вызвана — потому что операция *exec()* асинхронная.
Теперь "ls -lah" — очень недорогая и быстрая операция (если только в директории не миллион файлов). Именно поэтому callback вызывается относительно оперативно — но это, всё же, происходит асинхронно.
Использование более дорогостоящих команд делает это более очевидным: "find /" занимает около 1 минуты на моей машине, но если я заменяю "ls -lah" на "find /" в обработчике запроса, то я всё ещё немедленно получаю HTTP-ответ, когда открываю URL /start. Ясно, что *exec()* делает что-то в фоновом режиме, пока Node.js продолжает исполнять приложение и мы можем предположить, что callback-функция, которую мы передали в *exec()*, будет вызвана только когда команда "find /" закончит выполняться.
Но как нам достичь нашей цели, то есть, показать пользователю список файлов в текущей директории?

Теперь, после изучения вопроса о том, как делать *не* надо, давайте обсудим, как заставить наши обработчики запроса реагировать на запросы браузера правильно.

Ответ обработчиков запроса с неблокирующими операциями.

Я употребил фразу «правильный способ». Опасная вещь. Довольно часто не существует единого «правильного способа».

Но одним из возможных решений для этого является, как это часто бывает с Node.js, передача функции внутри. Давайте рассмотрим это.

Сейчас наше приложение способно транспортировать контент (который обработчики запроса хотели бы показать пользователю) от обработчиков запроса к HTTP-серверу, возвращая его через слои приложения (обработчик запроса -> роутер -> сервер).

Наш новый подход заключается в следующем: вместо доставки контента серверу мы будем сервер доставлять к контенту. Чтобы быть более точным, мы будем внедрять объект *response* (из серверной callback-функции*onRequest()*) через роутер в обработчики запроса. Обработчики смогут тогда использовать функции этого объекта для ответа на сами запросы.

Достаточно разъяснений. Вот — пошаговый рецепт изменения нашего приложения.

Начнём с нашего *server.js*:

```
var http = require("http");
var url = require("url");

function start(route, handle) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    route(handle, pathname, response);
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

Вместо ожидания возврата значения от функции *route()*, мы передаём наш объект *response* в качестве третьего параметра. Кроме того, мы удалили всякие вызовы методов *response* из обработчика *onRequest()*, потому что мы рассчитываем, что *route* позаботится об этом.

Далее идёт *router.js*:

```
function route(handle, pathname, response) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response);
  } else {
    console.log("No request handler found for " + pathname);
    response.writeHead(404, {"Content-Type": "text/plain"});
    response.write("404 Not found");
    response.end();
  }
}

exports.route = route;
```

Та же схема: вместо ожидания возврата значения от наших обработчиков события, мы передаём объект *respond*.
Если обработчик запроса не может быть использован, мы заботимся об ответе с надлежащим заголовком «404» и телом ответа.
И последнее, но не менее важное, мы модифицируем *requestHandlers.js*:

```
var exec = require("child_process").exec;

function start(response) {
  console.log("Request handler 'start' was called.");

  exec("ls -lah", function (error, stdout, stderr) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write(stdout);
    response.end();
  });
}

function upload(response) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello Upload");
  response.end();
}

exports.start = start;
exports.upload = upload;
```

Наши функции-обработчики должны принять параметр response и использовать его, чтобы ответить на запрос напрямую.

Обработчик *start* будет отвечать изнутри анонимного обратного вызова*exec()*, а обработчик *upload* будет всё ещё выдавать «Hello Upload», но теперь посредством объекта *response*.
Если вы запустили наше приложение снова (*node index.js*), всё должно работать как и ожидалось.
Если хотите убедиться, что дорогостоящая операция в */start* больше не будет блокировать запросы на */upload*, модифицируйте ваш*requestHandlers.js* как показано далее:

