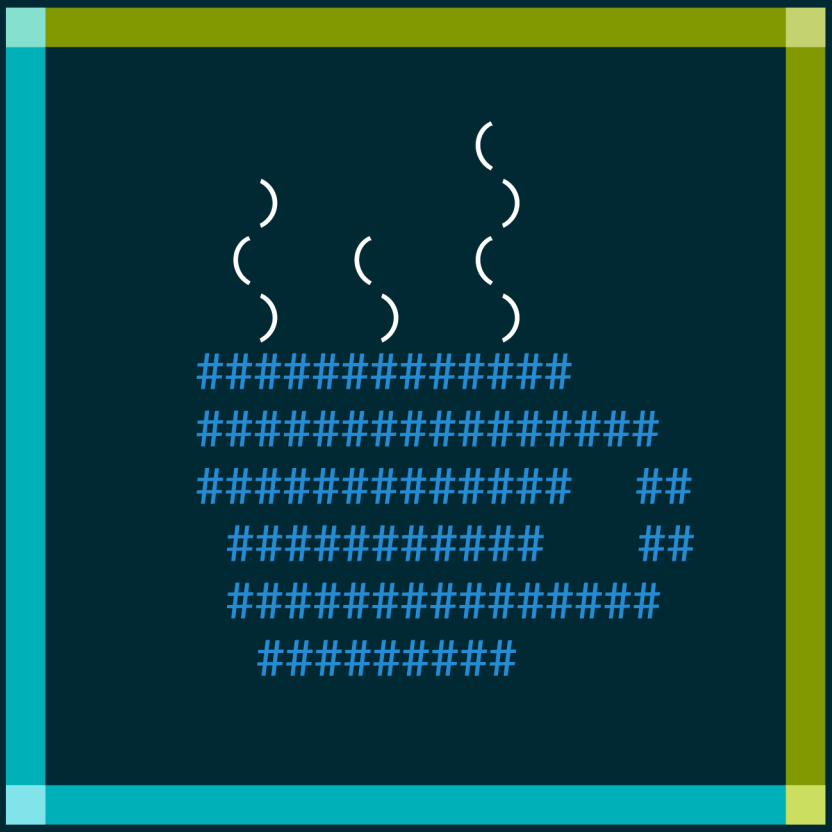
Начинай изучать Backbone.js, Node.js и MongoDB

[АЗАТ МАРДАН]
Перевод на русский Artod



БЫСТРОЕ ПРОТОТИПИРОВАНИЕ С JS

ГИБКАЯ РАЗРАБОТКА НА JAVASCRIPT

Быстрое Прототипирование с JS

Гибкая Разработка на JavaScript

Azat Mardan and Artod

This book is for sale at http://leanpub.com/rpjsru

This version was published on 2014-09-11

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Azat Mardan and Artod by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Начинай изучать Backbone.js, Node.js и MongoDB с @rpjsbook

The suggested hashtag for this book is #БПJS @rpjsbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#БПJS @rpjsbook

Also By Azat Mardan

Rapid Prototyping with JS

Oh My JS

Express.js Guide

JavaScript and Node FUNdamentals

Introduction to OAuth with Node.js

ProgWriter [programmer + writer]

Оглавление

Что говорят читатели . . . . . **i**

“Быстрое прототипирование с JS” в интернете . . . . . **ii**

Благодарность . . . . . **iii**

Введение . . . . . **iv**
Почему БПJS? iv
Чего ждать v
Для кого эта книга v
Чем эта книга не является v
Предпосылки vi
Как использовать эту книгу vi
Примеры vii
Обозначения viii
Термины viii

Об авторе . . . . . **ix**

I Быстрый старт **1**

1. Основы **2**
1.1 Front-End определения 2
1.2 Agile-методологии 12
1.3 Определения back-end 14

2. Установка **18**
2.1 Локальная настройка 18
2.2 Настройка облаков 34

II Front-end прототипирование **41**

3. jQuery и Parse.com **42**
3.1 Определения 42

3.2 jQuery . . . 45
3.3 Twitter Bootstrap . . . 46
3.4 LESS . . . 49
3.5 Пример использования стороннего API (Twitter) и jQuery . . . 53
3.6 Parse.com . . . 60
3.7 Chat с обзором Parse.com . . . 63
3.8 Chat с Parse.com: REST API и jQuery-версия . . . 64
3.9 Пуш на GitHub . . . 72
3.10 Развертка на Windows Azure . . . 74
3.11 Развертка на Heroku . . . 75
3.12 Обновление и удаление сообщений . . . 77

4. Введение в Backbone.js . . . 78
4.1 Настройка Backbone.js-приложения с нуля . . . 78
4.2 Работа с коллекциями . . . 83
4.3 Привязка событий . . . 88
4.4 Представления и подпредставления с Underscore.js . . . 93
4.5 Рефакторинг . . . 101
4.6 AMD и Require.js для разработки . . . 108
4.7 Require.js для продакшена . . . 116
4.8 Super Simple Backbone Starter Kit . . . 121

5. Backbone.js и Parse.com . . . 122
5.1 Chat с Parse.com: версия с JavaScript SDK и Backbone.js . . . 123
5.2 Развертка Chat на PaaS . . . 137
5.3 Усовершенствование Chat . . . 137

III Back-end прототипирование . . . 138

6. Node.js и MongoDB . . . 139
6.1 Node.js . . . 139
6.2 Chat: run-time версия . . . 146
6.3 Тесты для Chat . . . 146
6.4 MongoDB . . . 154
6.5 Chat: MongoDB-версия . . . 163

7. Собираем все вместе . . . 166
7.1 Разные домены развертывания . . . 166
7.2 Изменение конечных точек . . . 168
7.3 Приложение Chat . . . 171
7.4 Развертка . . . 173
7.5 Однодоменная развертка . . . 173

8. BONUS: Статьи Webapplog 176
8.1 Асинхронность в Node 176
8.2 MongoDB миграция с Monk 178
8.3 TDD в Node.js с Mocha 183
8.4 Wintersmith — генератор статических сайтов 186
8.5 Введение в Express.js: простое REST API приложение с Monk и MongoDB 189
8.6 Введение в Express.js: параметры, обработка ошибок и другие middleware . . . 193
8.7 JSON REST API сервер с Node.js и MongoDB с использованием Mongoskin и Express.js 199
8.8 Node.js MVC: Express.js + Derby “Hello World”-туториал 208

Заключение и что почитать далее 217
Заключение 217
Что почитать далее 218

Что говорят читатели

> “Туториал Азата имел решающее значение в разработке UI сайта Sidepon.com[1], в успехе от размещения нас на TheNextWeb.com[2] и полученной прибыли.” — Kenson Goo (Sidepon.com[3])

> “Мне очень понравилось читать эту книгу и разбирать примеры! Она демонстрирует и помогает вам открыть такое огромное количество технологий, что каждый должен попробывать их у себя в проекте.” — Chema Balsas

Быстрое прототипирование с JS успешно используеться в StartupMonthly[4] как учебное пособие[5]. Вот несколько отзывов наших учеников:

> “Спасибо большое всем и в особенности Азату и Юрию. Мне очень понравилась, мне хотелось усердно трудиться чтобы узнать эти технологии.” — Shelly Arora

> “Спасибо за включение этого семинара на выходные… то что мы сделали с Bootstrap + Parse было действительно быстро и здорово.” — Mariya Yao

> “Спасибо Юрию и вам всем. Это была отличная сессия — очень познавательная, она мне несомненно помогла освежить навыки в Javascript. С нетерпением жду встречи/работы с вами в будущем.” — Sam Sur

---

[1] http://Sidepon.com
[2] http://thenextweb.com
[3] http://Sidepon.com
[4] http://startupmonthly.org
[5] http://www.startupmonthly.org/rapid-prototyping-with-javascript-and-nodejs.html

“Быстрое прототипирование с JS” в интернете

Давайте дружить в Интернете

- Twitter: @RPJSbook[6] и @azat_co[7]
- Facebook: facebook.com/RapidPrototypingWithJS[8]
- Сайт: rapidprototypingwithjs.com[9]
- Блог: webapplog.com[10]
- GitHub: github.com/azat-co/rpjs[11]
- Storify: Rapid Prototyping with JS[12]

Другие способы связаться с нами

- Email: hi@rpjs.co[13]
- Google Group: rpjs@googlegroups.com[14] и https://groups.google.com/forum/#!forum/rpjs

Поделиться в Twitter

> “Я читаю «Быстрое прототипирование с JS» от @azat_co и @jechanceux #RPJS @RPJSbook” — http://ctt.ec/d738c

Перевод на русский

- По вопросам русского перевода пишите на gartod@gmail.com[15]

---

[6] https://twitter.com/rpjsbook
[7] https://twitter.com/azat_co
[8] https://www.facebook.com/RapidPrototypingWithJS
[9] http://rapidprototypingwithjs.com/
[10] http://webapplog.com
[11] https://github.com/azat-co/rpjs
[12] https://storify.com/azat_co/rapid-prototyping-with-js
[13] mailto:hi@rpjs.co
[14] mailto:rpjs@googlegroups.com
[15] mailto:gartod@gmail.com

Благодарность

Я благодарен моему литературному редактору David Moadel и техническому редактору Александру Власюку. Так же благодарен студентам (Hack Reactor[16], Marakana[17], pariSOMA[18] и General Assembly[19]), где при обучении я использовал “Быстрое прототипирование с JS” (или его части) как учебное пособие.

Я бы хотел поблагодарить команду StartupMonthly[20]: Юрия и Вадима за помощь с конструктивными предложениями касательно тренинга “Быстрое прототипирование с JS”[21] и мануала.

В добавок большная благодарнасть моим друзьям дизайнерам Бену, Ивану и Наталье, которые помогли мне с дизайном обложки.

[16] http://hackreactor.com
[17] http://marakana.com
[18] http://pariSOMA.com
[19] http://generalassemb.ly
[20] http://startupmonthly.org
[21] http://www.webapplog.com/training/

Введение

Резюме: причины написания этой книги; ответы на вопросы чего ждать и чего нет, что является предпосылками; предложения как использовать книгу и примеры; обьяснение формата обозначений в книге.

"Выбирайтесь из разработки." — Steve Blank[22]

"Быстрое прототипирование с JS" — это практичная книга которая знакомит вас с быстрым прототипированием программного обеспечения, используя передовые веб и мобильные технологии, включающие Node.js[23], MongoDB[24], Twitter Bootstrap[25], LESS[26], jQuery[27], Parse.com[28], Heroku[29] и другие.

Почему БПJS?

Эта книга была рождена чувством разочарования. Я занимался разработкой программного обеспечения уже много лет и когда я начал изучать Node.js и Backbone.js я понял, что их официальная документация и интернет нуждаются в кратких руководствах для быстрого старта и примерах. И разумеется, было практически невозможно найти все туториалы для современных JS-связаных технологий в одном месте.

Лучший способ *выучить* это *сделать*, не так ли? Для этого я использовал подход маленьких простых примеров, то есть руководств для быстрого старта, чтобы сразу погрузить себя в крутые новые технологии. После того, как я разделался с простыми приложениями мне понадобилась некоторая организованность. Я начал писать этот мануал в основном для себя, так я могу лучше понимать концепции и всегда иметь доступ к примерам. Тогда StartupMonthly[30] и я проводили несколько 2-хдневных интенсивов, помогая опытным разработчикам продвинуться по карьере с гибкой JavaScript разработкой. Руководство, которое мы использовали, много раз менялось на основе полученных отзывов. Конечный результат — эта книга.

[22] http://steveblank.com/
[23] http://nodejs.org
[24] http://mongodb.org
[25] http://twitter.github.com/bootstrap
[26] http://lesscss.org
[27] http://jquery.com
[28] http://parse.com
[29] http://heroku.com
[30] http://startupmonthly.org

Чего ждать

Типичному читатель БПJS следует ожидать коллекцию гайдов для быстрого старта, туториалов и советов (например, Git workflow). Много кода и совсем немного теории. Вся затронутая теория напрямую связана с некоторыми практическими аспектами и существенна для лучшего понимания технологий и специфических подходов в работе с ними, например, JSONP и кроссдоменные запросы.

В дополнение к примерам, в книге представлены все этапы установки и деплоя шаг за шагом.

На примере приложения Chat вы познакомитесь с веб/мобильными приложениями, начиная с front-end компонентов. Есть несколько версий этих приложений, но в итоге мы возьмем вместе front-end и back-end и задеплоим в производственную среду. Приложение Chat содержит все необходимые компоненты характерные для базового веб-приложения, так что мы дадим вам достаточно уверенности продолжить разработку самим, податься на новую работу/повышение или создать стартап!

Для кого эта книга

Книга расчитана на продвинуто-начального и среднего уровня веб-/мобильных разработчиков: это кто-то, кто был (или является) експертом в других языках типа Ruby on Rails, PHP, Perl, Python или/и Java. Это разработчики которые хотят узнать больше о JavaScript и Node.js методах для создания прототипов веб и мобильных приложений *быстро*. Наша целевая аудитория не имеет времени копаться в объемистой (или, еще хуже, крошечной) официальной документации. Цель "*Быстрое прототипирование с JS*" не сделать из читателя эксперта, а помочь ему/ей начать создавать приложение как можно быстрее.

В *"Быстрое прототипирование с JS: гибкая разработка на JavaScript"*, как можно понять из названия, речь пойдет о преобразовании вашей идеи в функциональный прототип в форме веб или мобильного приложения как можно быстрее. Мы придерживаемся Lean Startup[31] методологии, поэтому книга будет полезна не только для основателей стартапов, но и для работников больших компаний, особенно если они планируют добавить несколько новых скиллов в свои резюме.

Чем эта книга не является

"Быстрое прототипирование с JS" не является **ни** всеобъемлющей книгой по нескольким фреймворкам, библиотекам или технологиям (или просто по конкретной), **ни** справкой с советами и рекомендациями по веб-разработке. Примеры, приведенные в этой книге, могли бы быть *публично* доступны в Интернете.

Более того, если вы не знакомы с фундаментальными понятиями программирования, такими как циклы, if/else операторами, массивами, хэшами, объектами и функциями —

[31] http://theleanstartup.com

Быстрое прототипирование с JS вам в этом не поможет. К тому же, вам было бы сложно разобраться в наших примерах.

Множество томов больших книг написано на основополагающие темы (список подобных источников приведен в конце книги в главе *Что почитать далее*). Цель *Быстрое прототипирование с JS* — дать вам гибкие инструменты без повторения теориии программирования и компьютерной науки.

Предпосылки

Мы рекомендуем следующие вещи, чтобы полностью извлечь пользу из наших примеров и материалов:

- Знание основных понятий программирования, таких как объекты, функции, структуры данных (массивы, хэши), циклы (for, while), условия (if/else, switch)
- Базовые навыки веб-разработки, включающие, но не ограниченные, HTML и CSS
- Mac OS X или UNIX/Linux системы настоятельно рекомендованы для примеров этой книги и для веб-разработки в целом, хотя вполне возможно проломиться и на Windows-системах
- Доступ в Интернет
- 5-20 часов времени
- Какие-нибудь облачные сервисы, требующие информации о кредитной/дебетовой карте пользователя, даже для бесплатных аккаунтов

Как использовать эту книгу

Цифровая версия книги поставляется в 3 форматах:

1. PDF: подходит для распечатки; открывается Adobe Reader, Mac OS X Preview, iOS приложениями и другими программами для просмотра PDF.
2. ePub: подходит для приложения iBook на iPad и других iOS-устройств; для копирования на устройство используется iTunes, Dropbox или email себе.
3. mobi: подходит для Kindles всех поколений так же как для десктопного и мобильного приложения Amazon Kindle и Amazon Cloud Reader; для копирования на устройство используеться Whispernet, USB-кабель или email себе.

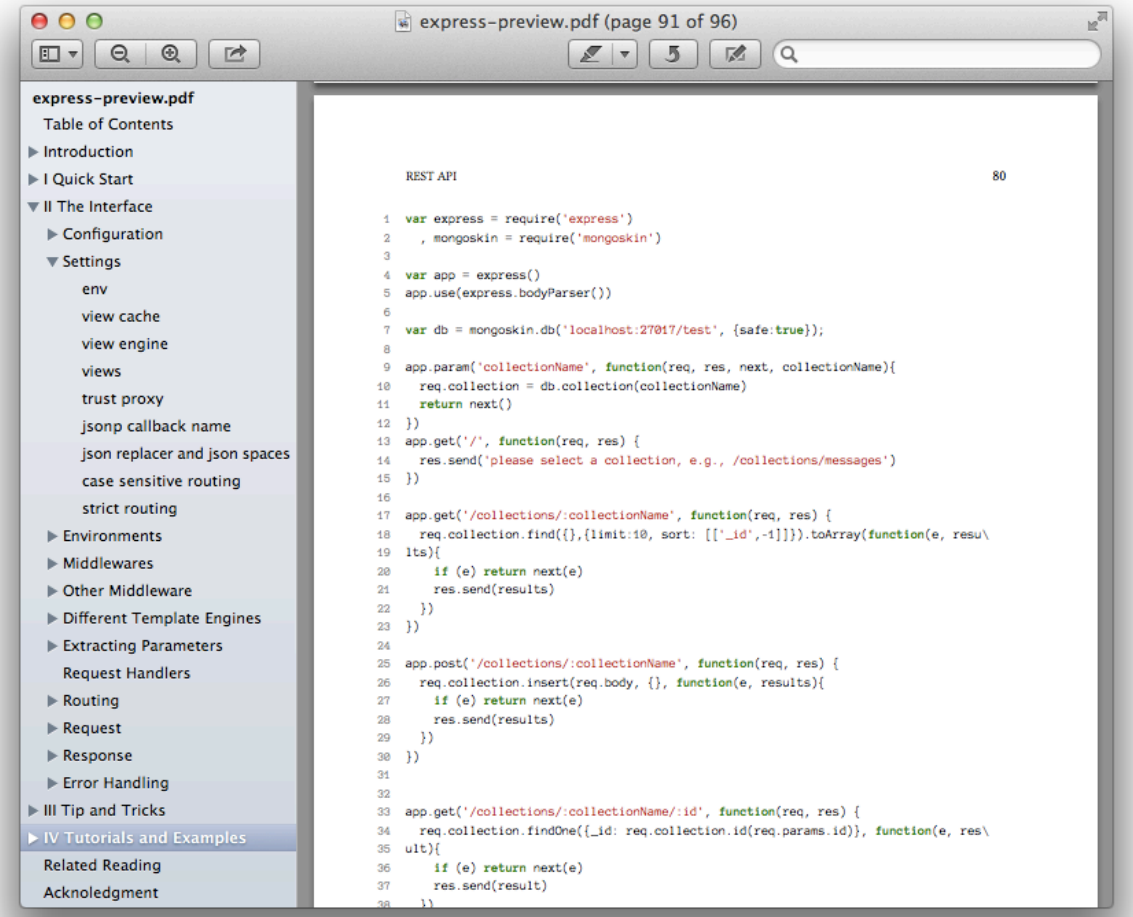
Это цифровая версия книги, так что большинство ссылок спрятано как на большинстве веб-страницах, например, jQuery[32] вместо http://jquery.com. В PDF-версии урлы находятся в сносках внизу страницы. Оглавление имеет локальные гиперссылки, которые позволят вам перепрыгивать на разные части или главы книги.

Имеются резюме в начале каждой главы, описывающие несколькими короткими предложениями какие примеры и темы затронуты в данной главе.

[32] http://jquery.com

В PDF, EPUB и Mobi версиях вы можете использовать **Оглавление**, которое находится в начале книги, имеющее внутренние ссылки для навигации по наиболее интересным частям или главам.

Для быстрой навигации между частями, главами и разделами книги, пожалуйста пользуйтесь панелью навигации книги, которая основана на **Оглавлении** (скриншот ниже).



Панель оглавления в приложении Mac OS X Preview

Примеры

Все исходные коды для примеров, использованые в этой книге, по большей части доступны в самой книге, так же как и в публичном GitHub-репозитории github.com/azat-co/rpjs[33]. Вы

[33] http://github.com/azat-co/rpjs

можете так же загрузить файлы как ZIP-архив[34] или использовать Git чтобы запуллить их. Подробнее о том, как установить и использовать Git будет рассказано позже в этой книге. Исходные коды файлов, структуры папок и файлы развёртывания предполагаются работоспособными локально и/или удалённо на PaaS-решениях, то есть Windows Azure и Heroku с минимальными изменения или вовсе без них.

Исходный код в книге технически ограничен платформой по ширине около 70 символов. Мы сделали все возможное, чтобы сохранить наилучший JavaScript и HTML стиль форматирования, но время от времени вы можете увидеть обратный слеш (\). В коде нет ничего неверного. Обратный слэш — это символ экранирования строки и если вы скопипастите код в редактор, пример должен нормально работать. Пожалуйста, отметьте себе, что код на GitHub и в книге может отличаться форматированием. Так же, дайте нам знать по email (hi@rpjs.co[35]) если нашли баги!

Обозначения

Вот так выглядят блоки исходного кода:

```
var object = {};
object.name = "Bob";
```

Терминальные команды выглядят так же, но начинаются со знака доллара $:

```
$ git push origin heroku
$ cd /etc/
$ ls
```

Названия файлов, имена путей/папок, цитаты и специальные слова/имена *выделены курсивом*, в то время как имена команд, например **mongod**, и акцентированиые слова, например **Отметьте**, **выделены жирным**.

Термины

Для целей этой книги мы используем некоторые взаимозаменяемые термины, в то время как, в зависимости от контекста, они могут означать разные вещи. Например, функция = метод = вызов, атрибут = свойство = член = ключ, значение = переменная, объект = хэш = класс, список = массив, фреймворк = библиотека = модуль.

[34] https://github.com/azat-co/rpjs/archive/master.zip
[35] mailto:hi@rpjs.co

Об авторе

Азат Мардан: инженер-программист, автор и йог.

Азат Мардан имеет свыше 12 лет опыта в разработке веб, мобильного и другого программного обеспечения. Со степенью бакалавра в информатике и степенью мастера наук в области технологий информационных систем, Азат обладает глубокими академическими знаниями, а так же обширным практическим опытом.

Сейчас Азат работает в качестве старшего инженера-программиста в DocuSign[36], где его команда работает над продуктом, которым пользуется 50 миллионов пользователей (веб приложение DocuSign) используя Node.js, Express.js, Backbone.js, CoffeeScript, Jade, Stylus и Redis.

Раньше он работал в качестве инженера в агрегаторе новостей социальных медиа Storify.com[37], (приобретенный LiveFyre[38]), который используется BBC, NBC, CNN, The White House и другими. В Storify все работает на Node.js в отличии от других компаний. Storify осуществляет поддержку open-source библиотеки jade-browser[39].

До этого Азат работал в качестве CTO/соучредителя в Gizmo[40] — предприятие на облачной платформе для мобильных маркетинговых компаний, выигравшее престижную 500 Startups[41] программу бизнес-акселерации.

Еще раньше Азат разрабатывал ответственные приложения для правительственных агентств в Вашингтоне, округ Колумбия, включая National Institutes of Health[42], National Center for Biotechnology Information[43], и Federal Deposit Insurance Corporation[44], а так же

[36] http://docusign.com
[37] http://storify.com
[38] http://livefyre.com
[39] http://npmjs.org/jade-browser
[40] http://www.crunchbase.com/company/gizmo
[41] http://500.co/
[42] http://nih.gov
[43] http://ncbi.nlm.nih.gov
[44] http://fdic.gov

Lockheed Martin[45].

Азат является частым участником конференций и хакатонов в Bay Area, финалист ‘12 хакатона (AngelHack[46] с командой FashionMetric.com[47]).

В дополнение, Азат преподает в General Assembly[48], Hack Reactor[49], pariSOMA[50] и Marakana[51] (приобретенная Twitter) и добился признания.

В свободное время он пишет о технологиях в своем блоге webAppLog.com[52], который является номером один[53] в “туториалах по express.js” по результатам Google-поиска. Азат является так же автором “Руководство по Express.js[54]”, “Быстрое прототипирование с JS[55]” и “Oh My JS[56]”; создателем open-source Node.js-проектов, включая ExpressWorks[57], mongoui[58] и HackHall[59].

Давайте дружить в Интернете

- Twitter: @RPJSbook[60] и @azat_co[61]
- Facebook: facebook.com/RapidPrototypingWithJS[62]
- Сайт: rapidprototypingwithjs.com[63]
- Блог: webapplog.com[64]
- GitHub: github.com/azat-co/rpjs[65]
- Storify: Rapid Prototyping with JS[66]

Другие способы связаться с нами

- Email: hi@rpjs.co[67]
- Google Group: rpjs@googlegroups.com[68] и https://groups.google.com/forum/#!forum/rpjs

[45] http://lockheedmartin.com
[46] http://angelhack.com
[47] http://fashionmetric.com
[48] http://generalassemb.ly
[49] http://hackreactor.com
[50] http://parisoma.com
[51] http://marakana.com
[52] http://webapplog.com
[53] http://expressjsguide.com/assets/img/expressjs-tutorial.png
[54] http://expressjsguide.com
[55] http://rpjs.co
[56] http://leanpub.com/ohmyjs
[57] http://npmjs.org/expressworks
[58] http://npmjs.org/mongoui
[59] http://hackhall.com
[60] https://twitter.com/rpjsbook
[61] https://twitter.com/azat_co
[62] https://www.facebook.com/RapidPrototypingWithJS
[63] http://rapidprototypingwithjs.com/
[64] http://webapplog.com
[65] https://github.com/azat-co/rpjs
[66] https://storify.com/azat_co/rapid-prototyping-with-js
[67] mailto:hi@rpjs.co
[68] mailto:rpjs@googlegroups.com

Поделиться в Twitter

> “Я читаю «Быстрое прототипирование с JS» от @azat_co и @jechanceux #RPJS @RPJSbook” — http://ctt.ec/d738c

Перевод на русский

- По вопросам русского перевода пишите на gartod@gmail.com[69]

[69] mailto:gartod@gmail.com

I Быстрый старт

1. Основы

Резюме: обзор HTML, CSS, и JavaScript синтаксиса; краткое введение в Agile-методологии; преимущества облачных вычислений, Node.js и MongoDB; описание HTTP запросов/ответов, RESTful API понятий.

> *“Я думаю, каждый должен учиться программировать, потому что это учит вас думать. Я вижу компьютерную науку как свободное искусство, которому каждый должен научиться.”* — Стив Джобс

1.1 Front-End определения

1.1.1 Общая картина

Общая картина разработки веб- и мобильных приложений состоит из следующих шагов

1. Юзер набирает URL или следует по ссылке в браузере (так же называемый клиентом)
2. Бразуер делает HTTP-запрос на сервер
3. Сервер обрабатывает запрос и, если имеются какие-либо параметры в строке запроса и/или в теле запроса, он принимает их во внимание
4. Сервер обновляет/берет/преобразовывает данные в базе данных
5. Сервер отправляет HTTP-ответ, содержащий данные в HTML, JSON или других форматах
6. Браузер получает HTTP-ответ
7. Браузер отображает HTTP-ответ юзеру в HTML-формате или в других форматах, например JPEG, XML, JSON

Мобильные приложения работают по такому же принципу, как обычный сайт, только вместо браузера используется свое приложение. Другие незначительные отличия включают: ограниченная пропускная способность данных, маленькие экраны и более эффективное использование локального хранилища.

Имеется несколько подходов в мобильной разработке, каждый со своими достоинствами и недостатками:

- Нативные iOS, Android, Blackberry приложения, написаные на Objective-C и Java

- Нативные приложения, написаные на JavaScript в Appcelerator[1] или с похожим инструментом, а затем скомпиленные в родной Objective-C или Java
- Мобильные сайты, сделанные с учетом маленьких экранов с респонсивным дизайном, с использованием CSS-фреймворков типа Twitter Bootstrap[2] или Foundation[3], обычного CSS или различных шаблонов
- HTML5-приложение, состоящее из HTML, CSS и JavaScript, и обычно созданное с фреймворками типа Sencha Touch[4], Trigger.io[5], JO[6], и затем обернутое в нативное приложение с PhoneGap[7]

1.1.2 Язык гипертекстовой разметки

Язык гипертекстовой разметки, или HTML, это не язык программирования сам по себе. Это набор тэгов разметки, которые описывают контент и представляют его в структурированном и форматированном виде. HTML-тэги состоят из **имени тега** внутри угловых скобок (<>). В большинстве случаев, тэги окружают контент закрывающим тегом, имеющим **прямой слэш** перед именем тэга.

В этом примере каждая строка представляет собой HTML-элемент:

```
<h2>Обзор HTML</h2>
<div>HTML — это ...</div>
<link rel="stylesheet" type="text/css" href="style.css" />
```

HTML-документ сам является элементом тэга *html* и все другие элементы являются потомками этого *html*-тэга:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <link rel="stylesheet" type="text/css" href="style.css" />
  </head>
  <body>
    <h2>Обзор HTML</h2>
    <p>HTML — это ...</p>
  </body>
</html>
```

Имеются различные виды и версии HTML, например, DHTML, XHTML 1.0, XHTML 1.1, XHTML 2, HTML 4, HTML 5. Эта статья хорошо объясняет различия между ними — Misunderstanding Markup: XHTML 2/HTML 5 Comic Strip[8].

[1] http://www.appcelerator.com/
[2] http://twitter.github.io/bootstrap/
[3] http://foundation.zurb.com/
[4] http://www.sencha.com/products/touch/
[5] https://trigger.io/
[6] http://joapp.com/
[7] http://phonegap.com/
[8] http://coding.smashingmagazine.com/2009/07/29/misunderstanding-markup-xhtml-2-comic-strip/

Любой HTMl-элемент может иметь атрибуты. Наиболее важные из них: class, id, style, data-name, onclick и другие атрибуты событий.

class

Атрибут class определяет класс, который используется для стилизации в CSS или для DOM-манипуляций, например:

```
<p class="normal">...</p>
```

id

Атрибут id определяет идентификатор, схож по целям на элемент class, но должен быть уникальным, например:

```
<div id="footer">...</div>
```

style

Атрибут style определяет встроенный CSS для стилизации элемента, например:

```
<font style="font-size:20px">...</font>
```

title

Атрибут title указывает дополнительную информацию, которая обычно показывается во всплывающих подсказках в большинстве браузерах, например:

```
<a title="Голосование за ответ">...</a>
```

data-name

Атрибут data-name позволяет хранить в DOM метаданные, например:

```
<tr data-token="fa10a70c-21ca-4e73-aaf5-d889c7263a0e">...</tr>
```

onclick

Атрибут onclick вызывает встроенный (inline) JavaScript-код, когда происходит клик на элементе, например:

```
<input type="button" onclick="validateForm();">...</a>
```

onmouseover

Атрибут onmouseover похож на `onclick`, но для события наведения курсора мыши на элемент, например:

```
<a onmouseover="javascript: this.setAttribute('css','color:red')">...</a>
```

Другие атрибуты HTML-элементов для встроенного JavaScript-кода:

- onfocus: когда браузер фокусируется на элементе
- onblur: когда браузер теряет фокус на элементе

- onkeydown: когда юзер нажимает на клавишу
- ondblclick: когда юзер делайет двойной клик
- onmousedown: когда юзер нажимает на клавишу мыши
- onmouseup: когда юзер отпускает клавишу мыши
- onmouseout: когда юзер убирает курсор мыши с элемента
- oncontextmenu: когда юзер вызывает контекстное меню

Полный список подобных событий и таблица браузерных совместимостей представлены в Event compatibility tables[9].

Мы будем активно использовать классы фреймворка Twitter Bootstrap. Покуда использование встроенного CSS и JavaScript-кода в основном считается **плохой** идеей, мы постараемся избегать этого. Однако, довольно полезно знать имена JavaScript-событий, потому что они используются повсеместно в jQuery, Backbone.js и конечно в чистом JavaScript. Чтобы конвертировать список атрибутов в список JS-событий, нужно просто откинуть префикс `on`, например, атрибут `onclick` означает событие `click`.

Больше информации доступно здесь: Example: Catching a mouse click[10], Википедия[11] и w3schools[12].

1.1.3 Каскадные таблицы стилей

Каскадные таблицы стилей, или CSS, — это способ форматирования и представления контента. HTML-документ может иметь внешние таблицы стилей, включенные в него посредством тэга **link**, как показано в предыдущих примерах, или может иметь CSS-код внутри тэга **style**:

```
<style>
  body {
    padding-top: 60px; /* 60px to make some space */
  }
</style>
```

Каждый HTML-элемент может иметь **id** и/или атрибут **class**:

```
<div id="main" class="large">
  Lorem ipsum dolor sit amet,
  Duis sit amet neque eu.
</div>
```

В CSS мы имеем доступ к элементам по их *id*, *class*, имени тэга и в некоторых крайних случаях по отношениям потомок/предок или значению атрибута.

Это устанавливает цвет всех параграфов (тэг **p**) в серый (`#999999`):

[9] http://www.quirksmode.org/dom/events/index.html
[10] https://developer.mozilla.org/en-US/docs/JavaScript/Getting_Started#Example:_Catching_a_mouse_click
[11] http://ru.wikipedia.org/wiki/HTML
[12] http://www.w3schools.com/html/html_intro.asp

```
p {
  color:#999999;
}
```

Это устанавливает внутренний отступ в элементе div с id main:

```
div#main {
  padding-bottom:2em;
  padding-top:3em;
}
```

Это устанавливает размер шрифта в 14 пикселов для всех элементов с классом `large`:

```
.large {
  font-size:14pt;
}
```

Это скрывает div, который является потомком элемента body:

```
body > div {
  display:none;
}
```

Это устанавливает ширину 150 пикселов для input с атрибутом name равным email:

```
input[name="email"] {
  width:150px;
}
```

Больше информации вы найдете тут: Википедия[13] и w3schools[14].

CSS3 — это улучшенный CSS, который включает в себя новые фишки, такие как закругление углов, borders и градиенты. Все это было возможно и в обычном CSS, но только при помощи PNG/GIF-картинок и других хитростей.

Для получения дополнительной информации смотрите CSS3.info[15], w3school[16] и статью о сравнении CSS3 и CSS на Smashing[17].

1.1.4 JavaScript

JavaScript был запущен в 1995 году в Netscape под названием LiveScript. JavaScript имеет такое же отношение к Java как лось к лососю. :-) В наше время JavaScript используется как на клиенте так и на веб-сервере, а так же и для разработки десктопных приложений.

Помещение JS-кода в тэг *script* является наиболее простым способом использовать JavaScript в HTML-документе:

[13] http://ru.wikipedia.org/wiki/CSS
[14] http://www.w3schools.com/css/
[15] http://css3.info
[16] http://www.w3schools.com/css3/default.asp
[17] http://coding.smashingmagazine.com/2011/04/21/css3-vs-css-a-speed-benchmark/

```
<script type="text/javascript" language="javascript">
  alert("Hello world!");
  //simple alert dialog window
</script>
```

Имейте ввиду, что смешивание HTML и JS-кода не есть хорошо, поэтому для их разделения мы можем переместить JS-код во внешний файл и подключить его посредством установки источника в атрибут `src="filename.js"` в тэге **script**, например, для *app.js*:

```
<script src="js/app.js" type="text/javascript" language="javascript">
</script>
```

Примечание

Закрывающий тэг `</script>` является обязательным даже с пустым элементом, который мы например использовали для подключения внешнего файла. Атрибуты **type** и **language** с течением времени стали необязательными из-за подавляющего господства JavaScript.

Другие способы запустить JavaScript включают:

- Прямое встравивание (inline) было уже рассмотрено выше
- Developer Tools WebKit-браузера и FireBug-консоль
- Интерактивная оболочка Node.js

Одним из преимуществ языка JavaScript является то, что он слабо типизированный. Эта слабая типизация, в отличие от строгой типизации[18] в таких языках, как C и Java, делает JavaScript лучшим языком для прототипирования. Вот некоторые из основных видов JavaScript объектов/классов (классов в чистом виде нету, объекты наследуют от объектов):

Числовые примитивы

Численные значения, например:

```
var num = 1;
```

Объект Number

Number[19] объект и его методы, например:

[18] http://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%B8_%D1%81%D0%BB%D0%B0%D0%B1%D0%B0%D1%8F_%D1%82%D0%B8%D0%BF%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F

[19] https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number

```
var numObj = new Number("123"); //объект Number
var num = numObj.valueOf(); //числовой примитив
var numStr = numObj.toString(); //строковое представление
```

Строковые примитивы

Последовательность символов внутри одинарных или двойных кавычек, например:

```
var str = "some string";
var newStr = "abcde".substr(1,2);
```

Для удобства, JS автоматически оборачивает строковые примитивы методами объекта String, но это не совсем то же самое[20].

Объект String

Объект String имеет большое количество полезных методов типа `length`, `match` и т.д., например:

```
var strObj = new String("abcde");//объект String
var str = strObj.valueOf(); //сртоковый примитив
strObj.match(/ab/);
str.match(/ab/); //оба вызова будут работать
```

Объект RegExp

Регулярные выражения или RegExps — это шаблоны символов, используемые для поиска совпадений или замены строк:

```
var pattern = /[A-Z]+/;
str.match(/ab/);
```

Специальные типы

Когда сомневаетесь, вы всегда можете вызвать `typeof obj`. Вот некоторые специальные типы, исопльзуемые в JS:

- NaN
- null
- undefined
- function

Глобальные

Вы можете вызвать эти методы в любом месте вашего кода, потому что это глобальные методы:

- decodeURI

[20] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String#Distinction_between_string_primitives_and_String_objects

- decodeURIComponent
- encodeURI
- encodeURIComponent
- eval
- isFinite
- isNaN
- parseFloat
- parseInt
- uneval
- Infinity
- Intl

JSON

JSON-библиотека позволяет нам парсить и сериализовать JavaScript-объекты, например:

```
var obj = JSON.parse('{a:1, b:"hi"}');
var stringObj = JSON.stringify({a:1,b:"hi"});
```

Объект Array

Массивы[21] — это индексные списки. Например, чтобы создать массив:

```
var arr = new Array();
var arr = ["apple", "orange", 'kiwi"];
```

Объект Arrary имеет множество прекрасных методов типа `indexOf`, `slice`, `join`. Убедитесь в том, что вы хорошо знакомы с ними, это поможет вам с экономить кучу времени в будущем.

Object

```
var obj = {name: "Gala", url:"img/gala100x100.jpg",price:129}
```

или

```
var obj = new Object();
```

Подробнее о модели наследования ниже.

Логические примитивы и объекты

Так же как со String и Number, Boolean[22] может быть примитивом и объектом.

[21] https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array

[22] https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Boolean

```
var bool1 = true;
var bool2 = false;
var boolObj = new Boolean(false);
```

Объект Date

Date[23] — это объект, позволяющий нам работать с датой и временем, например:

```
var timestamp = Date.now(); // 1368407802561
var d = new Date(); //Sun May 12 2013 18:17:11 GMT-0700 (PDT)
```

Объект Math

Математические константы и функции[24], например:

```
var x = Math.floor(3.4890);
var ran = Math.round(Math.random()*100);
```

Объект Browser

Дает нам доступ к браузеру и его свойствам типа URL, например:

```
window.location.href = 'http://rapidprototypingwithjs.com';
console.log("test");
```

Объект DOM

```
document.write("Hello World");
var table = document.createElement('table');
var main = document.getElementById('main');
```

Предупреждение

JavaScript поддерживает только числа размером до 53 битов. Существуют библиотеки для работы с большими числами если вам это нужно.

Полная справка по JavaScript и DOM-объектам доступна на Mozilla Developer Network[25] и w3school[26].

Ресурсы по JS, такие как спецификации ECMA, можно посмотреть здесь Ресурсы по JavaScript[27]. На момент написания статьи, последняя JavaScript-спецификация ECMA-262 Edition 5.1: PDF[28] и HTML[29].

Еще одним важным отличием JS является то, что это функциональный и прототипный язык. Типичный синтаксис для объявления функции выглядит следующим образом:

[23] https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date
[24] https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math
[25] https://developer.mozilla.org/ru/docs/Web/JavaScript
[26] http://www.w3schools.com/jsref/default.asp
[27] https://developer.mozilla.org/ru/docs/Web/JavaScript/Language_Resources
[28] http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
[29] http://www.ecma-international.org/ecma-262/5.1/

```
function Sum(a,b) {
  var sum = a+b;
  return sum;
}
console.log(Sum(1,2));
```

Функции в JavaScript являются гражданами первого класса[30] из-за функциональной[31] природы языка. Таким образом, функции могут быть использованы в качестве других переменных/объектов, например, функции могут быть переданы в другие функции в качестве аргументов:

```
var f = function(str1) {
  return function(str2) {
    return str1 + ' ' + str2;
  };
};
var a = f('hello');
var b = f('goodbye');
console.log( a('Catty') );
console.log( b('Doggy') );
```

Хорошо бы знать, что существует несколько способов для создания экземпляра объекта в JS:

- Классическая модель наследование[32]
- Псевдо классическая модель наследование[33]
- Функциональная модель наследования

Чтобы узнать больше по моделям наследования, почитайте Inheritance Patterns in JavaScript[34] и Inheritance revisited[35].

Более подробная информация о браузерном JavaScript доступна на Справочник по JavaScript 1.5[36], Википедия[37] и w3schools[38].

[30] http://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8_%D0%BF%D0%B5%D1%80%D0%B2%D0%BE%D0%B3%D0%BE_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%B0

[31] http://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5

[32] http://www.crockford.com/javascript/inheritance.html

[33] http://javascript.info/tutorial/pseudo-classical-pattern

[34] http://bolinfest.com/javascript/inheritance.php

[35] https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Inheritance_Revisited

[36] https://developer.mozilla.org/ru/docs/Web/JavaScript/%D0%A1%D0%BF%D1%80%D0%B0%D0%B2%D0%BE%D1%87%D0%BD%D0%B8%D0%BA_%D0%BF%D0%BE_JavaScript_1.5

[37] http://ru.wikipedia.org/wiki/JavaScript

[38] http://www.w3schools.com/js/default.asp

1.2 Agile-методологии

Гибкая методология разработки программного обеспечения (Agile-методология) эволюционировала по причине того, что традиционные методы, такие как Waterfall, не были достаточно хороши в ситуациях с высокой непредсказуемостью, например, когда решение неизвестно[39]. Agile-методология включает Scrum/Sprint, Test-Driven Development (TDD, разработка через тестирование), Continuous Deployment (непрерывная интеграция), парное программирование и другие техники, большинство которых заимствованы из экстремального программирования.

1.2.1 Scrum

В том, что касается управления, методология Agile использует Scrum подход. Подробнее о Scrum можно прочитать на:

- Scrum Guide in PDF[40]
- Scrum.org[41]
- Википедия[42]

Scrum-методология представляет собой последовательность коротких циклов, называемых **спринт**. Один спринт обычно длится от 1 до 2 недель. Типичный спринт начинается и кончается на спринт-совещании, где новые задачи назначаются членам команды. Новые задачи не могут быть добавлены в процессе спринта, только на спринт-совещании.

Важной частью методологии Scrum являются ежедневные **scrum**-совещания — отсюда и название (scrum англ. — схватка, драка). Каждая схватка — это 5-15 минутное совещание, которое часто проводится в прихожих. В Scrum-совещаниях каждый член команды отвечает на три вопроса:

1. Что ты сделал со вчерашнего дня?
2. Что ты собираешься делать сегодня?
3. Тебе что-нибудь надо от других членов команды?

Гибкость делает Agile более совершенным, чем Waterfall-методология, особенно в ситуациях высокой неопределенности, то есть в стартапах.

Преимущество методологии Scrum: эффективность там, где трудно планировать заранее, а также в ситуациях, когда обратная связь используется в качестве основного критерия при принятии решений.

[39] http://www.startuplessonslearned.com/2009/03/combining-agile-development-with.html
[40] http://www.scrum.org/storage/scrumguides/Scrum_Guide.pdf
[41] http://www.scrum.org/
[42] http://ru.wikipedia.org/wiki/Scrum

1.2.2 Test-Driven Development

Разработка через тестирование, или TDD, состоит из следующих шагов:

1. Написать непроходящий тест для новой фичи/задачи или улучшения, используя истинные или ложные утверждения.
2. Написать код, который успешно пройдет этот тест.
3. Отрефакторить код, если необходимо, и добавить новую функциональность так, чтобы тест оставался проходимым.
4. Повторить до тех пор, пока задача не будет выполнена.

Тесты могут быть разделены на функциональное и модульное тестирование. Последнее, это когда система тестирует отдельные модули, методы и функции с испытуемыми зависимостями, в то время как первое (так же известное, как интеграционное тестирование), это когда система тестирует кусок функциональности, включая зависимости.

Приемущества разработки через тестирования:

- Меньше багов/дефектов
- Большая эффективность кодовой базы
- Уверенность, что код работает и не ломает старой функциональности

1.2.3 Непрерывное развертывание и интеграция

Непрерывное развертывание, или CD (Continuous Deployment) — это множество методов для быстрой доставки новых фич, багфиксов и друхих улучшений клиентам. CD состоит из автоматического тестирования и автоматической развертки. Используя непрерывное развертывание, накладные расходы уменьшаются, временная петля обратной связи минимизируется. Проще говоря, чем раньше разработчик сможет получить обратную связь от клиента, тем раньше продукт сможет измениться, что ведет к приемуществу перед конкурентами. Многие стартапы разворачивают проект по несколько раз на дню, в сравнении с 6-12 месячным циклом, типичным для корпораций и крупных компаний.

Преимущества непрерывного развертывания: уменьшает временную петлю обратной связи и расходы при конечной эксплуатации продукта.

Разница между непрерывной разверткой и непрерывной интеграцией изложена в посте Continuous Delivery vs. Continuous Deployment vs. Continuous Integration — Wait huh?[43]

Некоторые наиболее популярные решения для непрерывной развертки:

- Jenkins[44]: Расширяемый с открытым исходным кодом сервер непрерывной интеграции
- CircleCI[45]: Доставка лучшего кода, быстрее
- Travis CI[46]: Сервис непрерывной интеграции для опенсорс-сообществ

[43] http://blog.assembla.com/assemblablog/tabid/12618/bid/92411/Continuous-Delivery-vs-Continuous-Deployment-vs-Continuous-Integration-Wait-huh.aspx

[44] http://jenkins-ci.org/

[45] https://circleci.com/

[46] https://travis-ci.org/

1.2.4 Парное программирование

Парное программирование — это метод, когда 2 разработчика работают вместе в одной среде. Один из них — ведущий, а другой — наблюдатель. Ведущий пишет код, а наблюдатель ассистирует, делая предложения. Потом они меняются ролями. Ведущий имеет больше тактическую роль фокусирования на текущей задаче. В отличие от этого, наблюдатель имеет больше стратегическую роль видения "всей картины" и нахождения багов и путей улучшить алгоритм.

Преимущества парного программирования:

- Парам характерна более короткая и эффективная кодовая база и меньшее количество багов и дефектов.
- В качестве дополнительного бонуса, программисты делятся знаниями друг с другом. Тем не менее, возможны конфликтные ситуации между разработчиками и случаются они не редко.

1.3 Определения back-end

1.3.1 Node.js

Node.js — это open-source событийно-ориентированная технология ввода/вывода для построения масштабируемых и эффективных веб-серверов. Node.js состоит из JavaScript-движка V8[47] от Google и поддерживаемый облачной компанией Joyent[48].

Цель и условия использования Node.js схожи с Twisted[49] для Python и EventMachine[50] для Ruby. Реализация Node на JavaScript была третьей после попытки использовать Ruby и C++.

Node.js сам по себе не является фреймворком как Ruby on Rails, его можно сравнить с парой PHP + Apache. Больше о Node.js-фреймворках будет сказано позже в главе *Node.js и MongoDB*

Преимущества использования Node.js:

- Разработчики имеют высокую вероятность быть знакомым с языком JavaScript по причине его статуса де-факто стандарта для веб и мобильной разработки.
- Один язык для разработки на front-end и back-end ускоряет процесс программирования. Девелоперским мозгам не приходиться переключаться на разные синтаксисы. Так называемое контекстное переключение. Изучение методов и классов идет быстрее.
- С Node.js вы могли бы прототипировать быстро и раньше выходить на рынок для изучения спроса и привлечения новых клиентов. Это важное конкурентное преимущество над компаниями, которые используют меньше гибких agile-технологий, например PHP и MySQL.

[47] http://ru.wikipedia.org/wiki/V8_(%D0%B4%D0%B2%D0%B8%D0%B6%D0%BE%D0%BA_JavaScript)
[48] http://joyent.com
[49] http://twistedmatrix.com/trac/
[50] http://rubyeventmachine.com/

- Node.js создан для поддержки приложений реального времени, используя веб-сокеты.

Для большей информации перейдите на Википедию[51], Nodejs.org[52] и статьи на ReadWrite[53] и O'Reilly[54].

О текущем состоянии Node.js на момент написания книги можно почитать на State of the Node slides by Isaac Z. Schlueter – 2013[55].

1.3.2 NoSQL и MongoDB

MongoDB, от hu**MONGO**us (англ. огромный) — это высокопроизводительная нереляционная система управления базами данных для огромных объемов данных. Понятие NoSQL пришло, когда традиционные системы управления базами данных, или СУБД, были не в состоянии разобраться с огромными объемами данных.

Преимущества использования MongoDB:

- Массштабируемость: по причине распределенной природы, несколько серверов и дата-центров могут размещать избыточные данные.
- Высокая производительность: MongoDB очень эффективна для хранения и получения данных, частично из-за отсутсвия связей между элементами и коллекциями в базе данных.
- Гибкость: хранение вида ключ-значение идеально для прототипирования, поскольку не требует от разработчика знать структуру, нет необходимости для фиксации моделей данных или комплексных миграций.

1.3.3 Облачные вычисления

Облачные вычисления состоят из:

- Инфраструктура как сервис (Infrastructure as s Service, IaaS), например, Rackspace, Amazon Web Services
- Платформа как сервис (Platform as a Service, PaaS), например, Heroku, Windows Azure
- Backend как сервис (Backend as a Service, BaaS — новенький, крутой малыш во дворе), например, Parse.com, Firebase
- Программное обеспечение как сервис (Software as a Service, SaaS), например, Google Apps, Salesforce.com

Платформы облачных приложений обеспечивают:

- Массштабируемость, например, генерация новых экземпляров в считанные минуты

[51] http://ru.wikipedia.org/wiki/Node.js
[52] http://nodejs.org/about/
[53] http://readwrite.com/2011/01/25/wait-whats-nodejs-good-for-aga
[54] http://radar.oreilly.com/2011/07/what-is-node.html
[55] http://j.mp/2013-state-of-the-node

- Простота развертки, например, для размещения кода в Heroku вы просто можете использовать `$ git push`
- Тарифные планы плати-походу, когда юзер добавляет или удаляет память и дисковое пространство по потребности
- Дополнения для простейшей инсталляции и конфигурации базы данных, серверы приложений, пакетов и т.д.
- Безопасность и поддержка

PaaS и BaaS идеальны для прототипирования, построения минимально жизнеспособных продуктов (minimal viable products, MVP) и стартапов на ранней стадии в основном.

Вот список наиболее популярных PaaS-решений:

- Heroku[56]
- Windows Azure[57]
- Nodejitsu[58]
- Nodester[59]

1.3.4 HTTP-запросы и ответы

Каждый HTTP-запрос и ответ состоит из следующих компонентов:

1. Заголовок: информация о кодировки, длине тела овтета, происхождение, тип контента и т.д.
2. Тело ответа: контент, обычно параметры или данные, переданные серверу или отправленные назад клиенту

Кроме того, HTTP-запрос содержит:

- Метод: существует несколько методов, наиболее распространенные из которых `GET`, `POST`, `PUT` и `DELETE`
- URL: хост, порт, путь, например, `https://graph.facebook.com/498424660219540`
- Строка запроса: все, что находится после знака вопроса в URL (e.g., `?q=rpjs&page=20`)

1.3.5 RESTful API

RESTful (REpresentational State Transfer, передача репрезентативного состояния) API стало популярным из-за необходимости в распределенных системах, когда каждая транзакция нуждается во включении достаточной информации о состоянии клиента. В некотором смысле, этот стандарт является лицом без гражданства, потому что никакой информации о состояниях клиентов на сервере не храниться, что позволяет любому запросу быть обслуженным другой системой.

Особые характеристики RESTful API:

[56] http://heroku.com
[57] http://windowsazure.com
[58] http://nodejitsu.com/
[59] http://nodester.com

- Обладает лучшей поддержкой масштабируемости благодаря тому, что различные компоненты могут быть независимо развернуты на различных серверах
- Заменил Simple Object Access Protocol (SOAP), по причине более простой структуры
- Использует HTTP-методы: GET, POST, DELETE, PUT, OPTIONS и т.д.

Вот пример REST API Create, Read, Update и Delete (CRUD) для коллекции Message:

Метод	**URL**	**Пояснение**
GET	/messages.json	Возвращает список сообщений в JSON-формате
PUT	/messages.json	Обновляет/заменяет все сообщения и возвращает статус/ошибку в JSON
POST	/messages.json	Создаёт новое сообщение и возвращает его id в JSON-формате
GET	/messages/{id}.json	Возвращает сообщение с id {id} в JSON-формате
PUT	/messages/{id}.json	Обновляет/заменяет сообщение с id {id}, если сообщение {id} не существует, то создает его
DELETE	/messages/{id}.json	Удаляет сообщение с id {id}, возвращает статус/ошибку в JSON-формате

REST является не протоколом, а архитектурой в том смысле, что он более гибок чем SOAP, являющийся протоколом. Таким образом, URL REST API могли бы выглядеть как `/messages/list.html` или `/messages/list.xml` в зависимости от того, хотим ли мы поддерживать эти форматы.

PUT и DELETE являются идемпотентными методами[60], это означает, что если сервер получает два или более похожих запроса, то конечный результат будет тем же.

GET является нуллипотентным, а POST неидемпотентными и может повлиять на состояние и быть причиной побочных эффектов.

Дополнительную информацию по REST API можно найти на Википедии[61] и A Brief Introduction to REST article[62].

[60] http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Idempotent_methods_and_web_applications
[61] http://ru.wikipedia.org/wiki/REST
[62] http://www.infoq.com/articles/rest-introduction

2. Установка

Резюме: своеты по инструментарию; пошаговая установка локальных компонентов; подготовка к использованию облачных сервисов.

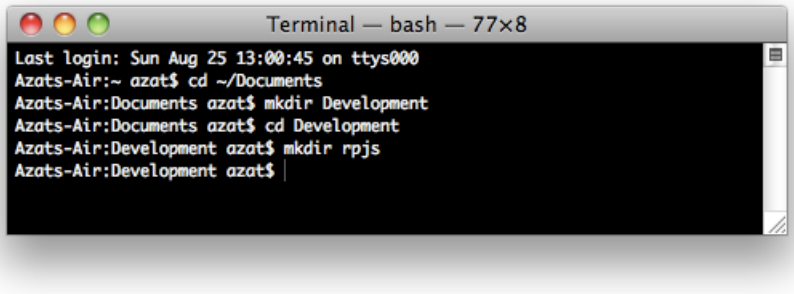
“Один из самых моих продуктивных дней был, когда я выбросил 1000 строк кода.”
— Кен Томпсон[1]

2.1 Локальная настройка

2.1.1 Папка для разработок

Если у вас нет специальной папки для ваших веб-проектов, то вы можете создать папку *Development* в папке *Documents* (путь будет *Documents/Development*). Чтобы работать с примерами кода, создайте папку *rpjs* внутри папки с разрабатываемыми проектами, например, если вы создадите папку *rpjs* внутри папки *Development*, путь будет *Documents/Development/rpjs*. Вы можете использовать Finder на Mac OS X или следующие терминальные команды на X/Linux системах:

```
$ cd ~/Documents
$ mkdir Development
$ cd Development
$ mkdir rpjs
```



Первоначальная настройка среды разработки.

[1] http://ru.wikipedia.org/wiki/%D0%A2%D0%BE%D0%BC%D0%BF%D1%81%D0%BE%D0%BD,_%D0%9A%D0%B5%D0%BD

Совет

Чтобы открыть приложение Mac OS Finder в текущей директории из терминала, просто наберите и запустите команду `$ open .`.

Чтобы получить список файлов и папок, исопльзуйте эту UNIX/Linix команду:

```
$ ls
```

или, чтобы показать скрытые файлы и папки типа *.git*:

```
$ ls -lah
```

Другая альтернатива `$ ls` это `$ ls -alt`. Разница между опциями `-lah` и `-alt` в том, что последний сортирует по хронологии, а первый по алфавиту.

Примечание

Вы можете использовать клавишу tab для автозаполнения имен файлов и папок.

Позже, вы сможете скопировать примеры в папку *rpjs*, а также создавать приложения в этой папке.

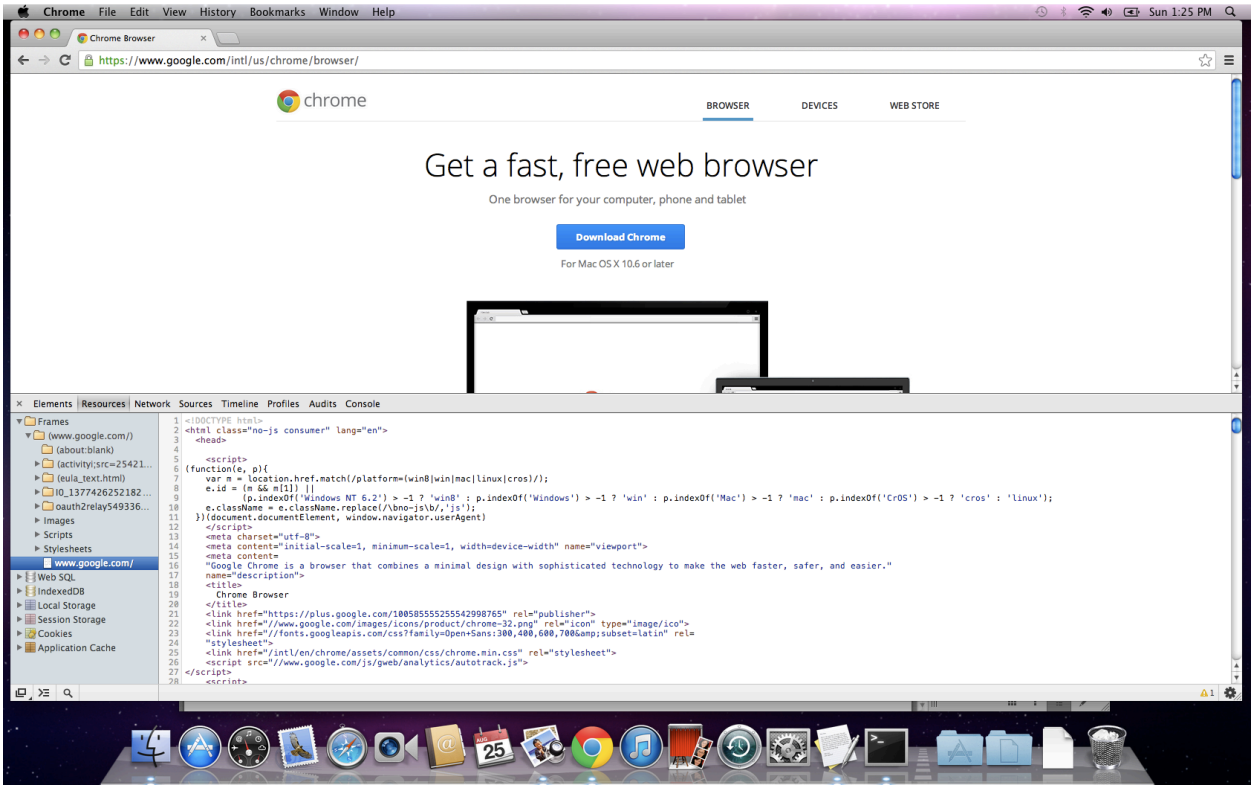
Примечание

Еще одна полезная вещь — иметь опцию “Новый терминал в папке” (“New Terminal at Folder”) в Finder на Mac OS X. Чтобы её включить, откройте ваши “Системные настройки” (“System Preferences”) (вы можете использовать Command + Space, так же известный как Spotlight). Найдите “Клавиатура” (“Keyboard”) и кликните на туда. Откройте “Горячие клавиши” (“Keyboard Shortcuts”) и кликните на “Сервисы” (“Services”). Проверьте опции “Новый терминал в папке” (“New Terminal at Folder”) и “Новый таб терминала в папке” (“New Terminal Tab at Folder”).

2.1.2 Браузеры

Мы рекомендуем загрузить WebKit[2] или Gecko[3] браузер на ваш выбор: Chrome[4], Safari[5] или Firefox[6]. В то время как Chrome и Safari уже поставляются с Developer Tools, для Firefox вам нужен плагин Firebug[7].

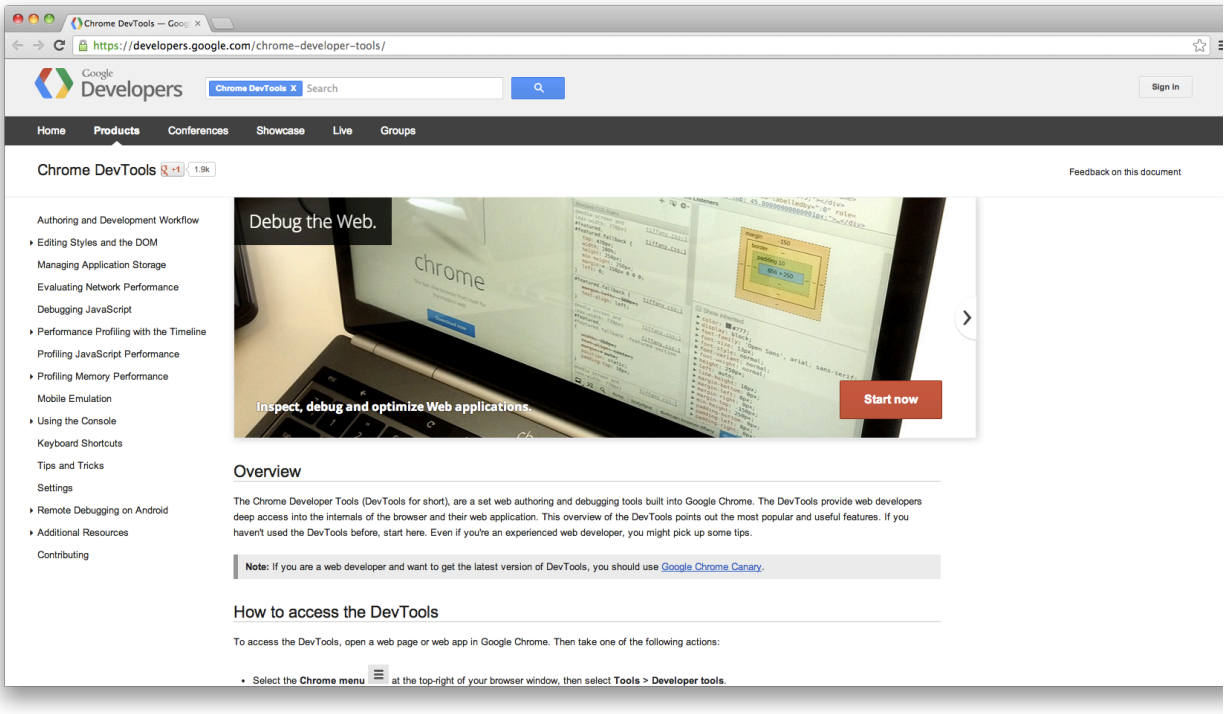
[2] http://ru.wikipedia.org/wiki/WebKit
[3] http://ru.wikipedia.org/wiki/Gecko
[4] http://www.google.com/chrome
[5] http://www.apple.com/safari/
[6] http://www.mozilla.org/en-US/firefox/new/
[7] http://getfirebug.com/



Средство разработки Chrome.

Firebug и Developer Tools позволяют делать разработчикам много вещей типа:

- Отлаживать JavaScript
- Манипулировать HTML- и DOM- элементами
- Измененять CSS на лету
- Мониторить HTTP- запросов и ответов
- Запускать профилирования и инспектирования хипдампов
- Смотреть загруженные активы, такие как изображения, CSS- и JS-файлы



Туториалы Google для освоения иструментов разработчика.

Хорошие туториалы по Chrome DevTools:

- Explore and Master Chrome DevTools[8] with Code School
- Chrome DevTools videos[9]
- Chrome DevTools overview[10]

[8] http://discover-devtools.codeschool.com/
[9] https://developers.google.com/chrome-developer-tools/docs/videos
[10] https://developers.google.com/chrome-developer-tools/

Освоение Chrome DevTools.

2.1.3 IDE и текстовые редакторы

Одна из лучших вещей в JavaScript это то, что вам не надо компилировать код. Потому что JS живет и работает в браузере, вы можете отлаживаться прямо здесь в браузере! Поэтому мы настоятельно рекомендуем легкие текстовые редакторы вместо тяжеловесных интегрированных средств разработки[11] или IDE (Integrated Development Environment). Но если вы уже привыкли к какой-либо IDE типа Eclipse[12], NetBeans[13] или Aptana[14], ничего страшного.

[11] http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8

[12] http://www.eclipse.org/

[13] http://netbeans.org/

[14] http://aptana.com/

Сайт Sublime Text.

Вот список самых популярных текстовых редакторов и интегрированных сред разработки, используемых в веб-разработке:

- TextMate[15]: только Mac OS X версия, 30-дневная триальная версия для v1.5, называемая *The Missing Editor for Mac OS X.*
- Sublime Text[16]: доступны версии для Mac OS X и Windows, даже лучшая альтернатива TextMate, неограниченный оценочный период.
- Coda[17]: редактор “всё в одном” с FTP-просмотрщиком, имеет поддержку для разработки с/на iPad.
- Aptana Studio[18]: полноразмерная IDE со встроенным терминалом и многими другими инструментами.
- Notepad ++[19]: бесплатный только для Windows легковесный редактор с поддержкой множества языков.
- WebStorm IDE[20]: многофункциональная IDE, позволяющая отладку Node.js; она разрабатывается JetBrains и позиционируется как *наиумнейшая JavaScript IDE.*

[15] http://macromates.com/
[16] http://www.sublimetext.com/
[17] http://panic.com/coda/
[18] http://aptana.com/
[19] http://notepad-plus-plus.org/
[20] http://www.jetbrains.com/webstorm/

Сайт WebStorm IDE.

2.1.4 Системы контроля версий

Система контроля версий[21] является обязательной даже если разработчик один. Так же множество облачных сервисов, например Heroku, требует Git для деплоя. Мы так же настоятельно рекомендуем привыкать к Git и к терминальным командам Git вместо использования визуальных Git-клиентов/приложений с графическим интерфейсом: GitX[22], Gitbox[23] или GitHub for Mac[24].

Subversion не является распределенной системой контроля версий. Эта статья показывает различия Git vs. Subversion[25].

Вот шаги для установки и настройки Git на вашу машину:

1. Загрузите последнюю версию для вашей OS с http://git-scm.com/downloads.

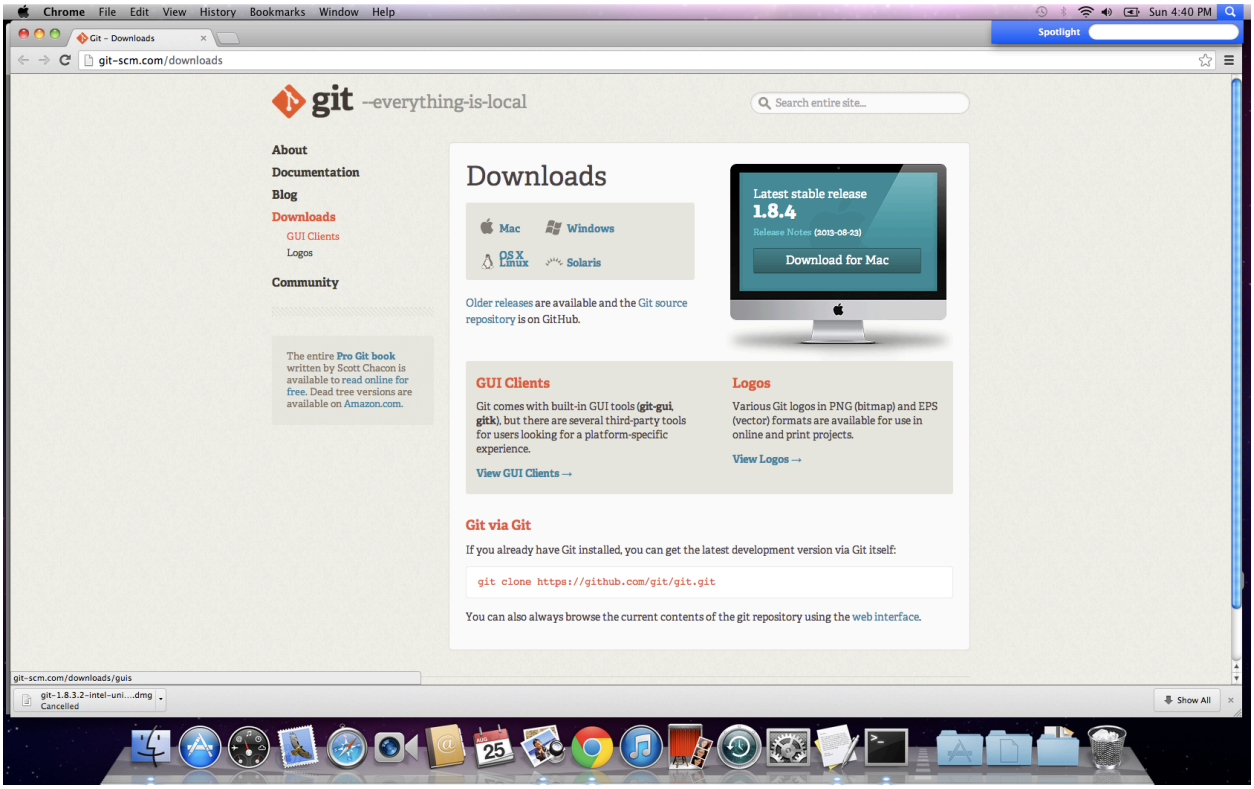
[21] http://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%8F%D0%BC%D0%B8

[22] http://gitx.frim.nl/

[23] http://www.gitboxapp.com/

[24] http://mac.github.com/

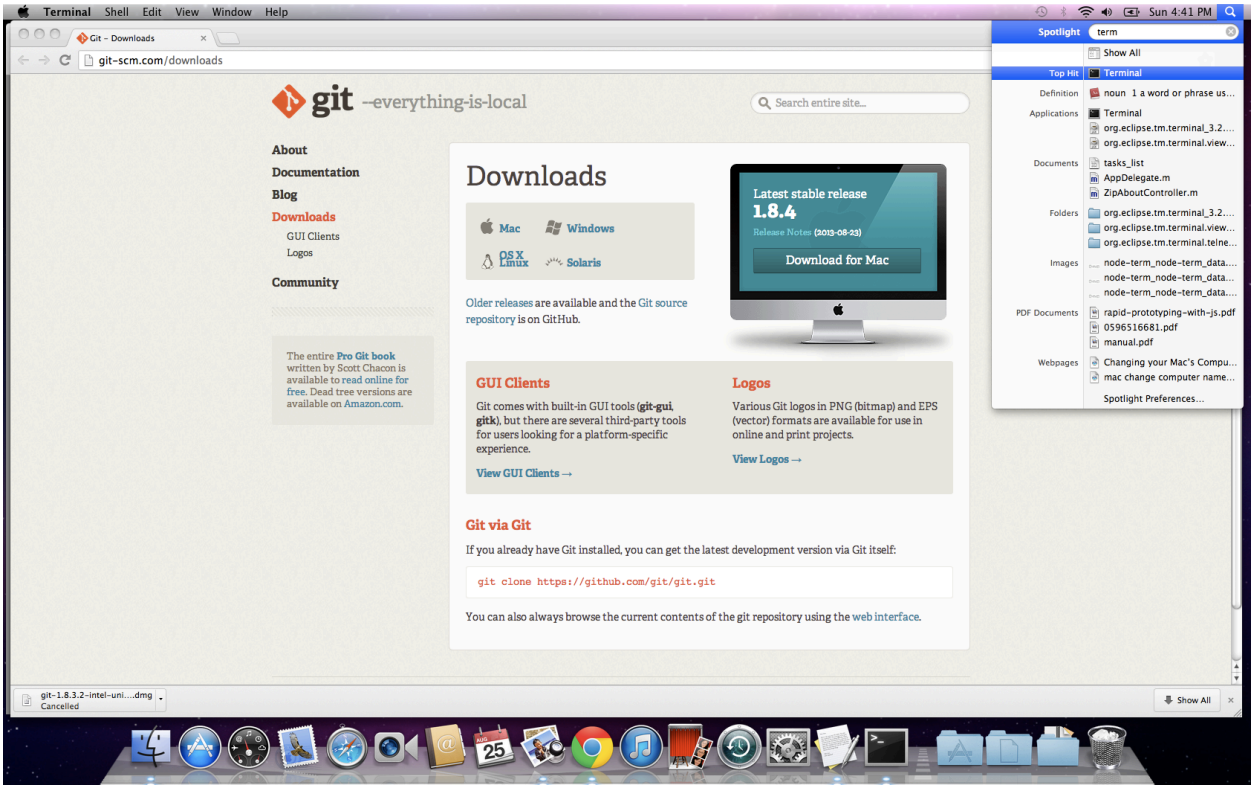
[25] https://git.wiki.kernel.org/index.php/GitSvnComparison



Загрузка последнего релиза Git.

2. Установите Git с загруженного `*.dmg`-пакета, то есть запустите `*.pkg`-файл и следуйте указаниям мастера.
3. Найдите приложение терминала, используя Command + Space (Spotlight) (пожалуйста, смотрите скриншот ниже) на OS X. На Windows вы можете использовать PuTTY[26] или Cygwin[27].

[26] http://www.chiark.greenend.org.uk/~sgtatham/putty/
[27] http://www.cygwin.com/



Использование Spotlight, чтобы найти и запустить приложение.

4. В вашем терминале введите эти команды, **заменяя** “John Doe” и johndoe@example.com вашими именем и email:

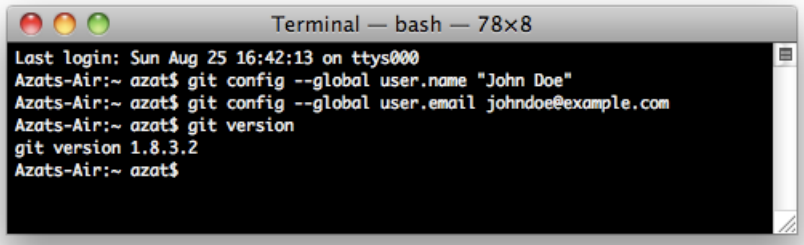
```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

5. Для проверки установки, запустите команду:

```
$ git version
```

6. Вы должны увидеть в вашем окне терминала нечто типа этого(ваша версия может отличаться; в нашем случае это 1.8.3.2):

```
git version 1.8.3.2
```



Конфигурирование и проверка инсталляции Git.

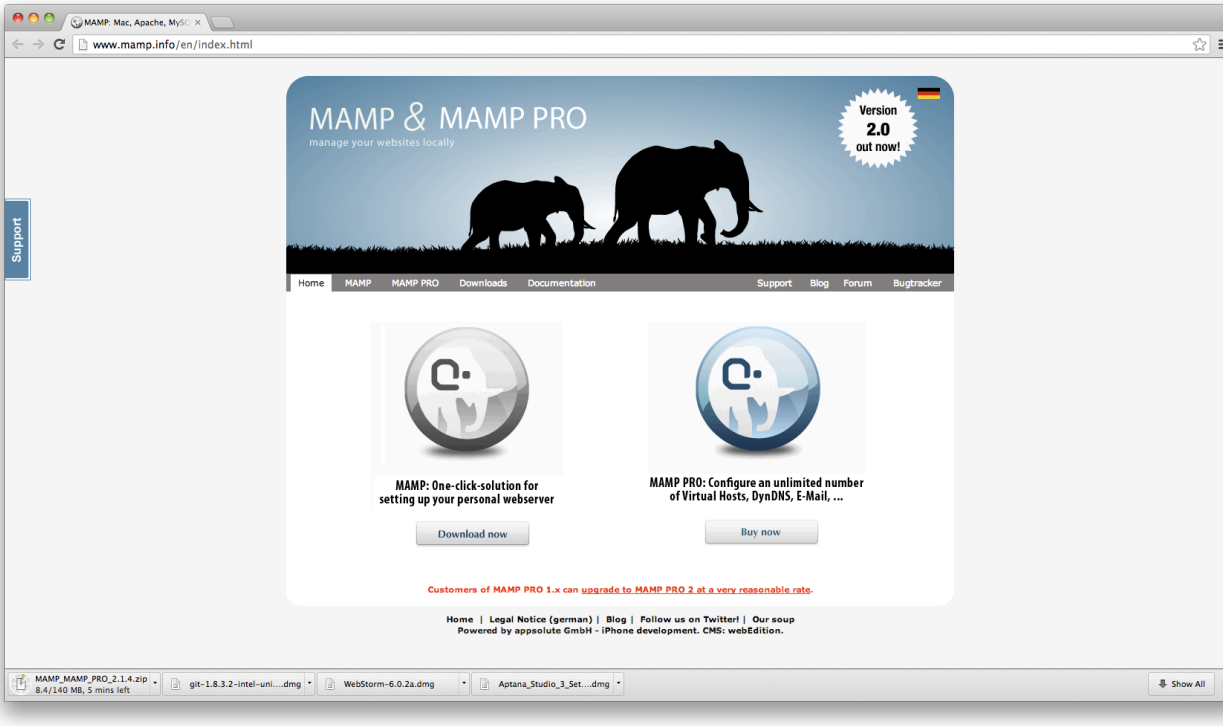
Генерация SSH-ключа и загрузка его на SaaS/PaaS-сайты будут разссмотренны позже.

2.1.5 Локальный HTTP-сервер

Пока вы можете делать большую часть front-end разработки без локального HTTP-сервера, он необходим только для загрузки файлов с вызовами HTTP-запросов/AJAX. Кроме того, это просто хорошая практика в целом использовать локальный HTTP-сервер. В таком случае, ваша среда разработки ближе к боевой среде на сколько это возможно. Вы можете рассмотреть следующие модификации веб-сервера Apache:

- MAMP[28]: Mac, Apache, MySQL, PHP персональный веб-сервер для Mac OS X
- MAMP Stack[29]: Mac-приложение с PHP, Apache, MySQL и phpMyAdmin, сборка от BitNami (Apple app store[30])
- XAMPP[31]: Apache-дистрибутив, содержащий MySQL, PHP и Perl для Windows, Mac, Linux и Solaris.

[28] http://www.mamp.info/en/index.html
[29] http://bitnami.com/stack/mamp
[30] https://itunes.apple.com/ru/app/mamp-stack/id571310406
[31] http://www.apachefriends.org/en/xampp.html



Страница MAMP for Mac.

MAMP, MAMP Stack и XAMPP имеют интуитивный графический интерфейс (GUI), который позволяет вам изменять конфигурации и настройки файла хостов.

Примечание

Node.js, как и многие другие back-end технологии, имеет собственный сервер для разработки.

2.1.6 База данных: MongoDB

Следующие шаги больше подходят для Mac OS X/Linux систем, но с некоторыми модификациями может быть использован для Windows-систем, то есть переменная $PATH — шаг #3. Ниже мы описали установку MongoDB из официального пакета, потому что мы считаем что этот подход более здравый и приводит к меньшим конфликтам. Тем не менее, есть много других способов установки на Mac[32], например использование Brew, а так же на других системах[33].

1. MongoDB можно загрузить с http://www.mongodb.org/downloads. Для последних ноутбуков Apple, типа MacBook Air, выберите версию OS X 64-bit. Владельцы старых маков

[32] http://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x/

[33] http://docs.mongodb.org/manual/installation/

должны выбрать ссылку http://dl.mongodb.org/dl/osx/i386.

Совет

Чтобы выяснить тип архитектуры вашего процессора, введите `$ uname -p` в командной строке.

1. Распакуйте пакет в вашу папку веб-разработки (~/*Documents/Development* или другую). Если хотитет, можете инсталлировать MongoDB в папку */usr/local/mongodb*.
2. **Необязательный**: если хотите получить доступ в командам MongoDB везде в вашей системе, вам надо добавить путь **mongodb** в переменную $PATH. Для Mac OS X открыть файл *paths*:

```
sudo vi /etc/paths
```

или, если вы предпочитаете TextMate:

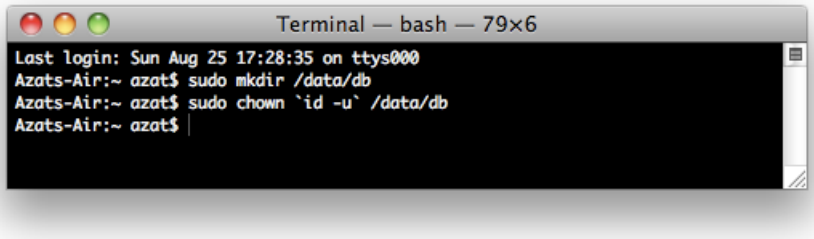
```
mate /etc/paths
```

и добавьте эту строчку в файл */etc/paths*:

```
/usr/local/mongodb/bin
```

3. Создайте папку для данных, по умолчанию MongoDB использует */data/db*. Пожалуйста отметье, что многое может отличаться в новых версиях MongoDB. чтобы создать её, введите и выполните следующие команды в терминал:

```
$ sudo mkdir -p /data/db
$ sudo chown `id -u` /data/db
```

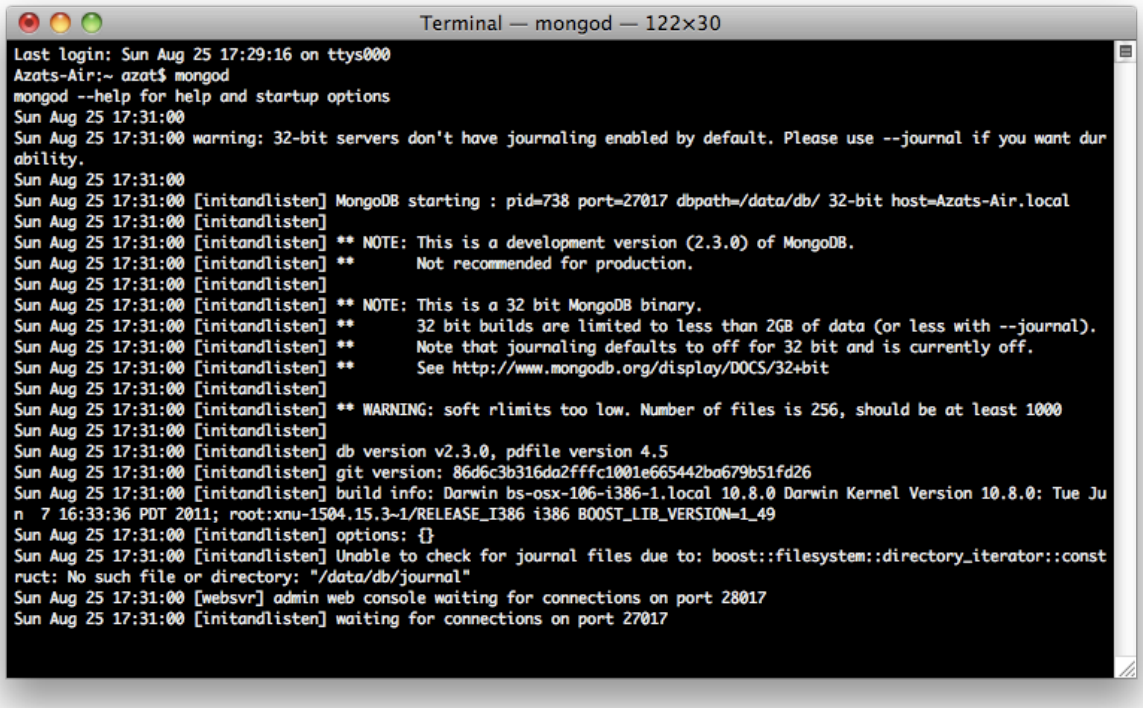


Начальная настройка MongoDB: создание папки данных.

Если вы предпочитаете использовать другой путь нежели */data/db*, вы можете указать его используя опцию –dbpath в **mongod** (главном сервисе MongoDB).

4. Перейдите в папку где вы распаковали MongoDB. Там должна быть папка *bin*. Оттуда введите следующую команду в вашем терминале:

```
$ ./bin/mongod
```



Запустите MongoDB-сервер.

5. Если вы видите нечто похожее на

```
MongoDB starting: pid =7218 port=27017...
```

это значит, что сервер базы данных MongoDB работает. По умолчанию он слушает http://localhost:27017. Если вы перейдете в ваш браузер и введете http://localhost:27017 вы должны увидеть номер версии, логи и другую полезную информацию. В данном случае сервер MongoDB использует **два** разных порта (27017 и 28017): первый — основной (родной) для общения с приложениями, а другой — это веб GUI для мониторинга/статистики. В нашем коде Node.js мы будем использовать только 27017.

Примечание

Не забудьте перезапустить окно терминала после добавления нового пути в переменную $PATH.