```
var exec = require("child_process").exec;

function start(response) {
  console.log("Request handler 'start' was called.");

  exec("find /",
    { timeout: 10000, maxBuffer: 20000*1024 },
    function (error, stdout, stderr) {
      response.writeHead(200, {"Content-Type": "text/plain"});
      response.write(stdout);
      response.end();
    });
}

function upload(response) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello Upload");
  response.end();
}

exports.start = start;
exports.upload = upload;
```

Благодаря этому, HTTP-запросы к http://localhost:8888/start будут занимать не менее 10 секунд, но запросы к http://localhost:8888/uploadбудут получать ответ немедленно, даже если /start всё ещё занят вычислениями.

Сделаем что-нибудь полезное

До сих пор мы делали всё прекрасно и изысканно, но мы не создали ничего значимого для клиентов нашего супер-сайта.
Наш сервер, роутер и обработчики запроса находятся на своих местах, таким образом, теперь мы можем начать добавлять контент на наш сайт, который позволяет нашим пользователям выбирать файл, загружать его и просматривать загруженный файл в браузере. Для простоты будем полагать, что через наше приложение будут загружаться и показываться только файлы картинок.

OK, давайте шаг за шагом, но с разъяснением больших техник и принципов JavaScript, и в то же время, давайте немного ускоримся. Автору слишком нравится слушать самого себя.
Здесь "шаг за шагом" означает примерно 2 шага: сначала мы посмотрим как обрабатывать входящие POST-запросы (но не загрузку файла), и на втором шаге мы используем внешний модуль Node.js для обработки загрузки файла. Я выбирал этот подход по двум причинам.
Во-первых, обрабатывать базовые POST-запросы относительно просто в Node.js, но для обучения это — достаточно стоящее упражнение. Во-вторых, обработка загрузки файла (к примеру, multipart POST-запросы) это *не так просто* в Node.js, поэтому выходит за рамки этого учебника, но пример использования для этого внешнего модуля имеет смысл включить в учебник для начинающих.

Обработка POST-запросов

Давайте сделаем попроще: предоставим текстовое поле, которое может быть заполнено пользователем и отправлено на сервер в POST-запросе. После получения и обработки этого запроса мы отобразим содержимое текстового поля.
HTML-код для формы текстового поля должен формировать наш обработчик запроса */start*, так давайте сразу же добавим его в файл*requestHandlers.js*:

```
function start(response) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html; '+
    'charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" method="post">'+
    '<textarea name="text" rows="20" cols="60"></textarea>'+
    '<input type="submit" value="Submit text" />'+
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
    response.end();
}

function upload(response) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
```

```
  response.write("Hello Upload");
  response.end();
}

exports.start = start;
exports.upload = upload;
```

Если теперь этот код не выиграет Webby Awards, то я не знаю, какой сможет. Вы должны увидеть эту очень простую форму, когда запроситеhttp://localhost:8888/start в вашем браузере. Если это не так — возможно, вы не перезагрузили приложение.

Я уже слышу вас: помещать содержимое представления прямо в обработчик запроса некрасиво. Тем не менее, я решил не включать этот дополнительный уровень абстракции (то есть, разделение представления и логики) в наш учебник, потому что, я думаю, что это не научит нас чему-нибудь стоящему в контексте JavaScript или Node.js.

Давайте лучше использовать появившееся окно для более интересных проблем, то есть, обработки POST-запроса в нашем обработчике запроса*/upload* при отправке этой формы пользователем.

Теперь, когда мы стали экспертными новичками, мы уже не удивляемся тому факту, что обработка POST-данных делается в неблокирующей манере, через использование асинхронных callback-ов.

Это имеет смысл, потому что POST-запросы могут быть потенциально очень большими — ничто не мешает пользователю ввести текст размером в несколько мегабайтов. Обработка всего массива данных за один раз может привести к блокирующей операции.

Чтобы сделать весь процесс неблокирующим, Node.js обслуживает POST-данные небольшими порциями, а callback-функции вызываются при определённых событиях. Эти события — *data* (когда приходит новая порция POST-данных) и *end* (когда все части данных были получены).

Надо сообщить Node.js, какие функции вызывать, когда эти события произойдут. Это делается путём добавления слушателей (*listeners*) в объект*request*, который передаётся в нашу callback-функцию *onRequest*, когда HTTP-запрос получен.

В основном, это выглядит так:

```
request.addListener("data", function(chunk) {
  // called when a new chunk of data was received
});

request.addListener("end", function() {
  // called when all chunks of data have been received
});
```

Возникает вопрос, где реализовать эту логику. В настоящее время мы можем получить доступ к объекту *request* только в нашем сервере — мы не передаём его в роутер и в обработчики запроса, как делаем это с объектом*response*.
На мой взгляд, это работа HTTP-сервера — предоставлять приложению все данные из запросов. Поэтому я предлагаю обрабатывать POST-данные прямо в сервере и передавать конечные данные в роутер и обработчики запроса, которые сами решат, что с ними делать.
Таким образом, идея — в том, чтобы поместить обратные вызовы событий*data* и *end* в сервер, собирать все куски POST-данных в *data* и вызывать роутер при получении события *end*, пока идёт передача собранных порций данных в роутер, который в свою очередь передаёт их в обработчики запроса.
Начинаем с *server.js*:

```
var http = require("http");
var url = require("url");

function start(route, handle) {
  function onRequest(request, response) {
    var postData = "";
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");

    request.setEncoding("utf8");

    request.addListener("data", function(postDataChunk) {
      postData += postDataChunk;
      console.log("Received POST data chunk '"+
      postDataChunk + "'.");
    });

    request.addListener("end", function() {
      route(handle, pathname, response, postData);
    });

  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

Здесь, в основном, мы сделали три вещи: во-первых, определили, что ожидаем полученные данные в кодировке UTF-8, затем добавили слушатель для события «data», который шаг за шагом заполняет нашу новую переменную *postData* всякий раз, когда прибывает новая порция POST-данных,

и далее — переходим к вызову нашего роутера в обратном вызове события *end*, чтобы убедиться, что вызов происходит, когда все POST-данные собраны. Мы также передаём POST-данные в роутере, потому что они нам понадобятся в обработчиках запроса.
Добавление логирования в консоли для каждой порции полученных данных — возможно, плохая идея для конечного кода (мегабайты POST-данных, помните?), но это имеет смысл, чтобы посмотреть, что происходит.
Я предлагаю немного поиграться с этим. Поместите в текстовое поле сначала немного текста, а потом больше, и вы увидите, что для больших текстов обратный вызов *data* действительно вызывается несколько раз.
Давайте добавим ещё больше крутизны в наше приложение. На странице */upload* мы будем показывать принятый контент. Чтобы сделать это возможным, нам необходимо передавать *postData* в обработчики запроса. В *router.js*:

```
function route(handle, pathname, response, postData) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response, postData);
  } else {
    console.log("No request handler found for " + pathname);
    response.writeHead(404, {"Content-Type": "text/plain"});
    response.write("404 Not found");
    response.end();
  }
}

exports.route = route;
```

И в *requestHandlers.js* мы включаем эти данные в нашем ответе обработчика запроса *upload*:

```
function start(response, postData) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html; '+
    'charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" method="post">'+
    '<textarea name="text" rows="20" cols="60"></textarea>'+
    '<input type="submit" value="Submit text" />'+
    '</form>'+
```

```
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
    response.end();
}

function upload(response, postData) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("You've sent: " + postData);
  response.end();
}

exports.start = start;
exports.upload = upload;
```

Вот и всё, теперь мы можем получить POST-данные и использовать их в наших обработчиках запроса.

И последнее по этой теме: то, что мы передаём в роутер и обработчики запроса, является полным телом нашего POST-запроса. Мы, вероятно, захотим использовать индивидуальные поля, составляющие POST-данные, в нашем случае значение поля *text*.