Теперь мы можем узнать, имеем ли мы доступ к консоли/оболочке MongoDB, которая будет действовать как клиент по отношению к серверу. Это значит, что мы должны будем держать окно терминала с сервером открытым и запущенным

1. Откройте другое окно терминала в той же папке и выполните:

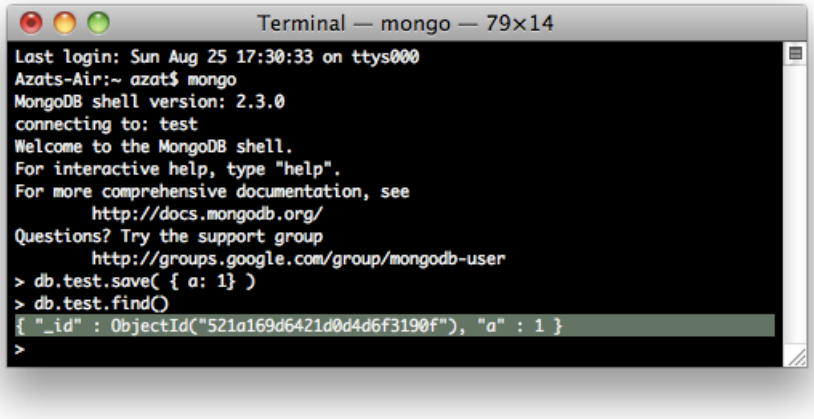
```
$ ./bin/mongo
```

Вы должны увидеть что-то вроде “MongoDB shell version 2.0.6 ...”

2. Затем введите и выполните:

```
> db.test.save( { a: 1 } )
> db.test.find()
```

Если вы видите, что ваша запись была сохранена, то все прошло хорошо:



Запуск клиента MongoDB и хранения образцов данных.

Команды *find* и *save* находят и сохраняют соответственно.

Подробные инструкции можно также получить на MongoDB.org: [Install MongoDB on OS X] (http://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x/). Для Windows есть хорошая статья: [Installing MongoDB] (http://www.tuanleaded.com/blog/2011/10/installing-mongodb/).

Примечание

Приложения MAMP и XAMPP поставляются с MySQL — традиционная база данных SQL с открытым исходным кодом, и phpMyAdmin — веб-интерфейс для базы данных MySQL.

Примечание

В Mac OS X (и большинстве систем Unix), чтобы закрыть процесс используется `control + c`. Если вы используете `control + z`, это положит процесс в сон (или отсоединит окно терминала), в этом случае вы можете в итоге заблокировать файлы данных и вам надо будет использовать команду kill или Activity Monitor и в ручную удалять залоченный файл в папке с данными. В ванильным Mac Terminal `command + .` является альтернативой `control + c`.

2.1.7 Другие компоненты

2.1.7.1 Установка Node.js

Node.js доступен на http://nodejs.org/#download (пожалуйста, смотрите скриншот ниже). Установка тривиальна, т.е., скачать архив, запустить пакет `*.pkg` установщика. Чтобы проверить установку Node.js можно ввести и выполнить:

```
1 $ node -v
```

Должно отобразиться нечто похожее на это (мы используем v.0.8.1, но ваша версия может отличаться):

```
1 v0.8.1
```

Пакет Node.js уже включает в себя Node Package Manager[34] (NPM). Мы будем активно использовать NPM для установки модулей Node.js.

[34] https://npmjs.org

Сайт Node.js.

2.1.7.2 JS-библиотеки

Front-end JavaScript библиотеки загружаются и распаковываются из соответствующих веб-сайтов. Эти файлы обычно размещают в папке Development (например, `~/Documents/Development`) для использования в будущем. Часто, есть выбор между минифицированной production-версией (подробнее об этом в разделах AMD и Require.js главы *Введение в Backbone.js*) и расширенной с комментариями.

Другой подход заключается в подключении ссылок на эти скрипты с CDN таких как Google Hosted Libraries[35], CDNJS[36], Microsoft Ajax Content Delivery Network[37] и другие. Поступая таким образом, приложения будут грузиться быстрее для некоторых пользователей, но не будет работать локально без интернета.

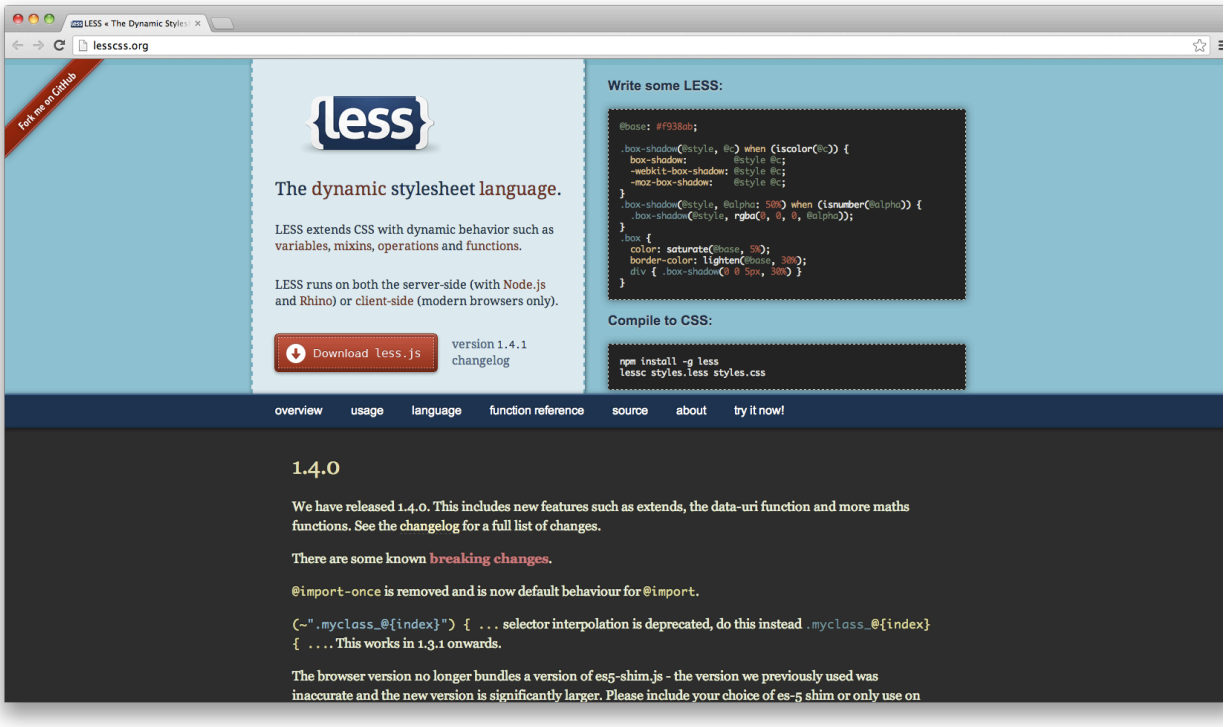
- Front-end интерпретатор LESS доступен на lesscss.org[38] — вы можете распаковать его в папку для разработок (*~/Documents/Development*) или в любую другую.
- Twitter Bootstrap — это CSS/LESS-фреймворк. Доступен на twitter.github.com/bootstrap[39].
- jQuery находиться на jquery.com[40].

[35] https://developers.google.com/speed/libraries/devguide
[36] http://cdnjs.com/
[37] http://www.asp.net/ajaxlibrary/cdn.ashx
[38] http://lesscss.org/
[39] http://twitter.github.com/bootstrap/
[40] http://jquery.com

- Backbone.js is на backbonejs.org[41].
- Underscore.js на underscorejs.org[42].
- Require.js на requirejs.org[43].

2.1.7.3 Приложение LESS

LESS App — это приложение Mac OS X для компиляции LESS в CSS на лету. Доступен на incident57.com/less[44].



Сайт LESS App для Mac.

2.2 Настройка облаков

2.2.1 Ключи SSH

Ключи SSH обеспечивают безопасное соединение без необходимости ввода имени пользователя и пароля. Для репозиториев GitHub последний подход используется с урлами HTTPS, например, “https://github.com/azat-co/rpjs.git, а первый с урлами SSH, например, git@github.com:azat-co/rpjs.git‘.

[41] http://backbonejs.org
[42] http://underscorejs.org
[43] http://requirejs.org
[44] http://incident57.com/less/

Для генерации SSH-ключей для GitHub на машинах Mac OS X/Unix выполните следующие действия:

1. Проверьте существующие SSH-ключи

```
$ cd ~/.ssh
$ ls -lah
```

2. Если вы видите какие-то файлы, такие как **id_rsa** (пожалуйста, посмотрите скриншот ниже для примера), вы можете удалить их или забэкапить в отдельную папку, используя следующие команды:

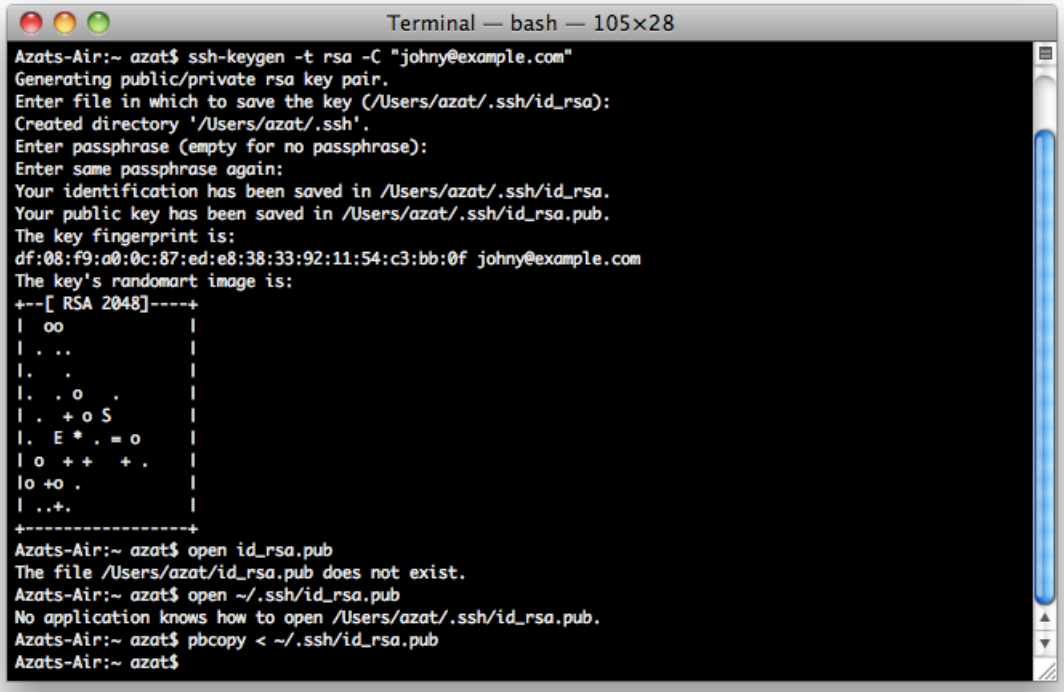
```
$ mkdir key_backup
$ cp id_rsa* key_backup
$ rm id_rsa*
```

3. Теперь мы можем сгенерировать пару SSH-ключа командой **ssh-keygen**, предполагая, что мы находимся в папке ~/.ssh:

```
$ ssh-keygen -t rsa -C "your_email@youremail.com"
```

4. Ответьте на вопросы (лучше оставить дефолтное имя: **id_rsa**). Затем скопируйте содержимое файла **id_rsa.pub** в буфер обмена:

```
$ pbcopy < ~/.ssh/id_rsa.pub
```



Генерация RSA-ключа для SSH и копирование открытого ключа в буфер обмена.

5. Или же, открываем файл **id_rsa.pub** в дефолтном редакторе:

```
$ open id_rsa.pub
```

6. Или в TextMate:

```
$ mate id_rsa.pub
```

2.2.2 GitHub

1. После копирования открытого ключа, перейдите на github.com[45], залогиньтесь, перейдите в настройки аккаунта, выберите “SSH key” и добавьте новый SSH-ключ. Назначьте имя, например, имя вашего компьютера, и вставьте значение вашего **публичного** ключа.
2. Чтобы проверить, есть ли у вас SSH-подключение к GitHub, введите и выполните следующую команду в вашем терминале:

```
$ ssh -T git@github.com
```

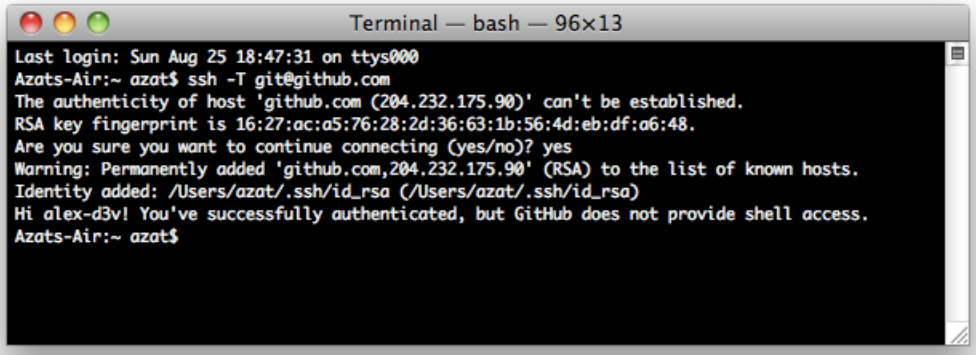
   Если вы увидите нечто вроде:

[45] http://github.com

```
Hi your-GitHub-username! You've successfully authenticated,
but GitHub does not provide shell access.
```

то все установлено.

3. Во время первого подключении к GitHub, вы можете получить предупреждение “authenticity of host ... can’t be established” (подлинность хоста... не может быть установлена). Пожалуйста, не пугайтесь такого сообщения — просто проследуйте дальше ответив ‘yes’, как показано на скриншоте ниже.



Тестирование SSH-соединение с GitHub для самого первого раза.

Если по некоторым причинам у вас другое сообщение, пожалуйста, повторите шаги 3-4 из предыдущего раздела *Ключи SSH* и/или повторно загрузите содержимое вашего файла `*.pub` в GitHub.

Предупреждение

Держите ваш файл **id_rsa** в секретном месте и никому не давайте!

Больше инструкций доступно на GitHub: Generating SSH Keys[46].
Пользователи Windows могут найти полезным функцию генерации SSH-ключа в [PuTTY].

2.2.3 Windows Azure

Вот шаги, чтобы настроить учетную запись Windows Azure:

1. Вам нужно зарегистрироваться для Windows Azure Web Site и Virtual Machine. В настоящее время у них есть 90-дневная бесплатная пробная версия http://azure.microsoft.com/ru-ru/.

[46] https://help.github.com/articles/generating-ssh-keys

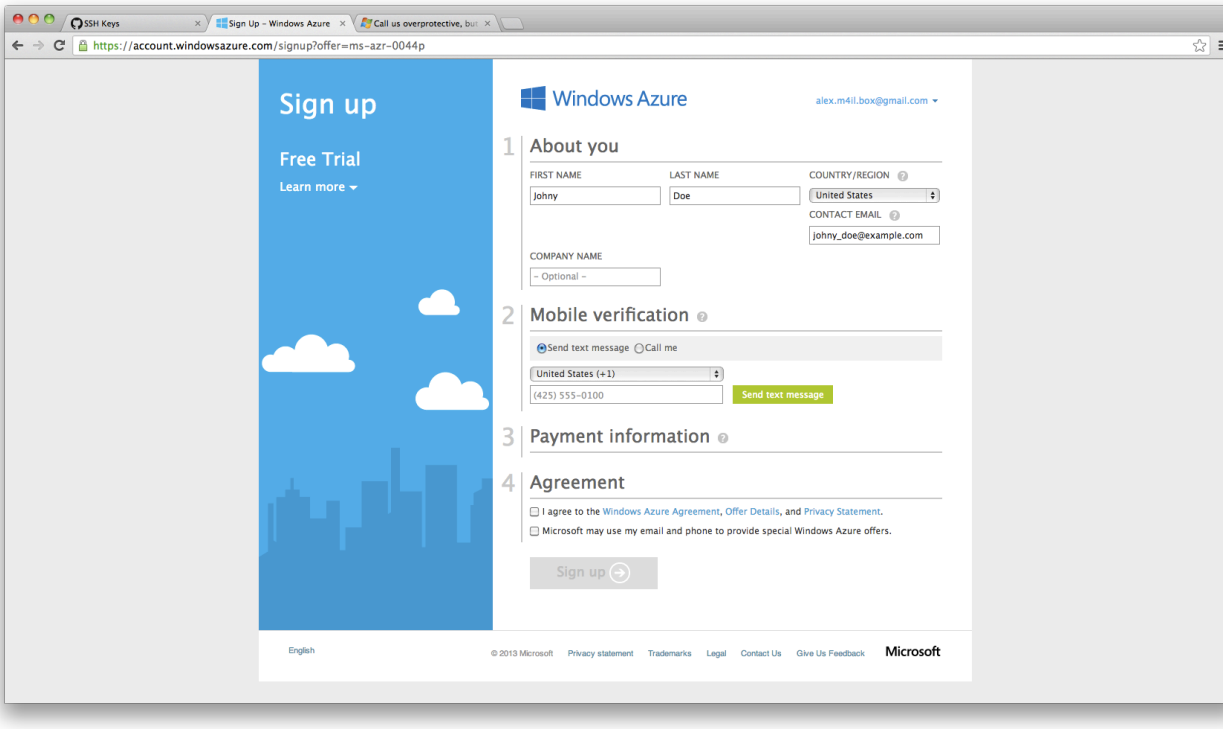
2. Разрешить Git Deployment и создать имя пользователя и пароль. Затем загрузить публичный SSH-ключ на Windows Azure.
3. Установить Node.js SDK, который доступен на https://www.windowsazure.com/en-us/develop/nodejs/.
4. Чтобы проверить установку введите:

```
$ azure -v
```

Вы должны увидеть нечто вроде:

```
Windows Azure: Microsoft's Cloud Platform... Tool Version 0.6.0
```

5. Залогиньтесь на Windows Azure Portal на https://windows.azure.com/.



Регситрация на Windows Azure.

6. Выберите “New”, затем выберите “Web Site”, “Quick Create”. Введите имя, которое будет использоваться в качестве ссылки для вашего сайта, и нажмите “OK”.
7. Перейдите в панель управления сайта и выберите “Set up Git publishing”. Придумайте имя пользователя и пароль. Эта комбинация может использоваться для развертывания на любой сайт вашей подписки, это означает, что вам не нужно устанавливать полномочия для каждого созданного сайта. Нажмите “OK”.
8. На следующем шаге должен отобразиться Git URL для перехода, что-то вроде

```
https://azatazure@azat.scm.azurewebsites.net/azat.git
```

и инструкции по развертыванию. Мы рассмотрим их позже.

9. **Опции для продвинутых пользователей**: следуйте этму туториалу для создания виртуальной машины и установки MongoDB на ней: Install MongoDB on a virtual machine running CentOS Linux in Windows Azure[47].

2.2.4 Heroku

Heroku — многофункциональная гибкая платформа для развертки приложений http://www.heroku.com/. Heroku работает как Windows Azure, в том смысле, что вы можете использовать Git для развертки приложений. Нет необходимости устанавливать Virtual Machine для MongoDB, потому что Heroku имеет MongoHQ add-on[48].

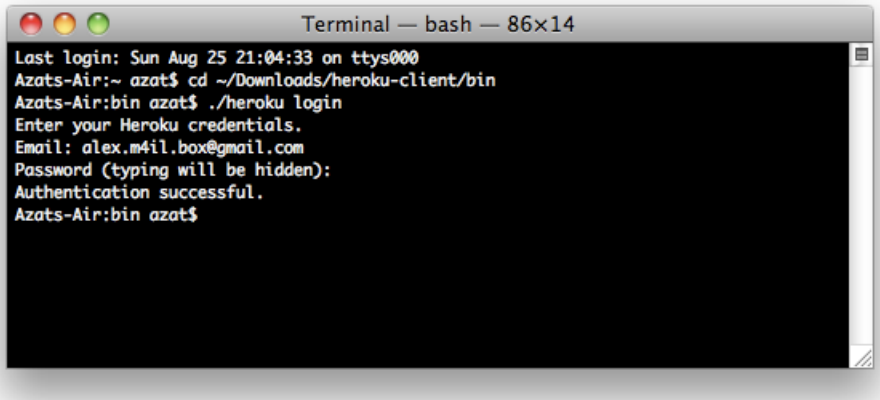
Чтобы настроить Heroku, выполните следующие действия:

1. Зарегистрируйтесь на http://heroku.com. В настоящее время у них есть бесплатный аккаунт, чтобы использовать его, установите все опции как minimum (0) и базы данных как shared.
2. Загрузите Heroku Toolbelt с https://toolbelt.heroku.com. Toolbelt — это пакет инструментов, т.е. библиотек, состоящих из Heroku, Git, и Foreman[49]. Для старых маков скачайте этот клиент[50] напрямую. Если вы используете другую OS, посмотрите Heroku Client GitHub[51].
3. После завершения установки, вы должны иметь доступ к команде **heroku**. Чтобы проверить это и войти в Heroku, введите:

```
$ heroku login
```

Вас спросят ваши полномочия на Heroku (имя пользователя и пароль) и, если вы уже создали SSH-ключ, он автоматически загрузит его на веб-сайте Heroku:

[47] https://www.windowsazure.com/en-us/manage/linux/common-tasks/mongodb-on-a-linux-vm/
[48] https://addons.heroku.com/mongohq
[49] https://github.com/ddollar/foreman
[50] http://assets.heroku.com/heroku-client/heroku-client.tgz
[51] https://github.com/heroku/heroku



Ответ на успешное выполнение команды `$ heroku login`.

4. Если все прошло хорошо, для создания приложения Heroku внутри вашей конкретной папки проекта, вы должны моч запустить:

```
1   $ heroku create
```

Более подробные пошаговые инструкции доступны на Heroku: Quickstart[52] и Heroku: Node.js[53].

2.2.5 Cloud9

Cloud9 — это браузерная IDE, с которой, с помощью вашей учетной записи GitHub или BitBucket, вы можете просматривать ваши репозитории, редактировать их и развертывать на Windows Azure или других сервисах. Не требуется никакой установки, все работает в браузере как Google Docs.

[52] https://devcenter.heroku.com/articles/quickstart
[53] https://devcenter.heroku.com/articles/nodejs

II Front-end прототипирование

3. jQuery и Parse.com

Резюме: обзор основных функций JQuery, скафолдинга Twitter Bootstrap, основных компонентов LESS; определения JSON, AJAX и CORS; иллюстрации JSONP-вызовов на примере Twitter REST API; объяснения того, как создать приложение Chat чисто на front-end с JQuery и Parse.com; пошаговые инструкции по развертыванию на Heroku и Windows Azure.

> *"Есть два способа построения дизайна программного обеспечения: один способ — сделать его настолько простым, что там, очевидно, не будет недостатков, и другой способ — сделать его настолько сложным, что там не будет очевидных недостатков. Первый метод намного труднее."* — Чарльз Энтони Ричард Хоар[1]

3.1 Определения

3.1.1 JavaScript Object Notation

Вот определение JavaScript Object Notation, или JSON, из json.org[2]:

> JavaScript Object Notation, или JSON — это легковесный формат данных для обмена. Прост для чтения и написания людьми. Прост для парсинга и генерации машинами. Он основан на подмножестве языка программирования JavaScript, Standard ECMA-262 3rd Edition — December 1999[3]. JSON — это текстовый формат, который является полностью независимым от языка, но использует соглашения, знакомые программистам языков C-семейства, включая C, C++, C#, Java, JavaScript, Perl, Python и многих других. Эти свойства делают JSON идеальным языком для обмена данными.

JSON стал стандартом для передачи данных между различными компонентами веб/-мобильных приложений и сторонних сервисов. JSON также широко используется внутри приложений в качестве формата для настроек, локалей, переводов файлов или любых других данных.

Типичные JSON-объект выглядит так:

[1] http://ru.wikipedia.org/wiki/%D0%A5%D0%BE%D0%B0%D1%80,_%D0%A7%D0%B0%D1%80%D0%BB%D1%8C%D0%B7_%D0%AD%D0%BD%D1%82%D0%BE%D0%BD%D0%B8_%D0%A0%D0%B8%D1%87%D0%B0%D1%80%D0%B4

[2] http://www.json.org/

[3] http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

```
{
  "a": "value of a",
  "b": "value of b"
}
```

Мы имеем объект с парами ключ-значение. Ключи находятся слева, а значения справа от двоеточия (:). В терминологии информатики, JSON эквивалентен хэш-таблице, индексированному списку или ассоциативному массиву (в зависимости от конкретного языка). Единственная большая разница между JSON и именно нотации JS-объекта (нативных JS-объектов) в том, что первые являются более строгими и требует двойные кавычки (") для ключей-идентификаторов и строковых значений. Оба типа могут быть сериализованы в строковое представление с `JSON.stringify()` и десериализованны с `JSON.parse()`, полагая, что мы имеем валидный JSON-объект в строковом формате.

Однако, каждый член объекта может быть массивом, примитивом или другим объектом, например:

```
  {
    "posts": [{
      "title": "Get your mind in shape!",
      "votes": 9,
      "comments": ["nice!", "good link"]
      }, {
      "title": "Yet another post",
      "votes":0,
      "comments": []
      }
    ],
    "totalPost":2,
    "getData": function () {
      return new Data().getDate();
    }
  }
```

В приведенном выше примере мы имеем объект со свойством **posts**. Значение свойства **posts** представляет собой массив объектов, каждый из которых имеет ключи **title**, **votes** и **comments**. Свойство votes содержит примитив number, а comments — это массив строк. Мы также можем иметь функцию в качестве значений, в этом случае ключ называется метод, например, `getData`.

JSON является гораздо более гибким и компактным, чем XML или другие форматы данных, как описано в этой статье — JSON: The Fat-Free Alternative to XML[4]. Удобно, что MongoDB использует схожий с JSON формат под названием BSON[5] или Binary JSON. Подробнее о BSON позже в главе *Node.js и MongoDB*.

[4] http://www.json.org/xml.html

[5] http://bsonspec.org/

3.1.2 AJAX

AJAX расшифровывается как Asynchronous JavaScript and XML и используется на стороне клиента (браузера), чтобы отправлять и получать данные с сервера, используя объект **XMLHttpRequest** языка JavaScript. Несмотря на название, использование XML не требуется и вместо него чаще используется JSON. Вот почему разработчики почти никогда не говорят больше AJAX. Имейте в виду, что HTTP-запросы могут быть сделаны синхронно, но это не очень хорошая практика. Наиболее типичный пример синхронного запроса — это включение тега **script**.

3.1.3 Кросс-доменные вызовы

Из соображений безопасности, начальная реализация объекта XMLHTTPRequest не позволяла кросс-доменные вызовы, т.е., когда клиентский код и серверный код находятся на разных доменах. Существуют методы, чтобы обойти эту проблему.

Одним из них является использование JSONP[6] — JSON с дополнением/префиксом. Это делается в основном динамическим (используя DOM-манипуляции) генерированием тега **script**. Теги script не подпадают под ограничение в один домен. JSONP-запрос включает имя функции обратного вызова в строке запроса. Например функция `jQuery.ajax()` автоматически генерирует уникальное имя функции и добавляет его в запрос (строка разбита на несколько строк для удобства чтения):

```
https://graph.facebook.com/search
  ?type=post
  &limit=20
  &q=Gatsby
  &callback=jQuery16207184716751798987_1368412972614&_=1368412984735
```

Второй подход — использование Cross-Origin Resource Sharing, или CORS[7], который является лучшим решением, но он требует контроля на сервере для изменения заголовков ответа. Мы будем использовать этот метод в финальной версии приложения Chat.

Пример CORS-заголовка ответа сервера:

```
Access-Control-Allow-Origin: *
```

Подробнее о CORS: Resources by Enable CORS [8] и [Using CORS by HTML5 Rocks Tutorials](http://www.html5rocks.com/en/tutorials/cors/). Тест CORS-запросов на test-cors.org[9].

[6] http://en.wikipedia.org/wiki/JSONP
[7] http://www.w3.org/TR/cors/
[8] http://enable-cors.org/resources.html
[9] http://client.cors-api.appspot.com/client

3.2 jQuery

В процессе тренинга мы будем использовать jQuery (http://jquery.com/) для манипуляций с DOM, HTTP-запросов и JSONP-вызовов. jQuery стало стандартом де-факто, благодаря его объекту/функции $, который обеспечивает простой, но эффективный способ доступа к любому HTML DOM-элементу на странице по его ID, class, имени тега, значению атрибута, структуре или любой комбинации таковых. Синтаксис очень похож на CSS, где мы используем # для id и . для выбора класса, напр.:

```
1 $("#main").hide();
2 $("p.large").attr("style","color:red");
3 $("#main").show().html("<div>new div</div>");
```

Вот список наиболее часто используемых функций jQuery API:

- **find()**[10]: выбирает элементы на основе предоставленной строки селектора
- **hide()**[11]: скрывает элемент, если он был виден
- **show()**[12]: показывает элемент, если он был скрыт
- **html()**[13]: получает или задает HTML внутри элемента
- **append()**[14] вставляет элемент в DOM после выбранного элемента
- **prepend()**[15] вставляет элемент в DOM перед выбранным элементом
- **on()**[16]: вешает слушатель события к элементу
- **off()**[17] убирает слушатель события от элемента
- **css()**[18]: получает или задает значение атрибута стиля элемента
- **attr()**[19] получает или задает любой атрибут элемента
- **val()**[20]: получает или задает атрибут value элемента
- **text()**[21]: получает сведенный текст элемента и его потомков
- **each()**[22]: перебирает набор совпавших элементов

Большинство jQuery-функций воздействуют не только на один элемент, на котором они вызываются, но и на набор соответствующих элементов в случае, если результат выборки

[10] http://api.jquery.com/find
[11] http://api.jquery.com/hide
[12] http://api.jquery.com/show
[13] http://api.jquery.com/html
[14] http://api.jquery.com/append
[15] http://api.jquery.com/prepend
[16] http://api.jquery.com/on
[17] http://api.jquery.com/off
[18] http://api.jquery.com/css
[19] http://api.jquery.com/attr
[20] http://api.jquery.com/val
[21] http://api.jquery.com/text
[22] http://api.jquery.com/each

имеет несколько элементов. Это одна из самых распространенных ошибок, которая приводит к багам, и это обычно происходит, когда jQuery-селектор слишком обобщенный.

Кроме того, в jQuery есть много плагинов и библиотек, напр., jQuery UI[23], jQuery Mobile[24], которые позволяют создавать расширенный пользовательский интерфейс или другие функциональные возможности.

3.3 Twitter Bootstrap

Twitter Bootstrap[25] — это набор CSS/LESS-правил и JavaScript-плагинов для создания хорошего пользовательского интерфейса, не тратя *много времени* на такие вещи, как закругление краев кнопок, кросс-совместимость, отзывчивость и др. Эта коллекция или фреймворк идеально подходит для быстрого прототипирования ваших идеи. Тем не менее, благодаря своей возжности кастомизации, Twitter Bootstrap — это также хороший фундамент для серьезных проектов. Исходный код написан на LESS[26], но так же можно загрузить и использовать обычный CSS.

Вот простой пример использования скаффолдинга Twitter Bootstrap. Структура проекта должна выглядеть так:

```
  /bootstrap
    -index.html
    /css
      -bootstrap.min.css
      ... (other files if needed)
    /img
      glyphicons-halflings.png
      ... (other files if needed)
```

Сначала давайте создадим файл **index.html** с соответствующими тегами:

```
<!DOCTYPE html>
<html lang="en">
  <head>

  </head>
  <body>
  </body>
</html>
```

Подключим библиотеку Twitter Bootstrap как сжатый CSS-файл:

[23] http://jqueryui.com/
[24] http://jquerymobile.com/
[25] http://twitter.github.com/bootstrap/
[26] http://lesscss.org/

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <link
      type="text/css"
      rel="stylesheet"
      href="css/bootstrap.min.css" />
  </head>
  <body>
  </body>
</html>
```

Добавим скелет с классами container-fluid и row-fluid:

```
  <body >
    <div class="container-fluid">
      <div class="row-fluid">
      </div>  <!-- row-fluid -->
    </div>  <!-- container-fluid -->
  </body>
```

Twitter Bootstrap использует 12-колоночную сетку. Размер индивидуальной ячейки может быть указан классом **spanN**, например, “span1”, “span2”, “span12”. Есть также классы **offsetN**, например, классы “offset1”, “offset2”, … “offset12” для перемещения ячеек вправо. Полный справочник доступен на http://twitter.github.com/bootstrap/scaffolding.html.

Мы будем использовать классы **span12** и **hero-unit** для основного блока контента:

```
    <div class="row-fluid">
     <div class="span12">
        <div id="content">
          <div class="row-fluid">
            <div class="span12">
              <div class="hero-unit">
                <h1>
                  Welcome to Super
                  Simple Backbone
                  Starter Kit
                </h1>
                <p>
                  This is your home page.
                  To edit it just modify
                  <i>index.html</i> file!
                </p>
                <p>
                  <a
```

```
                    class="btn btn-primary btn-large"
                    href="http://twitter.github.com/bootstrap"
                    target="_blank">
                    Learn more
                  </a>
                </p>
              </div>  <!-- hero-unit -->
            </div>  <!-- span12 -->
          </div> <!-- row-fluid -->
        </div>  <!-- content -->
      </div> <!-- span12 -->
      </div>  <!-- row-fluid -->
```

Полный исходный код **index.html** отсюда rpjs/boostrap[27]:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <link
    type="text/css"
    rel="stylesheet"
    href="css/bootstrap.min.css" />
</head>
  <body >
    <div class="container-fluid">
      <div class="row-fluid">
        <div class="span12">
          <div id="content">
            <div class="row-fluid">
              <div class="span12">
                <div class="hero-unit">
                  <h1>
                    Welcome to Super
                    Simple Backbone
                    Starter Kit
                  </h1>
                  <p>
                    This is your home page.
                    To edit it just modify
                    <i>index.html</i> file!
                  </p>
                  <p>
                    <a
                      class="btn btn-primary btn-large"
```

[27] https://github.com/azat-co/rpjs/tree/master/bootstrap

```
30                      href="http://twitter.github.com/bootstrap"
31                      target="_blank">
32                      Learn more
33                    </a>
34                  </p>
35                </div>  <!-- hero-unit -->
36              </div>  <!-- span12 -->
37            </div> <!-- row-fluid -->
38          </div>  <!-- content -->
39        </div> <!-- span12 -->
40      </div>  <!-- row-fluid -->
41    </div>  <!-- container-fluid -->
42  </body>
43  </html>
```

Этот пример доступен для скачивания/подкачки в публичном GitHub репозитории github.com/azat-co/rpjs[28] в папке rpjs/bootstrap[29].

Другие полезные инструменты — CSS-фреймворки и CSS-препроцессоры, которые стоит попробывать:

- Compass[30]: CSS-фреймворк
- SASS[31]: расширение CSS3, аналог LESS
- Blueprint[32]: CSS-фреймворк
- Foundation[33]: адаптивный front-end фреймворк
- Bootswatch[34]: коллекция настраиваемых тем Twitter Bootstrap
- WrapBootstrap[35]: market place для настраиваемых Bootstrap тем

3.4 LESS

LESS — это динамический язык стилей. Иногда, и в данном случае так и есть, меньше значит больше, а больше значит меньше[36]. :-)

Браузер не может интерпретировать синтаксис LESS, поэтому исходный код LESS должен быть скомпилирован в CSS одним из трех способов:

1. В браузере посредством JavaScript-библиотеки LESS[37]

[28] http://github.com/azat-co/rpjs
[29] https://github.com/azat-co/rpjs/tree/master/bootstrap
[30] http://compass-style.org/
[31] http://sass-lang.com/
[32] http://blueprintcss.org/
[33] http://foundation.zurb.com/
[34] http://bootswatch.com/
[35] https://wrapbootstrap.com/
[36] http://en.wikipedia.org/wiki/The_Paradox_of_Choice:_Why_More_Is_Less
[37] http://lesscss.googlecode.com/files/less-1.3.0.min.js

2. На стороне сервера языком программирования или фреймворком, например для Node.js есть LESS-модуль[38]
3. Локально на вашей Mac OS X машине LESS App[39], SimpLESS[40] или похожее приложение

Предупреждение

Первый вариант хорош для среды разработки, но неоптимален для production-среды.

LESS имеет переменные, миксины и операторы, которые позволяют повторно исопльзовать CSS-правила, что ускоряет разработку. Вот пример переменной:

3.4.1 Переменные

Переменные уменьшают избыточность и позволяют разработчикам быстро менять значения, потому что они находятся в одном каноническом место. А мы знаем, что в дизайне приходится менять значения *очень часто*!

Код LESS:

```
@color: #4D926F;
#header {
  color: @color;
}
h2 {
  color: @color;
}
```

Эквивалент в CSS:

```
#header {
  color: #4D926F;
}
h2 {
  color: #4D926F;
}
```

3.4.2 Миксины

Это синтаксис для миксинов (mixin англ. — примесь), которые работают как функции:

[38] https://npmjs.org/package/less
[39] http://incident57.com/less/
[40] http://wearekiss.com/simpless

```
.rounded-corners (@radius: 5px) {
  border-radius: @radius;
  -webkit-border-radius: @radius;
  -moz-border-radius: @radius;
}

#header {
  .rounded-corners;
}
#footer {
  .rounded-corners(10px);
}
```

Конвертируем это в CSS:

```
.rounded-corners (@radius: 5px) {
  border-radius: @radius;
  -webkit-border-radius: @radius;
  -moz-border-radius: @radius;
}

#header {
  border-radius: 5px;
  -webkit-border-radius: 5px;
  -moz-border-radius: 5px;
}
#footer {
  border-radius: 10px;
  -webkit-border-radius: 10px;
  -moz-border-radius: 10px;
}
```

Миксины могут быть использованы без параметров или с несколькими параметрами.

3.4.3 Операторы

С операторами мы можем выполнять математические функции с цифрами, цветами или переменными.

Пример оператора в LESS:

```
@the-border: 1px;
@base-color: #111;
@red:        #842210;

#header {
  color: @base-color * 3;
  border-left: @the-border;
  border-right: @the-border * 2;
}
#footer {
  color: @base-color + #003300;
  border-color: desaturate(@red, 10%);
}
```

Приведенный выше код компилируется в CSS:

```
@the-border: 1px;
@base-color: #111;
@red:        #842210;

#header {
  color: #333333;
  border-left: 1px;
  border-right: 2px;
}
#footer {
  color: #114411;
  border-color: #7d2717;
}
```

Как вы можете видеть, LESS значительно повышает возможность повторного использования простого CSS. Вот несоклько онлайн-инструментов для компиляции:

- LESS2CSS[41]: приятный браузерный конвертер LESS в CSS на Express.js
- lessphp[42]: онлайн демонстрационный компилятор
- Dopefly[43]: онлайн LESS-конвертер

Другие фичи LESS[44] включают:

- Шаблонизация
- Вложенные правила

[41] http://less2css.org/
[42] http://leafo.net/lessphp/
[43] http://www.dopefly.com/LESS-Converter/less-converter.html
[44] http://lesscss.org/#docs

- Функции
- Пространства имен
- Область видимости
- Комментарии
- Импорт

3.5 Пример использования стороннего API (Twitter) и jQuery

Пример **чисто** для демонстративных целей. Это не часть главного приложения Chat, описываемого в последующих главах. Цель состоит в том, чтобы просто проиллюстрировать комбинацию jQuery, JSONP и REST API технологии. Пожалуйста, просмотрите код и не пытайтесь запустить его, потому что Twitter отказался от API v1.0. Это приложение скорее всего **не будет работать в том виде, в котором оно есть**. Если вы все еще хотите запустить этот пример, делайте это на свой страх и риск, следуя инструкциям ниже. Скачайте его с GitHub или скопируйте его с PDF-версии.

Примечание

Этот пример был создан с использованием Twitter API v1.0 и может не работать с Twitter API v1.1, которая требует аутентификацию пользователя для вызовов REST API. Вы можете получить необходимые ключи на dev.twitter.com[45].

В этом примере, мы будем использовать функцию jQuery `$.ajax()`. Она имеет следующий синтаксис:

```
var request = $.ajax({
    url: url,
    dataType: "jsonp",
    data: {page:page, ...},
    jsonpCallback: "fetchData"+page,
    type: "GET"
});
```

В приведенном выше фрагменте кода функции `ajax()`, мы использовали следующие параметры:

- **url** — конечная точка API
- **dataType** — тип данных, которые мы ожидаем от сервера, например, “json”, “xml”
- **data** — данные для отправки на сервер

[45] https://dev.twitter.com

- **jsonpCallback** — имя функции в строковом формате, которая будет вызвана после ответа сервера
- **type** — HTTP-метод запроса, например, “GET”, “POST”

Больше параметров и примеров функции `ajax()` вы найдете на api.jquery.com/jQuery.ajax[46].

Чтобы привязать нашу функцию к пользовательскому событию, мы должны использовать функцию `click` библиотеки jQuery. Синтаксис очень прост:

```
$("#btn").click(function() {
...
}
```

`$("#btn")` является объектом jQuery, который указывает на HTML-элемент в объектной модели документа (Document Object Model или DOM) с **id** равным “btn”. HTML-код самой кнопки, с приложенными классами Twitter Bootstrap:

```
<input
  type="button"
  class="primary btn"
  id="btn"
  value="Show words in last 1000 tweets"/>
```

Чтобы убедиться, что все элементы, к которым мы хотим получить доступ, находятся в DOM, нужно поместить *весь* код для манипуляции с DOM внутри следующей функции jQuery:

```
$(document).ready(function(){
...
}
```

Примечание

Это общая ошибка с динамически генерируемыми HTML-элементами. Они не доступны, пока они не были созданы и введены в DOM.

Следующее одностраничное приложение выводит слова из 200 последних твитов пользователя Twitter отсортированные по частоте использования.

Например, если **@jack** написал:

[46] http://api.jquery.com/jQuery.ajax/

```
"hello world"
"hello everyone, and world"
"hi world"
```

Результат будет:

```
world
hello
and
hi
everyone.
```

Исходный код доступен в папке rpjs/jquery[47]. Это всего лишь однофайловое приложение — **index.html**, основной JavaScript-алгоритм реализован следующим образом:

```
$(document).ready(function(){
```

Мы используем `document.ready`, чтобы отложить выполнение, пока DOM не будет полностью загружен.

```
  $('#btn').click(function() {
```

Это позволяет нам привязать слушатель события click на элемент с id “btn”.

```
    var username=$('#username').val();
    //делаем ajax-вызов, callback
    var url =
     'https://api.twitter.com/1/statuses/user_timeline.json?' +
     'include_entities=true&include_rts=true&screen_name=' +
     username + '&count=1000';
```

Мы создаем экземпляр переменной username и присваеваем ему значение из поля ввода с id-атрибутом username. На следующей строке мы присваиваем целевой URL Twitter REST API переменной `url`. В ответ мы получаем твиты из ленты пользователя.

[47] https://github.com/azat-co/rpjs/tree/master/jquery

```
        if (username!=''){
          list = [ ];  //уникальный глобальный список слов
          counter = { };
          var pages = 0;
          getData(url);
        }
        else {
          alert('Пожалуйста, введите имя пользователя Twitter')
        }
```

Проверка на пустоту имени для избежания отправки неправильного запроса. Если все хорошо, `getData()` сделает запрос. Мы используем именованную функцию, которую нам предстоит определить далее, для того, чтобы обезопасить коллбэки от раздутия (печально известная пирамида гибели[48]).

```
      })
    });
```

Конструкции/блоки закрытия колбэков click и ready.

```
    function getData (url) {
      var request = $.ajax({
        url: url,
        dataType: 'jsonp',
        data: {page:0},
        jsonpCallback: 'fetchData',
        type: 'GET'
      });
    }
```

Функция JSONP волшебным образом (благодаря jQuery) делает кроссдоменные запросы, путем иньекции тега script и добавления имени функции обратного вызова в строку запроса.

Мы будем использовать массив **list** и объект **counter** для алгоритма:

```
    var list = [ ]; //уникальный глобальный список слов
    var counter = { }; //количество повторений слов
```

Функция, которая выполняет сопоставления и подсчет:

[48] http://tritarget.org/blog/2012/11/28/the-pyramid-of-doom-a-javascript-style-trap/

```
function fetchData (m) {
  for (i = 0; i < m.length; i++){
    var words=m[i].text.split(' ');
    for (j = 0; j < words.length; j++){
      words[j] = words[j].replace(/\,/g,'');
      //еще кода ...
      if (words[j].substring(0,4)!="http"&&words[j]!='') {
        if (list.indexOf(words[j])<0) {
          list.push(words[j]);
          counter[words[j]]=1;
        }
        else {
          //добавляем плюс один к счетчику слов
          counter[words[j]]++;
        }
      }
    }
  }
```

Этот код перебирает слова и использует хэш как поисковую таблицу и хранинилище количества.

```
  for (i=0;i<list.length;i++){
    var max=counter[list[i]];
    var p=0;
    for (j=i;j<list.length;j++) {
      if (counter[list[i]]<counter[list[j]]) {
        p=list[i];
        list[i]=list[j];
        list[j]=p;
        maxC=i;
      }
    }
  }
```

Следующий фрагмент сортирует слова (по количеству повторений) и красиво выводит их иньекциями в DOM:

```
    var str='';
    for (i=0;i<list.length;i++){
      str+=counter[list[i]]+': '+list[i]+'\n';
    }
    $('#log').val(str);
    $('#info').html('Analyzed: ' + list.length+
      'word(s) form '+m.length +
      'tweet(s).');
  }
```

Полный код файла **index.html**:

```
$(document).ready(function(){
  $('#btn').click(function() {
    //заменяем картинку лоадера
    var username=$('#username').val();
    //делаем аякс-запрос, коллбэк
    var url =
     'https://api.twitter.com/1/statuses/user_timeline.json?' +
     'include_entities=true&include_rts=true&screen_name=' +
     username + '&count=1000';
    if (username!=''){
      list = [ ];  //уникальный глобальный список слов
      counter = { };
      var pages = 0;
      getData(url);
    }
    else {
      alert('Please enter Twitter username')
    }
  })
});
function getData (url) {
  var request = $.ajax({
    url: url,
    dataType: 'jsonp',
    data: {page:0},
    jsonpCallback: 'fetchData',
    type: 'GET'
  });
}
//ajax callback
var list = [ ]; //уникальный глобальный список слов
var counter = { };
var pages = 0;

```

```
35    function fetchData (m) {
36      for (i = 0; i < m.length; i++){
37        var words=m[i].text.split(' ');
38        for (j = 0; j < words.length; j++){
39          words[j] = words[j].replace(/\,/g,'');
40       ...
41          if (words[j].substring(0,4)!="http"&&words[j]!='') {
42            if (list.indexOf(words[j])<0) {
43              list.push(words[j]);
44              counter[words[j]]=1;
45            }
46            else {
47              //добавляем плюс один к счетчику слов
48              counter[words[j]]++;
49            }
50          }
51        }
52      }
53      //сортировать по количеству повторений
54      for (i=0;i<list.length;i++){
55        var max=counter[list[i]];
56        var p=0;
57        for (j=i;j<list.length;j++) {
58          if (counter[list[i]]<counter[list[j]]) {
59            p=list[i];
60            list[i]=list[j];
61            list[j]=p;
62            maxC=i;
63          }
64        }
65      }
66      //сортированный вывод
67      //красивый вывод с количеством повторений
68      var str='';
69      for (i=0; i<list.length; i++){
70        str+=counter[list[i]]+': '+list[i]+'\n';
71      }
72      $('#log').val(str);
73      $('#info').html('Analyzed: ' + list.length+
74        'word(s) form '+m.length +
75        'tweet(s).');
76    }
```

Попробуйте запустить его и посмотреть, работает ли он с или без локального HTTP-сервера. **Подсказка**: он не должен работать без HTTP-сервера из-за своей зависимости от технологии JSONP.

Примечание

Этот пример был создан с использованием Twitter API v1.0 и может не работать с Twitter API v1.1, который требует проверки аутентификации пользователя для REST API вызовов. Вы можете получить необходимые ключи на dev.twitter.com[49]. Если вы считаете, что здесь должен быть рабочий пример, напишите отзыв на hi@rpjs.co.

3.6 Parse.com

Parse.com[50] — это сервис, который может быть заменой для базы данных и сервера. Он запустился, как средство для поддержки разработки мобильных приложений. Тем не менее, с REST API и JavaScript SDK, Parse.com может быть использован для хранения данных (и многого другого) в любом веб и настольном приложении, что делает его идеальным для быстрого прототипирования.

Перейдите на Parse.com и зарегистрируйте бесплатный аккаунт. Создайте приложение и скопируйте Application ID, REST API Key и JavaScript Key. Нам нужны эти ключи для доступа к нашей коллекцию на Parse.com. Пожалуйста, обратите внимание на вкладку “Data Browser” — там вы можете посмотреть ваши коллекции и записи.

Мы создадим простое приложение, которое позволит сохранять значения в коллекции, используя JavaScript SDK Parse.com. Наше приложение будет состоять из файлов **index.html** и **app.js**. Вот структура нашей проектной папки:

```
/parse
  -index.html
  -app.js
```

Образец можно найти в папке rpjs/parse[51] на GitHub, но вам рекомендуется ввести свой собственный код с нуля. Для начала создайте файл **index.html**:

```
<html lang="en">
<head>
```

Добавьте минифицированную библиотеку jQuery из Google CDN:

[49] https://dev.twitter.com
[50] http://parse.com
[51] https://github.com/azat-co/rpjs/tree/master/parse

```
  <script
    type="text/javascript"
    src=
    "http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js">
  </script>
```

Добавьте библиотеку Parse.com из Parse CDN:

```
  <script
    src="http://www.parsecdn.com/js/parse-1.0.14.min.js">
  </script>
```

Добавьте ваш файл **app.js**:

```
  <script type="text/javascript" src="app.js"></script>
</head>
<body>
<!-- We'll do something here -->
</body>
</html>
```

Создайте файл **app.js** и используйте функцию `$(document).ready`, чтобы убедиться, что DOM готова к манипуляции:

```
$(document).ready(function() {
```

Измените **parseApplicationId** и **parseJavaScriptKey** на значения из приборной панели приложения Parse.com:

```
  var parseApplicationId="";
  var parseJavaScriptKey="";
```

Поскольку мы добавили библиотеку Parse JavaScript SDK, у нас есть доступ к глобальному объекту `Parse`. С ключами мы инициируем соединение и создаем ссылку на коллекцию **Test**:

```
  Parse.initialize(parseApplicationId, parseJavaScriptKey);
  var Test = Parse.Object.extend("Test");
  var test = new Test();
```

Этот простой код будет сохранять объект с индексами **name** и **text** в коллекцию **Test** Parse.com:

```
  test.save({
    name: "John",
    text: "hi"}, {
```

Удобно, что метод save() принимает параметры обратного вызова success и error так же, как функция jQuery.ajax()‘. Чтобы получить подтверждение, мы просто должны смотреть в консоль браузера:

```
    success: function(object) {
      console.log("Parse.com object is saved: "+object);
     //в качестве альтернативы вы можете использовать
     //alert("Parse.com object is saved");
    },
```

Важно знать, почему мы не смогли сохранить объект:

```
    error: function(object) {
      console.log("Error! Parse.com object is not saved: "+object);
    }
  });
})
```

Полный исходный код **index.html**:

```
<html lang="en">
<head>
  <script
    type="text/javascript"
    src=
    "http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js">
  </script>
  <script
    src="http://www.parsecdn.com/js/parse-1.0.14.min.js">
  </script>
  <script type="text/javascript" src="app.js"></script>
</head>
<body>
<!-- We'll do something here -->
</body>
</html>
```

Полный исходный код файла **app.js**:

```
$(document).ready(function() {
  var parseApplicationId="";
  var parseJavaScriptKey="";
  Parse.initialize(parseApplicationId, parseJavaScriptKey);
  var Test = Parse.Object.extend("Test");
  var test = new Test();
  test.save({
    name: "John",
    text: "hi"}, {
    success: function(object) {
      console.log("Parse.com object is saved: "+object);
     //в качестве альтернативы вы можете использовать
     //alert("Parse.com object is saved");
    },
    error: function(object) {
      console.log("Error! Parse.com object is not saved: "+object);
    }
  });
})
```

Предупреждение

С этим подходом мы должны использовать JavaScript SDK Key из приборной панели Parse.com. Для примера с jQuery, мы будем использовать REST API Key с той же странице. Если вы получаете 401 Unauthorized error от Parse.com, это, вероятно, потому, что у вас неправильный ключ API.

Если все было сделано правильно, вы должны видеть **Test** на Parse.com во вкладке Data Browser с заполненными значениями “John” и “hi”. Кроме того, вы должны увидеть соответствующее сообщение в консоли браузера в Developer Tools. Parse.com автоматически создает objectID и временные метки, которые будут очень полезны в нашем приложении Chat.

Parse.com также имеет подробные инструкции для ‘Hello World’-приложения, которые доступны в разделе Quick Start Guide для новых проектов[52] и существующих[53].

3.7 Chat с обзором Parse.com

Chat будет состоять из поля ввода, списка сообщений и кнопки отправить. Нам нужно отобразить список существующих сообщений и быть в состоянии отправлять новые сообщения. Мы будем использовать Parse.com как back-end на данный момент, а позже перейдём на Node.js с MongoDB.

[52] https://parse.com/apps/quickstart#js/blank

[53] https://parse.com/apps/quickstart#js/existing

Вы можете получить бесплатный аккаунт на Parse.com. JavaScript Guide доступен по ссылке https://parse.com/docs/js_guide, JavaScript API тут https://parse.com/docs/js/.

После регистрации на Parse.com перейдите на приборную панель и создайте новое приложение, если вы не сделали этого ранее. Скопируйте Application ID, JavaScript Key и REST API Key вашего свежесозданного приложения. Нам это понадобится позже. Есть несколько способов использования Parse.com[54]:

- REST API: Мы собираемся использовать этот подход на примере с jQuery
- JavaScript SDK: Мы только что использовали этот подход в нашем тестовом примере выше и будем использовать его в примере с Backbone.js позже

REST API является более общим подходом. Parse.com предоставляет конечные точки, которые мы можем запросить методом `$.ajax()` библиотеки jQuery. Вот описание доступных URL и методов: parse.com/docs/rest[55].

3.8 Chat с Parse.com: REST API и jQuery-версия

Полный код доступен в папке rpjs/rest[56], но мы предлагаем вам сначало попробовать написать собственное приложение.

Мы будем использовать REST API Parse.com и jQuery. Parse.com поддерживает кросс-доменные AJAX-вызовы, поэтому нам не нужен JSONP.

Примечание

Если вы решите развернуть ваше back-end приложение на другом домене, которое будет выступать в качестве заменителя Parse.com, вам придется использовать либо JSONP на front-end, либо пользовательские заголовкои CORS на back-end. Эта тема будет описана в книге позже.

Сейчас структура приложения должна выглядеть следующим образом:

```
index.html
  css/bootstrap.min.css
  css/style.css
  js/app.js
```

Давайте создадим визуальное представление для приложения Chat. Все, что мы хотим, это просто отобразить список сообщений с именами пользователей в хронологическом порядке. Таким образом, таблица подойдет идеально. Мы можем динамически создавать элементы `<tr>` и постояно вставлять их, как только получаем новые сообщения.

Создайте простой HTML-файл **index.html** со следующим содержанием:

[54] http://parse.com

[55] https://parse.com/docs/rest

[56] https://github.com/azat-co/rpjs/tree/master/rest

- Включение JS- и CSS-файлов
- Адаптивная структура с Twitter Boostrap
- Таблица сообщений
- Форма для новых сообщений

Давайте начнем с **head** и зависимостей. Мы добавим CDN jQuery, локальный **app.js**, локальный сжатый Twitter Boostrap и пользовательские стили **style.css**:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <script
      src=
   "https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"
      type ="text/javascript"
      language ="javascript" ></script>
    <script src="js/app.js" type="text/javascript"
      language ="javascript" ></script>
    <link href="css/bootstrap.min.css" type="text/css"
      rel="stylesheet" />
    <link href="css/bootstrap-responsive.min.css" type="text/css"
      rel="stylesheet" />
    <link href="css/style.css" type="text/css" rel="stylesheet" />
  </head>
```

Элемент body будет иметь типичные Twitter Boostrap скаффолдинг-элементы определенные классами **container-fluid** и **row-fluid**:

```
  <body>
    <div class="container-fluid">
      <div class="row-fluid">
        <h1>Chat with Parse REST API</h1>
```

Таблица сообщений пуста, потому что мы будем заполнять ее программно внутри JS-кода:

```
        <table class="table table-bordered table-striped">
          <caption>Messages</caption>
          <thead>
            <tr>
              <th>
                Username
              </th>
              <th>
                Message
              </th>
            </tr>
          </thead>
          <tbody>
            <tr>
              <td colspan="2">No messages</td>
            </tr>
          </tbody>
        </table>
      </div>
```

Еще один ряд и вот наша новая форма для сообщений, в котором кнопка *SEND* использует классы Twitter Bootstrap **btn** и **btn-primary**:

```
      <div class="row-fluid">
        <form id="new-user">
          <input type="text" name="username"
            placeholder="Username" />
          <input type="text" name="message"
            placeholder="Message" />
          <a id="send" class="btn btn-primary">SEND</a>
        </form>
      </div>
    </div>
  </body>
</html>
```

Полный исходный код **index.html**:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <script
      src=
   "https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"
      type ="text/javascript"
      language ="javascript" ></script>
    <script src="js/app.js" type="text/javascript"
      language ="javascript" ></script>
    <link href="css/bootstrap.min.css" type="text/css"
      rel="stylesheet" />
    <link href="css/bootstrap-responsive.min.css" type="text/css"
      rel="stylesheet" />
    <link href="css/style.css" type="text/css" rel="stylesheet" />
  </head>
  <body>
    <div class="container-fluid">
      <div class="row-fluid">
        <h1>Chat with Parse REST API</h1>
        <table class="table table-bordered table-striped">
          <caption>Messages</caption>
          <thead>
            <tr>
              <th>
                Username
              </th>
              <th>
                Message
              </th>
            </tr>
          </thead>
          <tbody>
            <tr>
              <td colspan="2">No messages</td>
            </tr>
          </tbody>
        </table>
      </div>
      <div class="row-fluid">
        <form id="new-user">
          <input type="text" name="username"
            placeholder="Username" />
          <input type="text" name="message"
            placeholder="Message" />
```

```
46            <a id="send" class="btn btn-primary">SEND</a>
47          </form>
48        </div>
49      </div>
50    </body>
51  </html>
```

Таблица будет содержать наши сообщения. Форма будет обеспечивать ввод новых сообщений.