Мы уже читали про модуль *querystring*, который поможет нам с этим:

```
var querystring = require("querystring");

function start(response, postData) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" content="text/html; '+
    'charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" method="post">'+
    '<textarea name="text" rows="20" cols="60"></textarea>'+
    '<input type="submit" value="Submit text" />'+
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
```

```
    response.write(body);
    response.end();
}

function upload(response, postData) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("You've sent the text: "+
  querystring.parse(postData).text);
  response.end();
}

exports.start = start;
exports.upload = upload;
```

Это всё, что можно сказать про обработку POST-данных в рамках учебника для начинающих.

Обработка загрузки файлов

Давайте примемся за последний пункт нашего списка задач. Мы планировали дать возможность пользователям загружать файлы картинок и отображать загруженные картинки в браузере.

В 90-х это могло бы быть квалифицировано как бизнес модель для IPO, сейчас же этого достаточно, чтобы научить нас двум вещам: как установливать внешнии библиотки Node.js и как их использовать в нашем коде.

Внешний модуль, который мы собираемся использовать, *node-formidable*от Felix Geisendörfer. Этот модуль поможет нам абстрагироваться от мерзких деталей парсинга входящих файловых данных. В конце концов, обработка входящих файлов это не что иное, как «просто» обработка POST-данных, но, в действительности, дьявол кроется в деталях, поэтому в нашем случае имеет смысл использовать готовое решение.

Чтобы использовать код Феликса, соответствующий модуль Node.js должен быть инсталлирован. На борту Node.js есть собственный менеджер пакетов, называемый *NPM*. Он позволяет нам инсталировать внешние модули Node.js в очень удобной форме. С учетом установленного Node.js, всё сводится к

```
npm install formidable
```

в нашей командной строке. Если вы в конце увидели следующее:

```
npm info build Success: formidable@1.0.9
npm ok
```

...это значит — всё хорошо.

Модуль *formidable* теперь доступен в нашем коде — всё, что нужно, это просто запросить его как один из тех модулей, которые мы использовали ранее:

```
var formidable = require("formidable");
```

По сути, *formidable* делает форму, отправленную через HTTP POST, доступной для парсинга в Node.js. Всё, что нам надо — это создать новый экземпляр объекта *IncomingForm*, который является абстракцией отправленной формы и может быть использован для парсинга объекта*request* нашего HTTP-сервера, для полей и файлов, отправленных через эту форму.
Пример кода со страницы проекта node-formidable показывает, как разные части сочетаются друг с другом:

```
var formidable = require('formidable'),
    http = require('http'),
    sys = require('sys');

http.createServer(function(req, res) {
  if (req.url == '/upload' && req.method.toLowerCase() == 'post') {
    // parse a file upload
    var form = new formidable.IncomingForm();
    form.parse(req, function(err, fields, files) {
      res.writeHead(200, {'content-type': 'text/plain'});
      res.write('received upload:\n\n');
      res.end(sys.inspect({fields: fields, files: files}));
    });
    return;
  }

  // show a file upload form
  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(8888);
```

Если вы поместите этот код в файл и исполните его посредством *node*, вы сможете отправлять простые формы, включая загрузку фото, и увидите, как организован объект *files*, который передавался в callback, определенном в вызове *form.parse*.

```
received upload:
```

```
{ fields: { title: 'Hello World' },
  files:
   { upload:
          { size: 1558,
                path: '/tmp/1c747974a27a6292743669e91f29350b',
                name: 'us-flag.png',
                type: 'image/png',
                lastModifiedDate: Tue, 21 Jun 2011 07:02:41 GMT,
                _writeStream: [Object],
                length: [Getter],
                filename: [Getter],
                mime: [Getter] } } }
```

Чтобы выполнить последний пункт нашей задачи, мы должны включить логику парсинга форм модуля *formidable* в структуру нашего кода, плюс ко всему, надо разобраться, как отдавать контент загруженного файла (который сохранен в директории */tmp*) браузеру.

Давайте сначала решим последнюю задачу: если имеется файл картинки на вашем локальном диске, что сделать, чтобы передать его браузеру?