Теперь мы собираемся написать три основных функции:

1. `getMessages()`: функция для получения сообщений
2. `updateView()`: функция для рендеринга списка сообщений
3. `$('#send').click(...)`: функция для отправки нового сообщения

Для простоты мы поместим всю логику в один файл **app.js**. Конечно же, это хорошая идея, отделить базовый код функциональности, особенно когда ваш проект становится все больше.

Замените эти значения вашими собственным. Будьте осторожны, используйте REST API Key, а не JavaScript SDK Key из предыдущего примера:

```
1   var parseID='YOUR_APP_ID';
2   var parseRestKey='YOUR_REST_API_KEY';
```

Оберните все в `document.ready`, получите сообщения и определите событие click на *SEND*:

```
1  $(document).ready(function(){
2    getMessages();
3    $("#send").click(function(){
4      var username = $('input[name=username]').attr('value');
5      var message = $('input[name=message]').attr('value');
6      console.log(username)
7      console.log('!'')
```

Когда мы отправляем новое сообщение, мы делаем HTTP-вызов функцией `jQuery.ajax`:

```
    $.ajax({
      url: 'https://api.parse.com/1/classes/MessageBoard',
      headers: {
        'X-Parse-Application-Id': parseID,
        'X-Parse-REST-API-Key': parseRestKey
      },
      contentType: 'application/json',
      dataType: 'json',
      processData: false,
      data: JSON.stringify({
        'username': username,
        'message': message
      }),
      type: 'POST',
      success: function() {
        console.log('sent');
        getMessages();
      },
      error: function() {
        console.log('error');
      }
    });

  });
})
```

Метод для получения сообщений из нашего дистанционного REST API сервера также использует функцию `jQuery.ajax`:

```
function getMessages() {
  $.ajax({
    url: 'https://api.parse.com/1/classes/MessageBoard',
    headers: {
      'X-Parse-Application-Id': parseID,
      'X-Parse-REST-API-Key': parseRestKey
    },
    contentType: 'application/json',
    dataType: 'json',
    type: 'GET',
```

Если запрос успешно завершен (статус 200 или аналогичный), мы вызываем функцию `updateView`:

```
    success: function(data) {
      console.log('get');
      updateView(data);
    },
    error: function() {
      console.log('error');
    }
  });
}
```

Эта функция рендерит список сообщений, который мы получаем от сервера:

```
function updateView(messages) {
```

Мы используем селектор jQuery '.table tbody', чтобы создать объект, ссылающийся на этот элемент. Затем мы чистим весь innerHTML этого элемента:

```
  var table=$('.table tbody');
  table.html('');
```

Мы используем функцию `jQuery.each` для перебора каждого сообщения:

```
  $.each(messages.results, function (index, value) {
    var trEl =
```

Следующий код создает HTML-элементы (и объект jQuery этих элементов) программно:

```
      $('<tr><td>'
        + value.username
        + '</td><td>'
        + value.message +
        '</td></tr>');
```

И добавляет (инжектирует после) элемент таблицы `tbody`:

```
    table.append(trEl);
  });
  console.log(messages);
}
```

Весь **app.js**:

```
var parseID='YOUR_APP_ID';
var parseRestKey='YOUR_REST_API_KEY';

$(document).ready(function(){
  getMessages();
  $("#send").click(function(){
    var username = $('input[name=username]').attr('value');
    var message = $('input[name=message]').attr('value');
    console.log(username)
    console.log('!'')
    $.ajax({
      url: 'https://api.parse.com/1/classes/MessageBoard',
      headers: {
        'X-Parse-Application-Id': parseID,
        'X-Parse-REST-API-Key': parseRestKey
      },
      contentType: 'application/json',
      dataType: 'json',
      processData: false,
      data: JSON.stringify({
        'username': username,
        'message': message
      }),
      type: 'POST',
      success: function() {
        console.log('sent');
        getMessages();
      },
      error: function() {
        console.log('error');
      }
    });

  });
})
function getMessages() {
  $.ajax({
    url: 'https://api.parse.com/1/classes/MessageBoard',
    headers: {
      'X-Parse-Application-Id': parseID,
      'X-Parse-REST-API-Key': parseRestKey
    },
    contentType: 'application/json',
    dataType: 'json',
    type: 'GET',
```

```
46      success: function(data) {
47        console.log('get');
48        updateView(data);
49      },
50      error: function() {
51        console.log('error');
52      }
53    });
54  }
55
56  function updateView(messages) {
57    var table=$('.table tbody');
58    table.html('');
59    $.each(messages.results, function (index, value) {
60      var trEl =
61        $('<tr><td>'
62          + value.username
63          + '</td><td>'
64          + value.message +
65          '</td></tr>');
66      table.append(trEl);
67    });
68    console.log(messages);
69  }
```

Оно будет вызывать функцию `getMessages()` и делать GET-запрос с помощью функции `$.ajax` библиотеки jQuery. Полный список параметров для функции `ajax` доступен здесь api.jquery.com/jQuery.ajax[57]. Наиболее важными из них являются параметры URL, headers и type.

Затем, после успешного ответа, она будет вызывать функцию `updateView()`, которая очищает таблицу **tbody** и проходится по результатам из ответа с помощью функции jQuery `$.each` (api.jquery.com/jQuery.each[58]). Нажатие на кнопку send будет отправлять POST-запрос к Parse.com REST API, а затем, после успешного ответа, получать сообщения, вызывая функцию `getMessages()`.

Здесь один из способов динамического создания HTML-элемента *div*, используя jQuery:

```
1  $("<div>");
```

3.9 Пуш на GitHub

Чтобы создать GitHub-репозиторий, перейдите на github.com[59], войдите и создайте новый репозиторий. Там будет SSH-адрес — скопируйте его. В окне терминала перейдите в папку

[57] http://api.jquery.com/jQuery.ajax/
[58] http://api.jquery.com/jQuery.each/
[59] http://github.com

проекта, который вы хотели бы запушить на GitHub.

1. Создайте локальный Git и папку *.git* в корне папки проекта:

```
$ git init
```

2. Добавьте все файлы в репозиторий и начните их отслеживание:

```
$ git add .
```

3. Сделайте первый коммит:

```
$ git commit -am "initial commit"
```

4. Добавьте удаленный адресат GitHub:

```
$ git remote add your-github-repo-ssh-url
```

   Это может выглядеть примерно так:

```
$ git remote add origin git@github.com:azat-co/simple-message-board.git
```

5. Теперь у нас все готово, чтобы запушить ваш локальный Git-репозиторий на удаленный адрес назначения на GitHub с помощью следующей команды:

```
$ git push origin master
```

6. Вы должны увидеть ваши файлы на github.com[60] под вашим аккаунтом и репозиторием.

Позже, когда вы вносите изменения в файл, нет необходимости повторять все шаги, описанные выше. Просто запустите:

```
$ git add .
$ git commit -am "some message"
$ git push origin master
```

Если нет новых неотслеживаемых файлов, которые вы хотите начать отслеживать:

```
$ git commit -am "some message"
$ git push origin master
```

Для включения изменений из отдельных файлов, запустите:

[60] http://github.com

```
$ git commit filename -m "some message"
$ git push origin master
```

Для удаления файла из Git-репозитория:

```
$ git rm filename
```

Для справки по командам Git:

```
$ git --help
```

Развертывание приложений с Windows Azure или Heroku так же просто, как запушить код/файлы на GitHub. Последние три шага (#4-6) будут отличаться другим удаленным адресом (URL) и другим псевдонимом.

3.10 Развертка на Windows Azure

Вы сможете развернуть приложение на Windows Azure с Git.

1. Перейдите на портал Windows Azure https://windows.azure.com/, войдите с вашим Live ID и создайте веб-сайт, если вы не сделали этого ранее. Включите “Set up Git publishing” (Установка Git-публикации), предоставив имя пользователя и пароль (они должны отличаться от ваших полномочий Live ID). Скопируйте ваш URL куда-нибудь.
2. Создать локальный Git-репозиторий в папке проекта, который вы хотели бы опубликовать или развернуть:

```
$ git init
```

3. Добавьте все файлы в репозиторий и начните их отслеживание:

```
$ git add .
```

4. Сделайте первый коммит:

```
$ git commit -am "initial commit"
```

5. Добавьте Windows Azure в качестве адреса удаленного Git-репозитория:

```
$ git remote add azure your-url-for-remote-repository
```

В моем случае, эта команда выглядила следующим образом:

```
$ git remote add
```

```
> azure https://azatazure@azat.scm.azurewebsites.net/azat.git
```

6. Запушьте ваш локальный Git-репозиторий на удаленный репозиторий Windows Azure, который и развернет файлы и приложения:

```
$ git push azure master
```

Как и с GitHub, нет необходимости повторять первые шаги, когда вы позже обновили файлы, так как мы уже должны иметь локальный Git-репозиторий в виде папки *.git* в корне папки проекта.

3.11 Развертка на Heroku

Единственным серьезным отличием является то, что Heroku использует Cedar Stack, который не поддерживает статические проекты, они же простые HTML-приложения, такое как наше тестовое Parse.com приложение или Parse.com версия приложения Chat. Мы можем использовать "фейковый" PHP-проект, чтобы обойти это ограничение. Создайте файл **index.php** на том же уровне, что и **index.html** в папке проекта, который вы хотели бы опубликовать/развернуть на Heroku, со следующим содержанием:

```
<?php echo file_get_contents('index.html'); ?>
```

Для вашего удобства, файл **index.php** уже включен в *rpjs/rest*.

Существует еще более простой способ опубликовать статические файлы на Heroku с Cedar stack, который описан в посте Static Sites on Heroku Cedar[61]. Для того, чтобы Cedar Stack работал с вашими статическими файлами, все что вам нужно сделать, это ввести и выполнить следующие команды в папке проекта:

```
$ touch index.php
$ echo 'php_flag engine off' > .htaccess
```

Кроме того, можно использовать Ruby Bamboo stack. В этом случае, мы должны были бы иметь следующую структуру:

```
-project folder
  -config.ru
  /public
    -index.html
    -/css
    app.js
    ...
```

Путь к CSS и к другим активам в *index.html* должен быть относительным, то есть, 'css/style.css'. Файл *config.ru* должен содержать следующий код:

[61] http://kennethreitz.com/static-sites-on-heroku-cedar.html

```
use Rack::Static,
  :urls => ["/stylesheets", "/images"],
  :root => "public"

run lambda { |env|
  [
    200,
    {
      'Content-Type'  => 'text/html',
      'Cache-Control' => 'public, max-age=86400'
    },
    File.open('public/index.html', File::RDONLY)
  ]
}
```

Для более подробной информации, вы можете обратиться к devcenter.heroku.com/articles/static-sites-on-heroku[62].

Как только у вас появяться все файлы поддержки для Cedar или Bamboo, выполните следующие действия:

1. Создайте локальный Git-репозиторий и папку *.git*, если вы этого еще не сделали:

```
$ git init
```

2. Добавьте файлы:

```
$ git add .
```

3. Закоммитьте файлы и изменения:

```
$ git commit -m "my first commit"
```

4. Создайте Heroku Cedar stack приложение и добавьте удаленный адресат:

```
$ heroku create
```

   Если все прошло хорошо, вам должны сказать, что удаленный репозиторий был добавлен и приложение было создано и дать вам название приложения.

5. Чтобы просмотреть тип удаленного репозитория, выполните (*необязательно*):

```
$ git remote show
```

6. Разверните код на Heroku:

[62] https://devcenter.heroku.com/articles/static-sites-on-heroku

```
$ git push heroku master
```

Терминальные логи должны вам сказать прошла ли развертка гладко.

7. Чтобы открыть приложение в вашем браузере, напечатайте:

```
$ heroku open
```

или просто перейдите по URL вашего приложения, что то типа этого “http://yourappname-NNNN.herokuapp.com”.

8. Чтобы просмотреть логи Heroku для этого приложения, введите:

```
$ heroku logs
```

Чтобы обновить приложение с новым кодом, повторите **только** следующие шаги:

```
$ git add -A
$ git commit -m "commit for deploy to heroku"
$ git push -f heroku
```

Примечание

Вам будет назначен новый URL приложение каждый раз, когда вы создаете новое приложение Heroku с помощью команды: `$ heroku create`.

3.12 Обновление и удаление сообщений

В соответствии с REST API, обновление объекта выполняется с помощью метода PUT, а удаление — DELETE. Оба они могут быть легко реализованны той же функцией `jQuery.ajax`, что мы использовали для GET и POST, передавая ID объекта, над котором мы хотим выполнить операцию.

4. Введение в Backbone.js

Резюме: демонстрация того, как создать Backbone.js-приложение с нуля, как использовать представления, коллекции, дочерние представления, модели, привязку событий, AMD, Require.js на примере приложения яблочной базы данных.

_ “Код — это не имущество. Это ответственность. Чем больше вы пишете, тем больше надо будет поддерживать его позже.”_ — Неизвестный

4.1 Настройка Backbone.js-приложения с нуля

Мы собираемся создать типичное приложение “Hello World”, использующее Backbone.js и архитектуру Mode-View-Controller (MVC). Я понимаю, что в начале это может показаться излишним, но по мере продвижения мы будем добавлять больше и больше комплексности, в том числе моделей, подпредставлений и коллекций.

Полный исходный код приложения “Hello World” доступен на GitHub github.com/azat-co/rpjs/backbone/hello-world[1].

4.1.1 Зависимости

Скачайте следующие библиотеки:

- development-версия jQuery 1.9[2]
- development-версия Underscore.js[3]
- development-версия Backbone.js[4]

Добавьте эти фреймворки в файл **index.html**, как здесь:

[1] https://github.com/azat-co/rpjs/tree/master/backbone/hello-world
[2] http://code.jquery.com/jquery-1.9.0.js
[3] http://underscorejs.org/underscore.js
[4] http://backbonejs.org/backbone.js

```
<!DOCTYPE>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="underscore.js"></script>
  <script src="backbone.js"></script>

  <script>
    //TODO написать немного крутого JS-кода
  </script>

</head>
<body>
</body>
</html>
```

Примечание

Мы также можем поставить теги `<script>` сразу после тега `</body>` в конце файла. Это изменит порядок загрузки скриптов и остального HTML и скажется на производительности в больших файлах.

Давайте определим простой Router Backbone.js внутри тега `<script>`:

```
  ...
  var router = Backbone.Router.extend({
  });
  ...
```

Примечание

Пока, чтоб до предела все упростить (Keep It Simple Stupid (KISS)), мы поместим весь наш код JavaScript прямо в файл **index.html**. Это не очень хорошая идея для реальной разработки или боевого коде. Мы срефакторим его позже.

Затем установим специальное свойство `routes` внутри метода **extend**:

```
  var router = Backbone.Router.extend({
    routes: {
    }
  });
```

Свойство **routes** в Backbone.js должно быть в следующем формате: `'path/:param':'action'`, который будет через URL `filename#path/param`, запускать функцию по имени **action** (определенной в объекте Router). Пока добавим одиночный маршрут **home**:

```
var router = Backbone.Router.extend({
  routes: {
    '': 'home'
  }
});
```

Хорошо, теперь нам нужно добавить функцию **home**:

```
var router = Backbone.Router.extend({
  routes: {
    '': 'home'
  },
  home: function(){
    //TODO отобразить html
  }
});
```

Мы вернемся к функции **home** позже, чтобы добавить больше логики для создания и рендеринга View. Сейчас мы должны определить наш **homeView**:

```
var homeView = Backbone.View.extend({
});
```

Выглядит знакомым, не так ли? Backbone.js использует похожий синтаксис для всех его компонентов: функция **extend** и объект JSON в качестве параметра.

Есть множество способов сделать то, что мы собираемся, но лучше всего использовать свойства **el** и ***template***, которые являются магическими, т.е. специальными в Backbone.js:

```
var homeView = Backbone.View.extend({
  el: 'body',
  template: _.template('Hello World')
});
```

Свойство **el** — это просто строка, которая содержит селектор jQuery (вы можете использовать имя класса с '.' и id с '#'). Свойству template была назначена функция Underscore.js **template** с простым текстом 'Hello World'.

Для рендеринга нашего **homeView** мы используем `this.$el`, который является объектом jQuery ссылающимся на элемент из свойства **el**, и функцию jQuery `.html()` для замены HTML значением `this.template()`. Вот полный код для нашего Backbone.js View:

```
var homeView = Backbone.View.extend({
  el: 'body',
  template: _.template('Hello World'),
  render: function(){
    this.$el.html(this.template({}));
  }
});
```

Теперь, если мы вернемся к **router**, мы можем добавить эти две строки в функцию **home**:

```
var router = Backbone.Router.extend({
  routes: {
    '': 'home'
  },
  initialize: function(){

  },
  home: function(){
    this.homeView = new homeView;
    this.homeView.render();
  }
});
```

Первая строка создаст объект *homeView* и установит его в свойство маршрутизатора *homeView*. Вторая строка будет вызывать метод `render()` в объекте *homeView*, приводя к выводу 'Hello World'.

Наконец, чтобы запустить Backbone-приложение, мы вызовем `new Router` внутри обертки document-ready, чтобы убедиться, что DOM полностью загружен:

```
var app;
$(document).ready(function(){
  app = new router;
  Backbone.history.start();
})
```

Здесь полный код файла **index.html**:

```
<!DOCTYPE>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="underscore.js"></script>
  <script src="backbone.js"></script>

  <script>
    var app;
    var router = Backbone.Router.extend({
      routes: {
        '': 'home'
      },
      initialize: function(){
       //некоторый код для исполнения
       //когда создан экземпляр объекта
      },
      home: function(){
        this.homeView = new homeView;
        this.homeView.render();
      }
    });

    var homeView = Backbone.View.extend({
      el: 'body',
      template: _.template('Hello World'),
      render: function(){
        this.$el.html(this.template({}));
      }
    });

    $(document).ready(function(){
      app = new router;
      Backbone.history.start();
    })

  </script>
</head>
<body>
  <div></div>
</body>
</html>
```

Откройте **index.html** в браузере, чтобы убедиться, что оно работает: на странице должно быть сообщение ‘Hello World’.

4.2 Работа с коллекциями

Полный исходный код этого примера находится в rpjs/backbone/collections[5]. Он создан на базе примера “Hello World” из главы **Настройка Backbone.js-приложения с нуля**, которое доступно для скачивания по ссылке rpjs/backbone/hello-world[6].

Мы должны добавить некоторые данные, чтобы поиграться с ними и размяться. Чтобы сделать это, добавьте следующий код прямо после тега **script** и перед другим кодом:

```
var appleData = [
  {
    name: "fuji",
    url: "img/fuji.jpg"
  },
  {
    name: "gala",
    url: "img/gala.jpg"
  }
];
```

Это наша яблочная *база данных*. :-) А если быть точным, наш заменитель конечной точки REST API, которая обеспечивает нас именами и URL изображений яблок (модели данных).

Примечание

Этот макет набора данных можно легко заменить путем присвоения конечных точек REST API back-end свойству `url` в Backbone.js коллекциях и/или моделей и вызывания метода `fetch()` для них.

Теперь, чтобы сделать юзабилити (User Experience, UX) немного лучше, мы можем добавить новый маршрут в объект **routes** в Backbone Route:

```
...
routes: {
  '': 'home',
  'apples/:appleName': 'loadApple'
},
...
```

Это позволит пользователям перейти в `index.html#apples/SOMENAME` и ожидать увидеть некоторую информацию о яблоке. Эта информация будет найдена и отрендерина функцией **loadApple** в определении Backbone Router:

[5] https://github.com/azat-co/rpjs/tree/master/backbone/collections
[6] https://github.com/azat-co/rpjs/tree/master/backbone/hello-world

```
  loadApple: function(appleName){
    this.appleView.render(appleName);
  }
```

Вы заметили переменную **appleName**? У нее точно такое же имя, что и то, который мы использовали в **маршруте**. Вот как мы можем получить доступ к параметрам строки запроса (например, `?param=value&q=search`) в Backbone.js.

Теперь нам нужно отрефакторить немного больше кода, чтобы создать Backbone Collection, заполнить его данными из нашей переменной **appleData** и передать коллекцию в **homeView** и **appleView**. Довольно удобно, что мы делаем все это в методе конструктора Router **initialize**:

```
  initialize: function(){
    var apples = new Apples();
    apples.reset(appleData);
    this.homeView = new homeView({collection: apples});
    this.appleView = new appleView({collection: apples});
  },
```

На данный момент, мы очень много сделали с классом Router и он должен выглядеть следующим образом:

```
  var router = Backbone.Router.extend({
    routes: {
      '': 'home',
      'apples/:appleName': 'loadApple'
    },
    initialize: function(){
      var apples = new Apples();
      apples.reset(appleData);
      this.homeView = new homeView({collection: apples});
      this.appleView = new appleView({collection: apples});
    },
    home: function(){
      this.homeView.render();
    },
    loadApple: function(appleName){
      this.appleView.render(appleName);
    }
  });
```

Давайте изменим наш **homeView** немного, чтобы увидеть всю *базу данных*:

```
var homeView = Backbone.View.extend({
  el: 'body',
  template: _.template('Apple data: <%= data %>'),
  render: function(){
    this.$el.html(this.template({
    data: JSON.stringify(this.collection.models)
  }));
  }
});
```

В настоящее время, мы просто вывели строку представления объекта JSON в браузере. Это вообще не удобно, но позже мы улучшим все с помощью списка и подпредставлений.

Наша яблочная Backbone-коллекция очень простая и понятная:

```
var Apples = Backbone.Collection.extend({
});
```

Примечание

Backbone автоматически создает модели внутри коллекции, когда мы используем функции `fetch()` или `reset()`.

Представление Apple не сложнее — оно имеет только два свойства: **template** и **render**. В шаблоне мы хотим отобразить теги **figure**, **img** и **figcaption** с определенными значениями. Движок для шаблонов от Underscore.js удобен для этой задачи:

```
var appleView = Backbone.View.extend({
  template: _.template(
        '<figure>\
          <img src="<%= attributes.url %>"/>\
          <figcaption><%= attributes.name %></figcaption>\
         </figure>'),
...
});
```

Чтобы сделать строку JavaScript, содержащую HTML-теги, более читаемой, мы можем использовать символ экранирования перевода строки обратный слеш (\) или закрывать строки и соединять их знаком плюс (+). Это пример **appleView**, представленный выше, который отрефакторин с использованием последнего подхода:

```
  var appleView = Backbone.View.extend({
    template: _.template(
        '<figure>'+
          +'<img src="<%= attributes.url %>"/>'+
          +'<figcaption><%= attributes.name %></figcaption>'+
        +'</figure>'),
  ...
```

Обратите внимание на символы '<%=' и '%>' — это инструкции для Undescore.js выводить значения в свойствах **url** и **name** из объекта **attributes**.

Наконец, мы добавляем функцию рендеринга в класс **appleView**.

```
  render: function(appleName){
    var appleModel = this.collection.where({name:appleName})[0];
    var appleHtml = this.template(appleModel);
    $('body').html(appleHtml);
  }
```

Мы находим модель внутри коллекции методом `where()` и используем `[]`, чтобы выбрать первый элемент. Прямо сейчас, функция **render** отвечает и за загрузку данных, и за рендеринг. Позже мы реорганизуем функцию, чтобы отделить эти две функциональности в разные методы.

Все приложение, которое находится в папке rpjs/backbone/collections/index.html[7], выглядит следующим образом:

```
<!DOCTYPE>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="underscore.js"></script>
  <script src="backbone.js"></script>

  <script>
   var appleData = [
      {
        name: "fuji",
        url: "img/fuji.jpg"
      },
      {
        name: "gala",
        url: "img/gala.jpg"
      }
    ];
    var app;
```

[7] https://github.com/azat-co/rpjs/tree/master/backbone/collections

```
20      var router = Backbone.Router.extend({
21        routes: {
22          "": "home",
23          "apples/:appleName": "loadApple"
24        },
25        initialize: function(){
26          var apples = new Apples();
27          apples.reset(appleData);
28          this.homeView = new homeView({collection: apples});
29          this.appleView = new appleView({collection: apples});
30        },
31        home: function(){
32          this.homeView.render();
33        },
34        loadApple: function(appleName){
35          this.appleView.render(appleName);
36        }
37      });
38
39      var homeView = Backbone.View.extend({
40        el: 'body',
41        template: _.template('Apple data: <%= data %>'),
42        render: function(){
43          this.$el.html(this.template({
44          data: JSON.stringify(this.collection.models)
45        }));
46        }
47        //TODO подпредставления
48      });
49
50      var Apples = Backbone.Collection.extend({
51
52      });
53      var appleView = Backbone.View.extend({
54        template: _.template('<figure>\
55                    <img src="<%= attributes.url %>"/>\
56                    <figcaption><%= attributes.name %></figcaption>\
57                  </figure>'),
58        //TODO переписать с загрузкой яблок и привязкой событий
59        render: function(appleName){
60          var appleModel = this.collection.where({
61            name:appleName
62          })[0];
63          var appleHtml = this.template(appleModel);
64          $('body').html(appleHtml);
```

```
65        }
66      });
67      $(document).ready(function(){
68        app = new router;
69        Backbone.history.start();
70      })
71
72    </script>
73  </head>
74  <body>
75    <div></div>
76  </body>
77  </html>
```

Откройте файл `collections/index.html` в вашем браузере. Вы должны увидеть данные из нашей “базу данных”, то есть, яблочные данные: [{“name”:”fuji”,”url”:”img/fuji.jpg”},{“name”:”gala”,”url”:”img/gala

Теперь, давайте перейдем в `collections/index.html#apples/fuji` или `collections/index.html#apples/g` в вашем браузере. Мы ожидаем увидеть изображение с подписью. Это подробный вид элемента, который в данном случае является яблоком. Хорошая работа!

4.3 Привязка событий

В реальной жизни, получения данных не происходит мгновенно, так что давайте перепишем наш код для имитации этого. Для лучшего UI/UX, мы должны показать значок загрузки (также известный, как крутилка или ajax-loader) для пользователей, чтобы уведомить их, что информация загружается.

Хорошо, что у нас есть привязка событий в Backbone. Без него нам надо будет передать функцию, которая рендерит HTML, в качестве коллбэка в функцию загрузки данных, чтобы убедиться, что функция рендеринга не выполнится прежде, чем мы получим актуальные данные для показа.

Поэтому, когда пользователь переходит к детальному просмотру (`apples/:id`), мы только вызываем функцию, которая загружает данные. Затем, с соответствующими слушателями событий, наш view автоматически (это не опечатка) обновит сам себя, когда есть новые данные или при изменении данных (Backbone.js поддерживает несколько событий и даже кастомные события).

Давайте изменим код в маршрутизаторе:

```
1   ...
2     loadApple: function(appleName){
3       this.appleView.loadApple(appleName);
4     }
5   ...
```

Все остальное остается прежним, пока мы не доберемся до класса **appleView**. Нам нужно добавить конструктор или метод **initialize**, название которого является специальным

словом/свойством в Backbone.js. Он вызывается каждый раз, когда мы создаем экземпляр объекта, т.е., `var someObj = new SomeObject()`. Мы также можем передать дополнительные параметры в функцию **initialize**, как мы это делали с нашими вьюхами (мы передали обьект с ключем **collection** и значением Backbone-коллекции **apples**). Подробнее о конструкторах Backbone.js backbonejs.org/#View-constructor[8].

```
...
var appleView = Backbone.View.extend({
  initialize: function(){
    //TODO: создать модель (aka яблоко)
  },
...
```

Отлично, у нас есть функция **initialize**. Теперь нам нужно создать модель, которая будет представлять одно яблоко и создавать соответсвующие слушатели событий для модели. Мы будем использовать два типа событий: `change` и кастомное событие под названием `spinner`. Чтобы сделать это, мы будем использовать функцию `on()`, которая принимает эти свойства: `on(event, actions, context)` — подробнее об этом backbonejs.org/#Events-on[9]:

```
...
var appleView = Backbone.View.extend({ initialize: function() {
    this.model = new (Backbone.Model.extend({}));
    this.model.bind('change', this.render, this);
    this.bind('spinner',this.showSpinner, this);
  },
...
```

Код выше в основном сводится к двум простым вещам:

1. Вызов функции `render()` из объекта **appleView**, когда модель изменена
2. Вызов метода `showSpinner()` из объекта **appleView**, когда произошло событие **spinner**.

Пока все идет хорошо, не так ли? Но как насчет крутилки иконкой GIF? Давайте создадим новое свойство в **appleView**:

```
...
  templateSpinner: '<img src="img/spinner.gif" width="30"/>',
...
```

Помните вызов `loadApple` в маршрутизаторе? Это то, как мы можем реализовать функцию в **appleView**:

[8] http://backbonejs.org/#View-constructor
[9] http://backbonejs.org/#Events-on

```
  ...
  loadApple:function(appleName){
    this.trigger('spinner');
    //показать GIF-картинку крутилки
    var view = this;
        //мы должны иметь доступ к this внутри замыкания
    setTimeout(function(){
        //симулирует лаги при
    //извлечение данных из удаленного сервера
      view.model.set(view.collection.where({
        name:appleName
      })[0].attributes);
    },1000);
  },
  ...
```

Первая строка запустит событие `spinner` (функция для которого, которую мы еще должны написать).

Вторая строка просто для передачи контекста (мы можем использовать **appleView** внутри замыкания).

Функция `setTimeout` имитирует лаги ответа реального удаленного сервера. Внутри нее, мы назначаем атрибуты выбранной модели в модель нашей view, используя функцию `model.set()` и свойство **model.attributes** (которое возвращает свойства модели).

Теперь мы можем удалить дополнительный код из метода `render` и реализовать функцию `showSpinner`:

```
  render: function(appleName){
    var appleHtml = this.template(this.model);
    $('body').html(appleHtml);
  },
  showSpinner: function(){
    $('body').html(this.templateSpinner);
  }
  ...
```

Вот и все! Откройте `index.html#apples/gala` или `index.html#apples/fuji` в вашем браузере и наслаждайтесь анимацией загрузки пока ждете картинки яблока.

Полный код **index.html** файла:

```
<!DOCTYPE>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="underscore.js"></script>
  <script src="backbone.js"></script>

  <script>
   var appleData = [
      {
        name: "fuji",
        url: "img/fuji.jpg"
      },
      {
        name: "gala",
        url: "img/gala.jpg"
      }
    ];
    var app;
    var router = Backbone.Router.extend({
      routes: {
        "": "home",
        "apples/:appleName": "loadApple"
      },
      initialize: function(){
        var apples = new Apples();
        apples.reset(appleData);
        this.homeView = new homeView({collection: apples});
        this.appleView = new appleView({collection: apples});
      },
      home: function(){
        this.homeView.render();
      },
      loadApple: function(appleName){
        this.appleView.loadApple(appleName);

      }
    });

    var homeView = Backbone.View.extend({
      el: 'body',
      template: _.template('Apple data: <%= data %>'),
      render: function(){
        this.$el.html(this.template({
          data: JSON.stringify(this.collection.models)
```

```
46          }));
47        }
48        //TODO подпредставления
49      });
50
51      var Apples = Backbone.Collection.extend({
52
53      });
54      var appleView = Backbone.View.extend({
55        initialize: function(){
56          this.model = new (Backbone.Model.extend({}));
57          this.model.on('change', this.render, this);
58          this.on('spinner',this.showSpinner, this);
59        },
60        template: _.template('<figure>\
61                  <img src="<%= attributes.url %>"/>\
62                  <figcaption><%= attributes.name %></figcaption>\
63                  </figure>'),
64        templateSpinner: '<img src="img/spinner.gif" width="30"/>',
65
66        loadApple:function(appleName){
67          this.trigger('spinner');
68          var view = this;
69              //мы должны иметь доступ к this внутри замыкания
70          setTimeout(function(){         //симулирует лаги при
71                    //извлечение данных из удаленного сервера
72            view.model.set(view.collection.where({
73              name:appleName
74            })[0].attributes);
75          },1000);
76
77        },
78
79        render: function(appleName){
80          var appleHtml = this.template(this.model);
81          $('body').html(appleHtml);
82        },
83        showSpinner: function(){
84          $('body').html(this.templateSpinner);
85        }
86
87      });
88      $(document).ready(function(){
89        app = new router;
90        Backbone.history.start();
```

```
91      })
92
93    </script>
94  </head>
95  <body>
96    <a href="#apples/fuji">fuji</a>
97    <div></div>
98  </body>
99  </html>
```

4.4 Представления и подпредставления с Underscore.js

Этот пример доступен по ссылке rpjs/backbone/subview[10].

Подпредставления (Subview) — это Backbone-представление, которое создается и используется внутри другого Backbone-представления. Концепция подпредставлений отличный абстрагировать (отделить) UI-события (например, клики) и шаблоны для аналогично структурированных элементов (например, яблок).

Использование подпредставлений может понадобиться при включении строки в таблицу, элемента списка в список, абзаца, новой строки, и т.д.

Мы переделаем нашу домашнюю страницу, чтобы показать нормальный список яблок. Каждый элемент списка будет иметь имя яблока и ссылку “buy” с событием **onClick**. Давайте начнем с создания подпредставления для одного яблока с нашей стандартной функцией Backbone `extend()`:

```
1   ...
2   var appleItemView = Backbone.View.extend({
3     tagName: 'li',
4     template: _.template(''
5           +'<a href="#apples/<%=name%>" target="_blank">'
6          +'<%=name%>'
7          +'</a> <a class="add-to-cart" href="#">buy</a>'),
8     events: {
9       'click .add-to-cart': 'addToCart'
10    },
11    render: function() {
12      this.$el.html(this.template(this.model.attributes));
13    },
14    addToCart: function(){
15      this.model.collection.trigger('addToCart', this.model);
16    }
```

[10] https://github.com/azat-co/rpjs/tree/master/backbone/subview

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

Теперь мы можем заполнить объект свойствами/методами **tagName**, **template**, **events**, **render** and **addToCart**.

```
  ...
  tagName: 'li',
  ...
```

tagName автоматически позволяет Backbone.js создать HTML-элемент с указанным именем тега, в этом случае `<li>` — элемент списка. Он будет представлением одного яблока, строка в нашем списке.

```
  ...
  template: _.template(''
         +'<a href="#apples/<%=name%>" target="_blank">'
        +'<%=name%>'
        +'</a> <a class="add-to-cart" href="#">buy</a>'),
  ...
```

Шаблон это просто строка с инструкциями Undescore.js. Они завернуты в символы `<%` и `%>`. `<%=` означает просто вывести значение. Тот же самый код можно записать с обратными слешами:

```
  ...
  template: _.template('\
         <a href="#apples/<%=name%>" target="_blank">\
        <%=name%>\
        </a> <a class="add-to-cart" href="#">buy</a>\
        '),
  ...
```

Каждый `<li>` будет иметь два якорных элемена (`<a>`), ссылки на детальный просмотр яблока (`#apples/:appleName`) и кнопку **buy**. Теперь мы собираемся привязать слушатель события к кнопке **buy**:

```
  ...
  events: {
    'click .add-to-cart': 'addToCart'
  },
  ...
```

Синтаксис следует этому правилу:

```
event + jQuery element selector: function name
```

И ключ и значение (правая и левая части, разделенны двоеточием) являются строками. Например:

```
'click .add-to-cart': 'addToCart'
```

или

```
'click #load-more': 'loadMoreData'
```

Для визуализации каждого элемента в списке, мы будем использовать функцию jQuery `html()` на объекте jQuery `this.$el`, которым является HTML-элемент `<li>` из нашего атрибута **tagName**:

```
  ...
  render: function() {
    this.$el.html(this.template(this.model.attributes));
  },
  ...
```

addToCart будет использовать функцию `trigger()`, чтобы уведомить коллекцию, что эта конкретная модель (apple) выбрана для покупки пользователем:

```
  ...
    addToCart: function(){
      this.model.collection.trigger('addToCart', this.model);
    }
  ...
```

Вот полный код **appleItemView** класса Backbone View:

```
  ...
  var appleItemView = Backbone.View.extend({
    tagName: 'li',
    template: _.template(''
           +'<a href="#apples/<%=name%>" target="_blank">'
          +'<%=name%>'
          +'</a> <a class="add-to-cart" href="#">buy</a>'),
    events: {
      'click .add-to-cart': 'addToCart'
    },
    render: function() {
      this.$el.html(this.template(this.model.attributes));
    },
    addToCart: function(){
      this.model.collection.trigger('addToCart', this.model);
    }
  });
  ...
```

Изи-бризи! Но как насчет главного представления, которое, как предполагается, рендерит все из наших элементов (яблок) и предоставляют обертку `<ul>`, контейнер для `<li>`? Мы должны изменить и улучшить наш **homeView**.

Начнем с того, что мы можем добавить дополнительные свойства строкового типа, понимаемые jQuery как селекторы в **homeView**:

```
  ...
  el: 'body',
  listEl: '.apples-list',
  cartEl: '.cart-box',
  ...
```

Мы можем использовать эти свойства в шаблоне или просто захардкодить их (мы срефакторим наш код позже) в **homeView**:

```
  ...
  template: _.template('Apple data: \
    <ul class="apples-list">\
    </ul>\
    <div class="cart-box"></div>'),
  ...
```

Функция **initialize** будет вызываться при создании **homeView** (`new homeView()`) — в нем мы рендерим наш шаблон (нашей любимой функцией `html()`) и привязываем слушатель события в коллекцию (которая представляет собой набор моделей яблок):

```
  ...
    initialize: function() {
      this.$el.html(this.template);
      this.collection.on('addToCart', this.showCart, this);
    },
  ...
```

Синтаксис для привязки событий описан в предыдущем разделе. В сущности, это вызов функции `showCart()` из **homeView**. В этой функции мы добавляем **appleName** в корзину (вместе с переносом строки `<br/>`):

```
  ...
    showCart: function(appleModel) {
      $(this.cartEl).append(appleModel.attributes.name+'<br/>');
    },
  ...
```

Наконец, вот наш долгожданный метод `render()`, в котором мы перебираем каждую модель в коллекции (каждое яблоко), создаем **appleItemView** для каждого яблока, создаем элемент `<li>` для каждого яблока и добавляем этот элемент к **view.listEl** — элемент `<ul>` с классом apples-list:

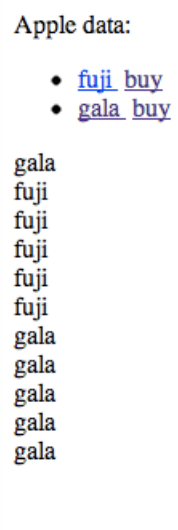
```
...
render: function(){
  view = this;
  //теперь мы можем использовать view внутри замыкания
  this.collection.each(function(apple){
    var appleSubView = new appleItemView({model:apple});
    // создает подпредставление с моделью apple
    appleSubView.render();
    // компилирует шаблон и данные одного яблока
    $(view.listEl).append(appleSubView.$el);
    //добавляем объект jQuery из данного
    //apple в DOM-элемент apples-list
  });
}
...
```

Давайте убедимся, что мы ничего не пропустили в Backbone-представлении **homeView**:

```
...
var homeView = Backbone.View.extend({
  el: 'body',
  listEl: '.apples-list',
  cartEl: '.cart-box',
  template: _.template('Apple data: \
    <ul class="apples-list">\
    </ul>\
    <div class="cart-box"></div>'),
  initialize: function() {
    this.$el.html(this.template);
    this.collection.on('addToCart', this.showCart, this);
  },
  showCart: function(appleModel) {
    $(this.cartEl).append(appleModel.attributes.name+'<br/>');
  },
      render: function(){
        view = this;  //теперь мы можем использовать view внутри замыкания
        this.collection.each(function(apple){
              var appleSubView = new appleItemView({model:apple});
              // создает подпредставление с моделью apple
              appleSubView.render();
              // компилирует шаблон и данные одного яблока
              $(view.listEl).append(appleSubView.$el);
              //добавляем объект jQuery из данного
              //apple в DOM-элемент apples-list
        });
      }
```

```
29    });
30    ...
```

Вы должны мочь кликнуть на buy и корзина будет заполняться яблоками на ваш выбор. Просмотр одного яблока больше не требует ввода его имени в адресную строку браузера. Мы можем кликнуть на название и открыть новое окно с детальным просмотром.



Список яблок, отрендереных подпредставлениями.

Используя подпредставления, мы повторно используем шаблоны для всех элементов (apples) и привязываем определенные события для каждого из них. Эти события достаточно умны, чтобы передать информацию о модели другим объектам: представлениям и коллекциям.

На всякий случай, вот полный код примера для подпредставлений, который также доступен по ссылке rpjs/backbone/subview/index.html[11]:

```
1   <!DOCTYPE>
2   <html>
3   <head>
4     <script src="jquery.js"></script>
5     <script src="underscore.js"></script>
6     <script src="backbone.js"></script>
7
8     <script>
9      var appleData = [
10       {
11         name: "fuji",
12         url: "img/fuji.jpg"
13       },
```

[11] https://github.com/azat-co/rpjs/blob/master/backbone/subview/index.html

```
14        {
15          name: "gala",
16          url: "img/gala.jpg"
17        }
18      ];
19      var app;
20      var router = Backbone.Router.extend({
21        routes: {
22          "": "home",
23          "apples/:appleName": "loadApple"
24        },
25        initialize: function(){
26          var apples = new Apples();
27          apples.reset(appleData);
28          this.homeView = new homeView({collection: apples});
29          this.appleView = new appleView({collection: apples});
30        },
31        home: function(){
32          this.homeView.render();
33        },
34        loadApple: function(appleName){
35          this.appleView.loadApple(appleName);
36
37        }
38      });
39      var appleItemView = Backbone.View.extend({
40        tagName: 'li',
41        // template: _.template(''
42        //    +'<a href="#apples/<%=name%>" target="_blank">'
43        //    +'<%=name%>'
44        //    +'</a> <a class="add-to-cart" href="#">buy</a>'),
45        template: _.template('\
46               <a href="#apples/<%=name%>" target="_blank">\
47              <%=name%>\
48              </a> <a class="add-to-cart" href="#">buy</a>\
49              '),
50
51        events: {
52          'click .add-to-cart': 'addToCart'
53        },
54        render: function() {
55          this.$el.html(this.template(this.model.attributes));
56        },
57        addToCart: function(){
58          this.model.collection.trigger('addToCart', this.model);
```

```
59        }
60      });
61
62      var homeView = Backbone.View.extend({
63        el: 'body',
64        listEl: '.apples-list',
65        cartEl: '.cart-box',
66        template: _.template('Apple data: \
67          <ul class="apples-list">\
68          </ul>\
69          <div class="cart-box"></div>'),
70        initialize: function() {
71          this.$el.html(this.template);
72          this.collection.on('addToCart', this.showCart, this);
73        },
74        showCart: function(appleModel) {
75          $(this.cartEl).append(appleModel.attributes.name+'<br/>');
76        },
77        render: function(){
78          view = this; //теперь мы можем использовать view внутри замыкания
79          this.collection.each(function(apple){
80            var appleSubView = new appleItemView({model:apple});
81            // создает подпредставление с моделью apple
82            appleSubView.render();
83            // компилирует шаблон и данные одного яблока
84            $(view.listEl).append(appleSubView.$el);
85            //добавляем объект jQuery из данного
86            //apple в DOM-элемент apples-list
87          });
88        }
89      });
90
91      var Apples = Backbone.Collection.extend({
92      });
93
94      var appleView = Backbone.View.extend({
95        initialize: function(){
96          this.model = new (Backbone.Model.extend({}));
97          this.model.on('change', this.render, this);
98          this.on('spinner',this.showSpinner, this);
99        },
100       template: _.template('<figure>\
101                 <img src="<%= attributes.url %>"/>\
102                 <figcaption><%= attributes.name %></figcaption>\
103               </figure>'),
```

```
104       templateSpinner: '<img src="img/spinner.gif" width="30"/>',
105       loadApple:function(appleName){
106         this.trigger('spinner');
107         var view = this;
108         //мы должны иметь доступ к this внутри замыкания
109         setTimeout(function(){
110         //симулирует лаги при
111         //извлечение данных из удаленного сервера
112           view.model.set(view.collection.where({
113             name:appleName
114           })[0].attributes);
115         },1000);
116       },
117       render: function(appleName){
118         var appleHtml = this.template(this.model);
119         $('body').html(appleHtml);
120       },
121       showSpinner: function(){
122         $('body').html(this.templateSpinner);
123       }
124     });
125
126     $(document).ready(function(){
127       app = new router;
128       Backbone.history.start();
129     })
130
131   </script>
132 </head>
133 <body>
134   <div></div>
135 </body>
136 </html>
```

Ссылка на отдельный элемент, например, `collections/index.html#apples/fuji`, также независимо должна работать при введении её в адресной строке браузера.

4.5 Рефакторинг

В этот момент вы, вероятно, задаетесь вопросом, какова выгода от использования фреймворка, если у нас классы, объекты и элементы с различными функциональными возможностями находятся в *одном* файле. Это было сделано с целью соблюдения принципа *Keep it Simple Stupid* (KISS, англ. `делай это проще, тупица`).

Чем больше приложение, тем больше боли в неорганизованной кодовой базе. Давайте

разберем наше приложение на несколько файлов, где каждый файл будет одним из этих типов:

- view
- template
- router
- collection
- model

Давайте пропишем эти скрипты в нашем **index.html** в head или body, как отмечалось ранее:

```
  <script src="apple-item.view.js"></script>
  <script src="apple-home.view.js"></script>
  <script src="apple.view.js"></script>
  <script src="apples.js"></script>
  <script src="apple-app.js"></script>
```

Имена не должны следовать соглашению тире и точек, до тех пор, пока легко определить, что каждый файл делает.

Теперь, давайте скопируем наши объекты/классы в соответствующие файлы.

Наш главный файл **index.html** должен выглядеть очень минималистичным:

```
<!DOCTYPE>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="underscore.js"></script>
  <script src="backbone.js"></script>

  <script src="apple-item.view.js"></script>
  <script src="apple-home.view.js"></script>
  <script src="apple.view.js"></script>
  <script src="apples.js"></script>
  <script src="apple-app.js"></script>

</head>
<body>
  <div></div>
</body>
</html>
```

Другие файлы, так же содержат код, соответствующий их именам.

Содержание **apple-item.view.js**:

```
var appleView = Backbone.View.extend({
  initialize: function(){
    this.model = new (Backbone.Model.extend({}));
    this.model.on('change', this.render, this);
    this.on('spinner',this.showSpinner, this);
  },
  template: _.template('<figure>\
            <img src="<%= attributes.url %>"/>\
            <figcaption><%= attributes.name %></figcaption>\
          </figure>'),
  templateSpinner: '<img src="img/spinner.gif" width="30"/>',

  loadApple:function(appleName){
    this.trigger('spinner');
    var view = this;
    //мы должны иметь доступ к this внутри замыкания
    setTimeout(function(){
    //симулирует лаги при
    //извлечение данных из удаленного сервера
      view.model.set(view.collection.where({
        name:appleName
      })[0].attributes);
    },1000);

  },

  render: function(appleName){
    var appleHtml = this.template(this.model);
    $('body').html(appleHtml);
  },
  showSpinner: function(){
    $('body').html(this.templateSpinner);
  }

});
```

Файл **apple-home.view.js** имеет расширение объекта **homeView**:

```
var homeView = Backbone.View.extend({
  el: 'body',
  listEl: '.apples-list',
  cartEl: '.cart-box',
  template: _.template('Apple data: \
    <ul class="apples-list">\
    </ul>\
    <div class="cart-box"></div>'),
  initialize: function() {
    this.$el.html(this.template);
    this.collection.on('addToCart', this.showCart, this);
  },
  showCart: function(appleModel) {
    $(this.cartEl).append(appleModel.attributes.name+'<br/>');
  },
  render: function(){
    view = this; //теперь мы можем использовать view внутри замыкания
    this.collection.each(function(apple){
      var appleSubView = new appleItemView({model:apple});
      // создает подпредставление с моделью apple
      appleSubView.render();
      // компилирует шаблон и данные одного яблока
      $(view.listEl).append(appleSubView.$el);
      //добавляем объект jQuery из данного
      //apple в DOM-элемент apples-list
    });
  }
});
```

Файл **apple.view.js** содержит главный список яблок:

```
var appleView = Backbone.View.extend({
  initialize: function(){
    this.model = new (Backbone.Model.extend({}));
    this.model.on('change', this.render, this);
    this.on('spinner',this.showSpinner, this);
  },
  template: _.template('<figure>\
          <img src="<%= attributes.url %>"/>\
          <figcaption><%= attributes.name %></figcaption>\
        </figure>'),
  templateSpinner: '<img src="img/spinner.gif" width="30"/>',
  loadApple:function(appleName){
    this.trigger('spinner');
    var view = this;
    //мы должны иметь доступ к this внутри замыкания
```

```
16        setTimeout(function(){
17        //симулирует лаги при
18        //извлечение данных из удаленного сервера
19          view.model.set(view.collection.where({
20            name:appleName
21          })[0].attributes);
22        },1000);
23      },
24      render: function(appleName){
25        var appleHtml = this.template(this.model);
26        $('body').html(appleHtml);
27      },
28      showSpinner: function(){
29        $('body').html(this.templateSpinner);
30      }
31    });
```

apples.js — это пустая коллекция:

```
1     var Apples = Backbone.Collection.extend({
2     });
```

apple-app.js — это главный файл приложения с данными, маршрутизатором и командой на запуск:

```
1     var appleData = [
2        {
3          name: "fuji",
4          url: "img/fuji.jpg"
5        },
6        {
7          name: "gala",
8          url: "img/gala.jpg"
9        }
10    ];
11    var app;
12    var router = Backbone.Router.extend({
13      routes: {
14        '': 'home',
15        'apples/:appleName': 'loadApple'
16      },
17      initialize: function(){
18        var apples = new Apples();
19        apples.reset(appleData);
20        this.homeView = new homeView({collection: apples});
21        this.appleView = new appleView({collection: apples});
```

```
22        },
23        home: function(){
24          this.homeView.render();
25        },
26        loadApple: function(appleName){
27          this.appleView.loadApple(appleName);
28        }
29      });
30      $(document).ready(function(){
31        app = new router;
32        Backbone.history.start();
33      })
```

Теперь давайте попробуем открыть приложение. Оно должен работать точно так же, как и в предыдущем примере подпредставлений.

Это лучший способ организации кода, но все еще далеко от идеала, потому что мы все еще имеем HTML-шаблоны непосредственно в коде JavaScript. Проблема в том, что дизайнеры и разработчики не могут работать в одних и тех же ϕайлах, а любые изменения в представлении требуют внесения изменений в основной код приложения.

Мы можем добавить еще несколько JS-ϕайлов в наш ϕайл **index.html**:

```
1   <script src="apple-item.tpl.js"></script>
2   <script src="apple-home.tpl.js"></script>
3   <script src="apple-spinner.tpl.js"></script>
4   <script src="apple.tpl.js"></script>
```

Как правило, один Backbone view имеет один шаблон, но в случае с нашим **appleView**, в детальном представлении яблока в отдельном окне, у нас есть крутилка, GIF-анимация загрузки.

Содержимое ϕайлов просто глобальные переменные, которым присваиваются некоторые строковые значения. Впоследствии мы можем использовать эти переменные в наших представлениях, когда мы вызываем вспомогательный метод Underscore.js **_.template()**.

Файл **apple-item.tpl.js**:

```
1   var appleItemTpl = '\
2        <a href="#apples/<%=name%>" target="_blank">\
3       <%=name%>\
4       </a> <a class="add-to-cart" href="#">buy</a>\
5       ';
```

Файл **apple-home.tpl.js**:

```
var appleHomeTpl = 'Apple data: \
        <ul class="apples-list">\
        </ul>\
        <div class="cart-box"></div>';
```

Файл **apple-spinner.tpl.js**:

```
var appleSpinnerTpl = '<img src="img/spinner.gif" width="30"/>';
```

Файл **apple.tpl.js**:

```
var appleTpl = '<figure>\
                <img src="<%= attributes.url %>"/>\
                <figcaption><%= attributes.name %></figcaption>\
              </figure>';
```

Попробуйте запустить приложение сейчас. Полный код находится в папке rpjs/backbone/refactor[12].

Как вы можете видеть в предыдущем примере, мы использовали глобальную область видимости переменных (без ключевого слова `window`).

Предупреждение

Будьте осторожны, когда вводите много переменных в глобальное пространство имен (ключевое слово `window`). Это может привести к конфликтам и другим непредсказуемым последствиям. Например, если вы написали библиотеку с открытым исходным кодом и другие разработчики начали использовать методы и свойства напрямую, вместо использования интерфейса, что произойдет позже, когда вы решите, наконец, удалить/отказаться от этих глобальных утечек? Чтобы предотвратить это, правильно написанные библиотеки и приложения используют замыкания[13].

Пример использования замыкания и определения глобальной переменной модуля:

```
(function() {
  var apple= function() {
  ...//делаем что-то полезное, например, возврат объекта apple
  };
  window.Apple = apple;
}())
```

Или в случае, когда требуется получить доступ к объекту приложения (что создает зависимость от объекта):

[12] https://github.com/Азат-co/rpjs/tree/master/backbone/refactor

[13] https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide_ru/Замыкания

```
(function() {
  var app = this.app;
  //эквивалент window.appliation
  //в случае, если нужна зависимость (app)
  this.apple = function() {
  ...//возврат объекта/класса apple
  //используем переменную приложения
  }
  // эквивалент window.apple = function(){...};
}())
```

Как вы можете видеть, мы создали функцию и вызвали ее немедленно, обернув все в скобки ().

4.6 AMD и Require.js для разработки

AMD позволяет нам организовывать разрабатываемый код в модули, управлять зависимостями и загружать их асинхронно. Эта статья хорошо объясняет, почему AMD это хорошая вещь: WHY AMD?[14]

Запустите свой локальный HTTP-сервер, например, MAMP[15].

Давайте улучшим наш код с помощью библиотеки Require.js.

Наш **index.html** сократится еще больше:

```
<!DOCTYPE>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="underscore.js"></script>
  <script src="backbone.js"></script>
  <script src="require.js"></script>
  <script src="apple-app.js"></script>
</head>
<body>
  <div></div>
</body>
</html>
```

Мы только добавили библиотеки и единственный файл JavaScript с нашим приложением. Этот файл имеет следующую структуру:

[14] http://requirejs.org/docs/whyamd.html

[15] http://www.mamp.info/en/index.html

```
require([...],function(...){...});
```

Или более понятно:

```
require([
  'name-of-the-module',
  ...
  'name-of-the-other-module'
  ],function(referenceToModule, ..., referenceToOtherModule){
  ...//некоторый полезный код
  referenceToModule.someMethod();
});
```

В основном мы говорим браузеру загрузить файлы из массива имен файлов (первый параметр функции `require()`), и затем передаем наши модули из тех файлов в анонимную функцию обратного вызова (второй аргумент), как переменные. Внутри главной функции (анонимного коллбэка) мы можем использовать наши модули ссылаясь на эти переменные. Поэтому наш **apple-app.js** превращается в:

```
  require([
    'apple-item.tpl', //можно исопльзовать shim-плагин
    'apple-home.tpl',
    'apple-spinner.tpl',
    'apple.tpl',
    'apple-item.view',
    'apple-home.view',
    'apple.view',
    'apples'
  ],function(
    appleItemTpl,
    appleHomeTpl,
    appleSpinnerTpl,
    appleTpl,
    appelItemView,
    homeView,
    appleView,
    Apples
    ){
   var appleData = [
      {
        name: "fuji",
        url: "img/fuji.jpg"
      },
      {
        name: "gala",
```

```
27          url: "img/gala.jpg"
28        }
29      ];
30      var app;
31      var router = Backbone.Router.extend({
32        //check if need to be required
33        routes: {
34          '': 'home',
35          'apples/:appleName': 'loadApple'
36        },
37        initialize: function(){
38          var apples = new Apples();
39          apples.reset(appleData);
40          this.homeView = new homeView({collection: apples});
41          this.appleView = new appleView({collection: apples});
42        },
43        home: function(){
44          this.homeView.render();
45        },
46        loadApple: function(appleName){
47          this.appleView.loadApple(appleName);
48
49        }
50      });
51
52      $(document).ready(function(){
53        app = new router;
54        Backbone.history.start();
55      })
56  });
```

Мы поместили весь код внутрь функции, которая является вторым аргумент `require()`, перечислили модули по их именам файла, и использовали зависимости с помощью соответствующих параметров. Теперь мы должны определить сам модуль. Вот так мы можем сделать это методом `define()`:

```
1  define([...],function(...){...})
```

Смысл аналогичен функции `require()`: зависимости — это строки имен файлов (и пути) в массиве, который передается в качестве первого аргумента. Второй аргумент — это основная функция, которая принимает другие библиотеки как параметры (важен порядок параметров и модулей в массиве):

```
define(['name-of-the-module'],function(nameOfModule){
  var b = nameOfModule.render();
  return b;
})
```

Примечание

Нет необходимости дописывать **.js** к именам файлов. Require.js делает это автоматически. Shim-плагин используется для импорта текстовых файлов, таких как HTML-шаблоны.