Мы, очевидно, собираемся считать содержимое этого файла в наш Node.js-сервер, и неудивительно, что для этого имеется соответствующий модуль под названием *fs*.

Давайте добавим ещё один обработчик запроса для URL */show*, который будет "захардкоженно" показывать содержимое файла */tmp/test.png*. Конечно же, имеет смысл в первую очередь поместить реальную png-картинку в этот каталог.

Мы собираемся изменить *requestHandlers.js*, как показано далее:

```
var querystring = require("querystring"),
    fs = require("fs");

function start(response, postData) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" '+
    'content="text/html; charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" method="post">'+
    '<textarea name="text" rows="20" cols="60"></textarea>'+
    '<input type="submit" value="Submit text" />'+
    '</form>'+
```

```
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
    response.end();
}

function upload(response, postData) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("You've sent the text: "+
  querystring.parse(postData).text);
  response.end();
}

function show(response, postData) {
  console.log("Request handler 'show' was called.");
  fs.readFile("/tmp/test.png", "binary", function(error, file) {
    if(error) {
      response.writeHead(500, {"Content-Type": "text/plain"});
      response.write(error + "\n");
      response.end();
    } else {
      response.writeHead(200, {"Content-Type": "image/png"});
      response.write(file, "binary");
      response.end();
    }
  });
}

exports.start = start;
exports.upload = upload;
exports.show = show;
```

Также, надо преобразовать новый обработчик запроса в URL вида */show* в файле *index.js*:

```
var server = require("./server");
var router = require("./router");
var requestHandlers = require("./requestHandlers");

var handle = {}
handle["/"] = requestHandlers.start;
handle["/start"] = requestHandlers.start;
```

```
handle["/upload"] = requestHandlers.upload;
handle["/show"] = requestHandlers.show;

server.start(router.route, handle);
```

Перезапускаем сервер, открываем http://localhost:8888/show в браузере и видим картинку */tmp/test.png*.
Хорошо. Всё, что нам надо теперь — это:

- добавить поле для загрузки файлов в форму, находящуюся по адресу*/start*,
- интегрировать *node-formidable* в обработчик запроса *upload*, чтобы сохранять загруженные файлы в */tmp/test.png*,
- внедрить загруженную картинку в HTML, отдаваемый по URL */upload*.

Первый шаг — простой. Нам надо добавить тип кодировки *multipart/form-data* в нашу HTML-форму, удалить текстовое поле, добавить поле загрузки файла и поменять текст кнопки отправки формы на «Upload file». Давайте просто сделаем это в файле *requestHandlers.js*:

```
var querystring = require("querystring"),
    fs = require("fs");

function start(response, postData) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" '+
    'content="text/html; charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" enctype="multipart/form-data" '+
    'method="post">'+
    '<input type="file" name="upload">'+
    '<input type="submit" value="Upload file" />'+
    '</form>'+
    '</body>'+
    '</html>';

    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(body);
    response.end();
}

function upload(response, postData) {
  console.log("Request handler 'upload' was called.");
```

```
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("You've sent the text: "+
  querystring.parse(postData).text);
  response.end();
}

function show(response, postData) {
  console.log("Request handler 'show' was called.");
  fs.readFile("/tmp/test.png", "binary", function(error, file) {
    if(error) {
      response.writeHead(500, {"Content-Type": "text/plain"});
      response.write(error + "\n");
      response.end();
    } else {
      response.writeHead(200, {"Content-Type": "image/png"});
      response.write(file, "binary");
      response.end();
    }
  });
}

exports.start = start;
exports.upload = upload;
exports.show = show;
```

Замечательно. Следующий шаг — немного более сложный, конечно. Первая проблема следующая: мы хотим обрабатывать загрузку файлов в нашем обработчике запроса *upload*, и тут надо будет передать объект*request* при вызове *form.parse* модуля *node-formidable*.

Но всё, что у нас есть — это объект *response* и массив *postData*. Грустно. Похоже, что придётся передавать каждый раз объект *request* из сервера в роутер и обработчик запроса. Может быть, имеется более элегантное решение, но этот способ может делать работу уже сейчас.

Давайте полностью удалим всё, что касается *postData* в нашем сервере и обработчиках запроса — он нам не нужен для обработки загрузки файла и, мало того, — даже создает проблему: мы уже «поглотили» события *data*объекта *request* в сервере, а следовательно, *form.parse*, которому так же надо поглащать эти события, не сможет получить больше данных (потому что Node.js не буферизирует данные).

Начнём с *server.js* — удалим обработку postData и строку c*request.setEncoding* (node-formidable сам всё сделает) и передадим *request* в роутер:

```
var http = require("http");
var url = require("url");
```

```
function start(route, handle) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");
    route(handle, pathname, response, request);
  }

  http.createServer(onRequest).listen(8888);
  console.log("Server has started.");
}

exports.start = start;
```

Следующий — *router.js* — мы больше не передаём *postData*, а вместо этого передаём *request*:

```
function route(handle, pathname, response, request) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response, request);
  } else {
    console.log("No request handler found for " + pathname);
    response.writeHead(404, {"Content-Type": "text/html"});
    response.write("404 Not found");
    response.end();
  }
}

exports.route = route;
```

Теперь объект *request* может быть использован в функции обработчика запроса *upload*. *node-formidable* будет заниматься сохранением загруженного файла в локальный файл */tmp*, но, конечно, мы сами должны сделать, чтобы этот файл переименовывался в */tmp/test.png*. Да, мы придерживаемся действительно простых вещей и принимаем, что могут загружаться только PNG-картинки.

Имеется небольшая дополнительная сложность в логике переименования: Windows-реализации Node.js не нравится, когда пытаются переименовать существующий файл, вот почему нам необходимо удалять файл в случае ошибки.

Давайте добавим в *requestHandlers.js* код управления загрузкой файла и переименованием:

```
var querystring = require("querystring"),
        fs = require("fs"),
```

```
        formidable = require("formidable");

function start(response) {
  console.log("Request handler 'start' was called.");

  var body = '<html>'+
        '<head>'+
        '<meta http-equiv="Content-Type" '+
        'content="text/html; charset=UTF-8" />'+
        '</head>'+
        '<body>'+
        '<form action="/upload" enctype="multipart/form-data" '+
        'method="post">'+
        '<input type="file" name="upload" multiple="multiple">'+
        '<input type="submit" value="Upload file" />'+
        '</form>'+
        '</body>'+
        '</html>';

        response.writeHead(200, {"Content-Type": "text/html"});
        response.write(body);
        response.end();
}

function upload(response, request) {
  console.log("Request handler 'upload' was called.");

  var form = new formidable.IncomingForm();
  console.log("about to parse");
  form.parse(request, function(error, fields, files) {
        console.log("parsing done");

/* Возможна ошибка в Windows: попытка переименования уже существующего
файла */
        fs.rename(files.upload.path, "/tmp/test.png", function(err) {
          if (err) {
                fs.unlink("/tmp/test.png");
                fs.rename(files.upload.path, "/tmp/test.png");
          }
        });
        response.writeHead(200, {"Content-Type": "text/html"});
        response.write("received image:<br/>");
        response.write("<img src='/show' />");
        response.end();
  });
```

```
}

function show(response) {
  console.log("Request handler 'show' was called.");
  fs.readFile("/tmp/test.png", "binary", function(error, file) {
          if(error) {
            response.writeHead(500, {"Content-Type": "text/plain"});
            response.write(error + "\n");
            response.end();
          } else {
            response.writeHead(200, {"Content-Type": "image/png"});
            response.write(file, "binary");
            response.end();
          }
  });
}

exports.start = start;
exports.upload = upload;
exports.show = show;
```

Вот и всё. Перезапускаем сервер, и последний пункт нашей задачи реализован. Выбираем локальную PNG-картинку с диска, загружаем на сервер и видим её на нашей веб-страничке.