Начнем с шаблонов и конвертируем их в модули Require.js.
Новый файл **apple-item.tpl.js**:

```
define(function() {
  return '\
             <a href="#apples/<%=name%>" target="_blank">\
            <%=name%>\
            </a> <a class="add-to-cart" href="#">buy</a>\
            '
});
```

Файл **apple-home.tpl**:

```
define(function(){
  return 'Apple data: \
        <ul class="apples-list">\
        </ul>\
        <div class="cart-box"></div>';
});
```

Файл **apple-spinner.tpl.js**:

```
define(function(){
  return '<img src="img/spinner.gif" width="30"/>';
});
```

Файл **apple.tpl.js**:

```
define(function(){
  return '<figure>\
          <img src="<%= attributes.url %>"/>\
          <figcaption><%= attributes.name %></figcaption>\
        </figure>';
 });
```

Файл **apple-item.view.js**:

```
define(function() {
  return '\
             <a href="#apples/<%=name%>" target="_blank">\
            <%=name%>\
            </a> <a class="add-to-cart" href="#">buy</a>\
            '
});
```

В файле **apple-home.view.js**, нам нужно объявить зависимости от файлов apple-home.tpl и apple-item.view.js:

```
define(['apple-home.tpl','apple-item.view'],function(
  appleHomeTpl,
  appleItemView){
return  Backbone.View.extend({
      el: 'body',
      listEl: '.apples-list',
      cartEl: '.cart-box',
      template: _.template(appleHomeTpl),
      initialize: function() {
        this.$el.html(this.template);
        this.collection.on('addToCart', this.showCart, this);
      },
      showCart: function(appleModel) {
        $(this.cartEl).append(appleModel.attributes.name+'<br/>');
      },
      render: function(){
        view = this; //теперь мы можем использовать view внутри замыкания
        this.collection.each(function(apple){
          var appleSubView = new appleItemView({model:apple});
          // создает подпредставление с моделью apple
          appleSubView.render();
          // компилирует шаблон и данные одного яблока
          $(view.listEl).append(appleSubView.$el);
          //добавляем объект jQuery из данного
          //apple в DOM-элемент apples-list
        });
```

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

Файл **apple.view.js** зависит от 2 щаблонов:

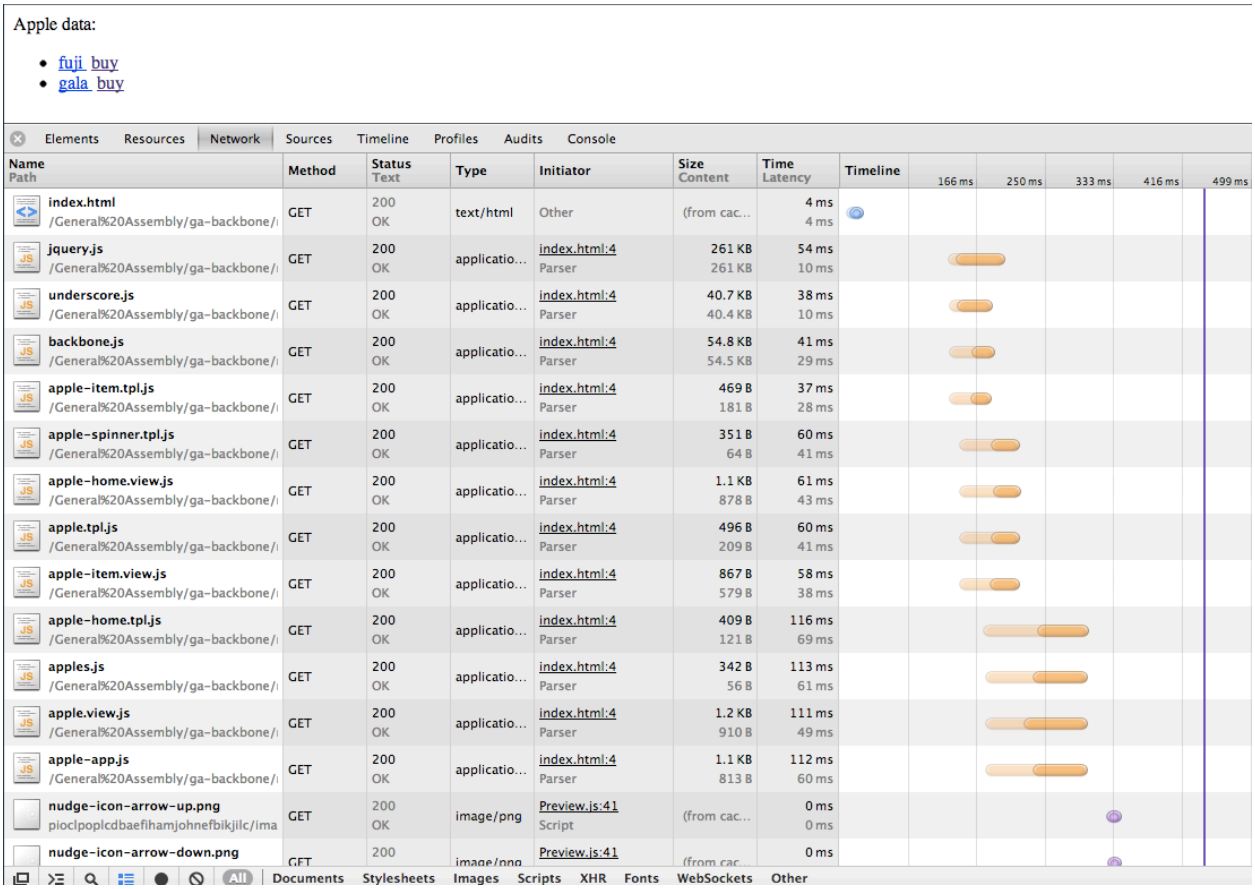
```
define([
  'apple.tpl',
  'apple-spinner.tpl'
],function(appleTpl,appleSpinnerTpl){
  return  Backbone.View.extend({
    initialize: function(){
      this.model = new (Backbone.Model.extend({}));
      this.model.on('change', this.render, this);
      this.on('spinner',this.showSpinner, this);
    },
    template: _.template(appleTpl),
    templateSpinner: appleSpinnerTpl,
    loadApple:function(appleName){
      this.trigger('spinner');
      var view = this;
      //мы должны иметь доступ к this внутри замыкания
      setTimeout(function(){
      //симулирует лаги при
      //извлечение данных из удаленного сервера
        view.model.set(view.collection.where({
          name:appleName
        })[0].attributes);
      },1000);
    },
    render: function(appleName){
      var appleHtml = this.template(this.model);
      $('body').html(appleHtml);
    },
    showSpinner: function(){
      $('body').html(this.templateSpinner);
    }
  });
});
```

Файл **apples.js**:

```
define(function(){
    return Backbone.Collection.extend({})
});
```

Я надеюсь, что вы поняли шаблон действий. Весь наш код разбит на отдельные файлы, основываясь на логике (например, класс представления, класс коллекции, шаблон). Основной файл загружает все зависимости функцией `require()`. Если нам нужен какой-либо модуль в неосновном файле, то мы можем запросить его в методе `define()`. Как правило, в модулях мы хотим возвратить какой-либо объект, например, в шаблонах мы возвращаем строки, в представлениях мы возвращаем классы/объекты Backbone View.

Попробуйте запустить пример из папки rpjs/backbone/amd[16]. Визуально не должно быть никаких изменений. Если открыть вкладку Network в Developers Tool, вы можете увидеть разницу в загрузке файлов. Старый файл rpjs/backbone/refactor/index.html[17] загружает JS-скрипты в последовательной манере, а новый файл rpjs/backbone/amd/index.html[18] загружает их параллельно.

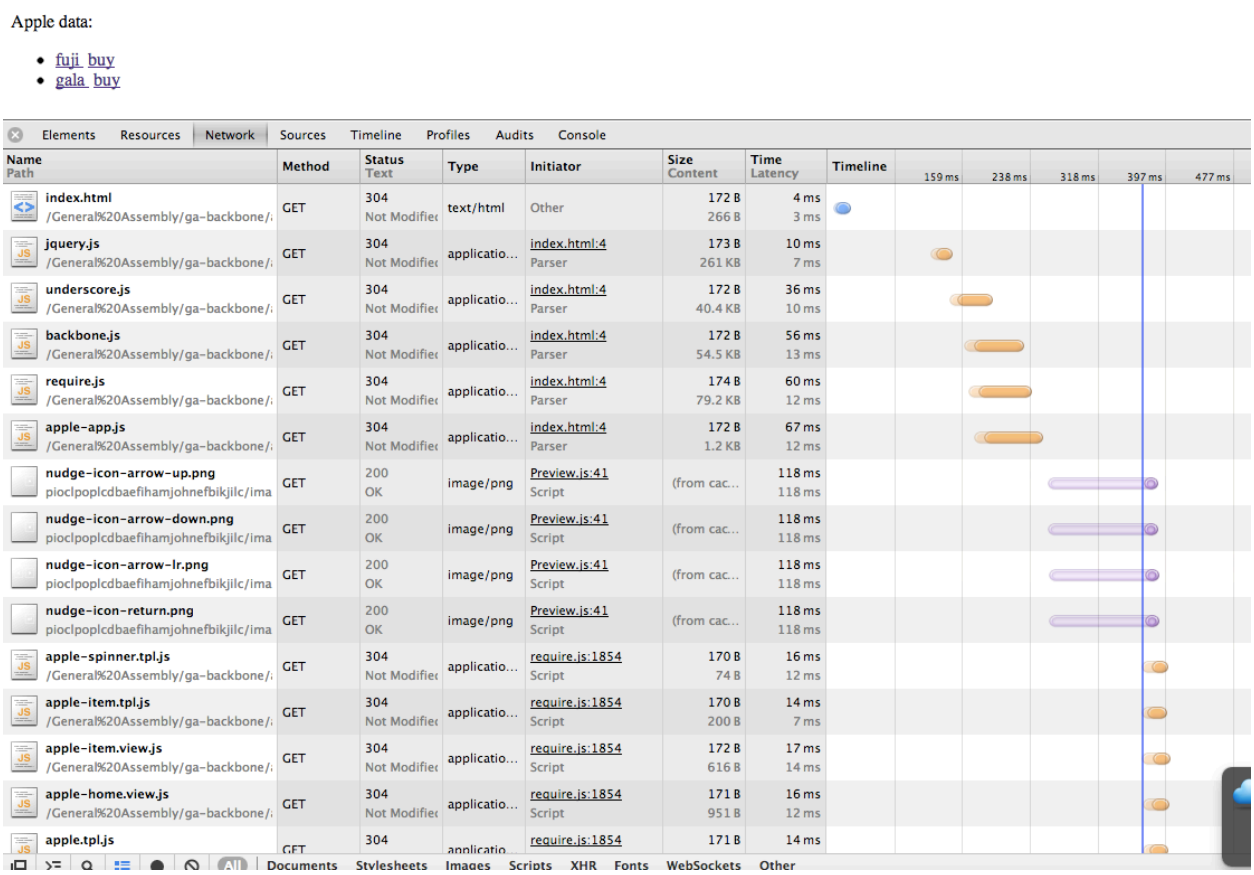


Старый файл rpjs/backbone/refactor/index.html

[16] https://github.com/azat-co/rpjs/tree/master/backbone/amd

[17] https://github.com/azat-co/rpjs/blob/master/backbone/refactor/index.html

[18] https://github.com/azat-co/rpjs/blob/master/backbone/amd/index.html



Новый файл rpjs/backbone/amd/index.html

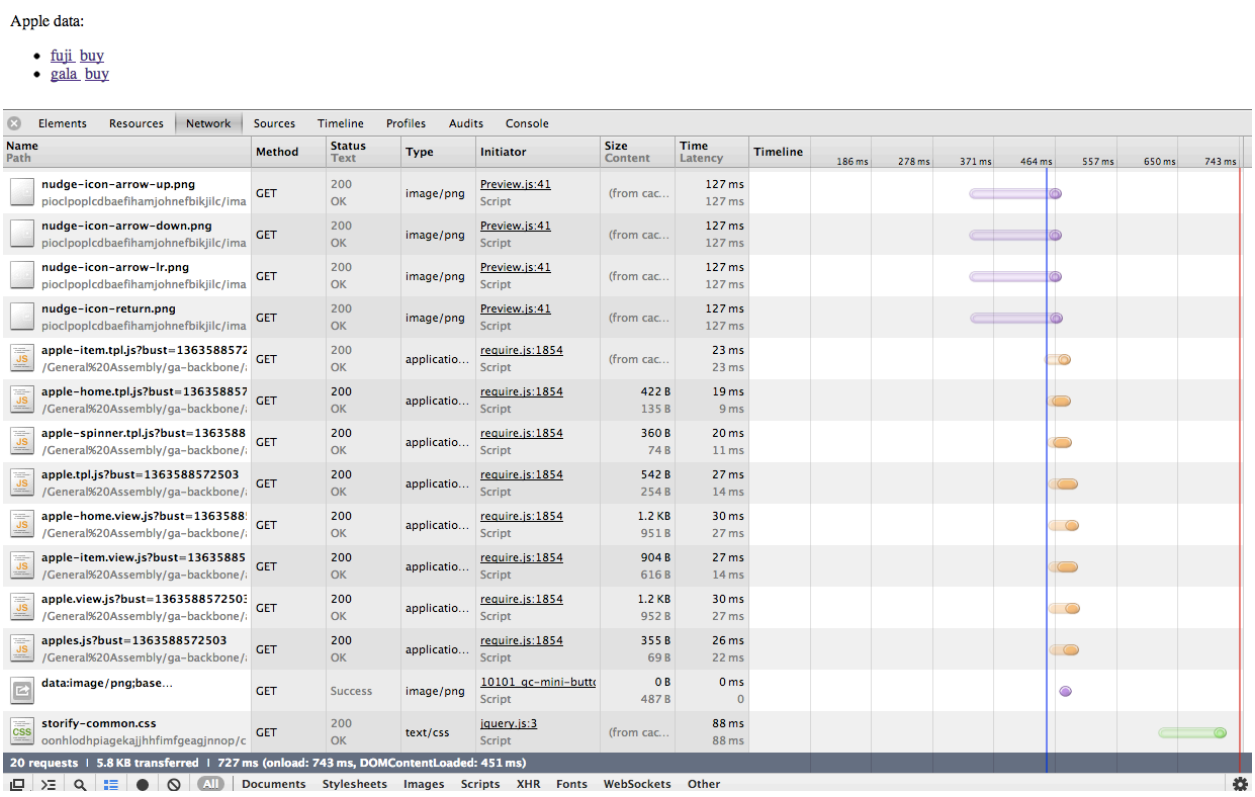
Require.js имеет много опций, которые определяются через вызов `requirejs.config()` на верхнем уровне HTML-страницы. Более подробную информацию можно найти по адресу requirejs.org/docs/api.html#config[19].

Давайте добавим параметр bust к нашему примеру. Аргумент bust будет добавляться к URL-адресу каждого файла для предотвращения кэширования файлов браузером. Идеально подходит для разработки и ужасно для производства. :-)

Добавьте это в файл **apple-app.js** впереди всего остального:

```
requirejs.config({
  urlArgs: "bust=" +  (new Date()).getTime()
});
require([
...
```

[19] http://requirejs.org/docs/api.html#config



Вкладка Network с параметром bust

Обратите внимание, что каждый файл по запросу теперь имеет статус 200 вместо 304 (not modified).

4.7 Require.js для продакшена

Мы будем использовать Node Package Manager (NPM) для установки библиотеки **requirejs** (это не опечатка, нет точки в имени). В папке вашего проекта, выполните эту команду в терминале:

```
$ npm install requirejs
```

Или добавьте `-g` для глобальной установки:

```
$ npm install -g requirejs
```

Создайте файл **app.build.js**:

```
({
    appDir: "./js",
    baseUrl: "./",
    dir: "build",
    modules: [
        {
            name: "apple-app"
        }
    ]
})
```

Переместите файлы скриптов в папку `js` (свойство appDir). Собираемые файлы будут помещены в папку `build` (параметр dir). Для получения дополнительной информации по сборке файла ознакомтесь с этим *подробным* примером с комментариями: https://github.com/jrburke/r.js/blob/master/build/example.build.js.

Теперь все должно быть готово для сборки одного гигантского JavaScript-файла, который будет иметь все наши зависимостей/модули:

```
$ r.js -o app.build.js
```

или

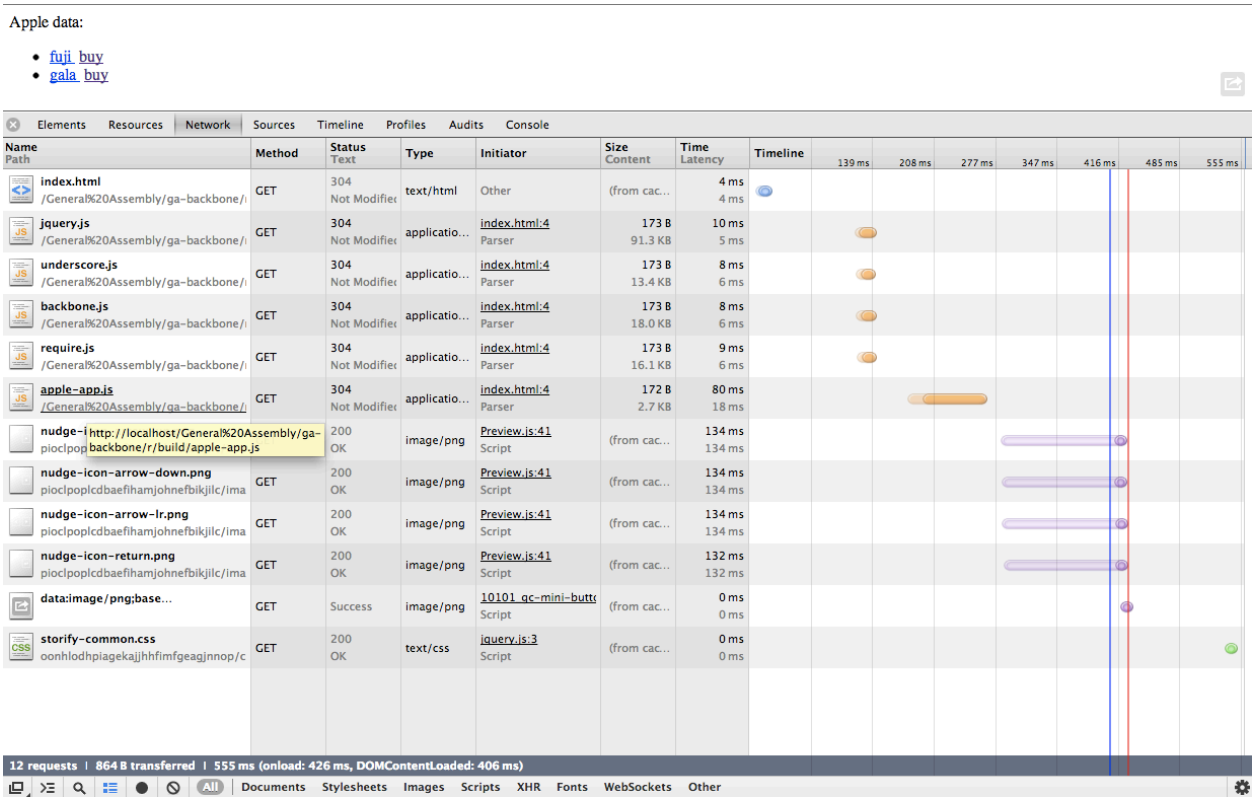
```
$ node_modules/requirejs/bin/r.js -o app.build.js
```

Вы должны получить список обработанных файлов r.js.

```
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple-app.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple-home.tpl.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple-home.view.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple-item.tpl.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple-item.view.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple-spinner.tpl.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple.tpl.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple.view.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apples.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/backbone.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/jquery.js
toTransport skipping /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/node_modules/.bin/r.js: Error: Line 1: Unexpe
cted token ILLEGAL
Error: Cannot parse file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/node_modules/.bin/r.js for comments. Ski
pping it. Error is:
Error: Line 1: Unexpected token ILLEGAL
toTransport skipping /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/node_modules/requirejs/bin/r.js: Error: Line
1: Unexpected token ILLEGAL
Error: Cannot parse file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/node_modules/requirejs/bin/r.js for comm
ents. Skipping it. Error is:
Error: Line 1: Unexpected token ILLEGAL
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/node_modules/requirejs/require.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/require.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/underscore.js

apple-app.js
---------------
apple-item.tpl.js
apple-home.tpl.js
apple-spinner.tpl.js
apple.tpl.js
apple-item.view.js
apple-home.view.js
apple.view.js
apples.js
apple-app.js

 r git:(master) ✗ $ node_modules/requirejs/bin/r.js -o app.build.js
```

Список обработанных файлов r.js.

Откройте **index.html** из папки сборки в окне браузера, и проверьте, показывает ли сейчас вкладка Network какие-либо улучшения только с одним запросом/файлом для загрузки.



Улучшение производительности с одним запросом/файлом для загрузки.

Для получения дополнительной информации, ознакомтесь с официальной документацией r.js requirejs.org/docs/optimization.html[20].

Код примера доступен в папках rpjs/backbone/r[21] и rpjs/backbone/r/build[22].

Для минификации JS-файлов (уменьшения размера файлов), мы можем использовать модуль Uglify2[23]. Чтобы установить ее с NPM, используйте:

```
$ npm install uglify-js
```

Затем обновите файл **app.build.js** с помощью свойства `optimize: "uglify2"`:

[20] http://requirejs.org/docs/optimization.html

[21] https://github.com/azat-co/rpjs/tree/master/backbone/r

[22] https://github.com/azat-co/rpjs/tree/master/backbone/r/build

[23] https://github.com/mishoo/UglifyJS2

```
({
    appDir: "./js",
    baseUrl: "./",
    dir: "build",
    optimize: "uglify2",
    modules: [
        {
            name: "apple-app"
        }
    ]
})
```

Запустите r.js с параметрами:

```
$ node_modules/requirejs/bin/r.js -o app.build.js
```

Вы должны получить что-то вроде этого:

```
define("apple-item.tpl",[],function(){return'             <a href="#apples/<%=name%>" targ\
et="_blank">            <%=name%>            </a> <a class="add-to-cart" href="#">buy\
</a>            '}),define("apple-home.tpl",[],function(){return'Apple data:         <ul c\
lass="apples-list">        </ul>        <div class="cart-box"></div>'}),define("apple-spin\
ner.tpl",[],function(){return'<img src="img/spinner.gif" width="30"/>'}),define("apple.tpl\
",[],function(){return'<figure>                                  <img src="<%= attributes.url \
%>"/>                              <figcaption><%= attributes.name %></figcaption>        \
                    </figure>'}),define("apple-item.view",["apple-item.tpl"],function(e){r\
eturn Backbone.View.extend({tagName:"li",template:_.template(e),events:{"click .add-to-car\
t":"addToCart"},render:function(){this.$el.html(this.template(this.model.attributes))},add\
ToCart:function(){this.model.collection.trigger("addToCart",this.model)}})}),define("apple\
-home.view",["apple-home.tpl","apple-item.view"],function(e,t){return Backbone.View.extend\
({el:"body",listEl:".apples-list",cartEl:".cart-box",template:_.template(e),initialize:fun\
ction(){this.$el.html(this.template),this.collection.on("addToCart",this.showCart,this)},s\
howCart:function(e){$(this.cartEl).append(e.attributes.name+"<br/>")},render:function(){vi\
ew=this,this.collection.each(function(e){var i=new t({model:e});i.render(),$(view.listEl).\
append(i.$el)})}})}),define("apple.view",["apple.tpl","apple-spinner.tpl"],function(e,t){r\
eturn Backbone.View.extend({initialize:function(){this.model=new(Backbone.Model.extend({})\
),this.model.on("change",this.render,this),this.on("spinner",this.showSpinner,this)},templ\
ate:_.template(e),templateSpinner:t,loadApple:function(e){this.trigger("spinner");var t=th\
is;setTimeout(function(){t.model.set(t.collection.where({name:e})[0].attributes)},1e3)},re\
nder:function(){var e=this.template(this.model);$("body").html(e)},showSpinner:function(){\
$("body").html(this.templateSpinner)}})}),define("apples",[],function(){return Backbone.Co\
llection.extend({})}),requirejs.config({urlArgs:"bust="+(new Date).getTime()}),require(["a\
pple-item.tpl","apple-home.tpl","apple-spinner.tpl","apple.tpl","apple-item.view","apple-h\
ome.view","apple.view","apples"],function(e,t,i,n,a,l,p,o){var r,s=[{name:"fuji",url:"img/\
fuji.jpg"},{name:"gala",url:"img/gala.jpg"}],c=Backbone.Router.extend({routes:{"":"home","\
apples/:appleName":"loadApple"},initialize:function(){var e=new o;e.reset(s),this.homeView\
```

```
=new l({collection:e}),this.appleView=new p({collection:e})},home:function(){this.homeView\
.render()},loadApple:function(e){this.appleView.loadApple(e)}});$(document).ready(function\
(){r=new c,Backbone.history.start()})}),define("apple-app",function(){});
```

Примечание

Файл не отформатирован специально для того, чтобы показать, как работает Uglify2. Без разрыва строк, escape-символов, код находится на одной строке. Также обратите внимание, что имена переменных и объектов сокращенны.

4.8 Super Simple Backbone Starter Kit

Для быстрого старта в разработки на Backbone.js рассмотрите возможность использования Super Simple Backbone Starter Kit[24] или похожие проекты:

- Backbone Boilerplate[25]
- Sample App with Backbone.js and Twitter Bootstrap[26]
- Больше туториалов по Backbone.js github.com/documentcloud/backbone/wiki/Tutorials%2C-blog-posts-and-example-sites[27].

[24] https://github.com/azat-co/super-simple-backbone-starter-kit
[25] http://backboneboilerplate.com/
[26] http://coenraets.org/blog/2012/02/sample-app-with-backbone-js-and-twitter-bootstrap/
[27] https://github.com/documentcloud/backbone/wiki/Tutorials%2C-blog-posts-and-example-sites

5. Backbone.js и Parse.com

Резюме: иллюстрация, как Backbone.js используется с Parse.com и его JavaScript SDK на измененном приложении Chat, предлагается перечень дополнительных функций для приложения.

"Java в JavaScript, что Car в Carpet." — Chris Heilmann[1]

Если вы уже писали сложные клиентские приложения, вы могли понять, как сложно поддерживать спагетти-код из JavaScript-коллбэков и событий пользовательского интерфейса. Backbone.js обеспечивает легкий, но мощный способ организовать вашу логику в структуре модель-представление-контроллер (Model-View-Controller, MVC). Он также имеет полезные вещи, такие как маршрутизация URL-адресов, поддержка REST API, слушатели событий и триггеры. Для ознакомления с дополнительными сведениями и пошаговыми примерами построения Backbone.js-приложения с нуля, пожалуйста, обратитесь к главе *Введенеи в Backbone.js*.

Вы можете скачать библиотеку Backbone.js из backbonejs.org[2]. Затем, после того, как вы подключили ее, как и любой другой JavaScript-файл в head (или body) вашего основного HTML-файла, вы сможете получить доступ к классу *Backbone*. Например, для создания маршрутизатора:

```
var ApplicationRouter = Backbone.Router.extend({
  routes: {
    "": "home",
    "signup": "signup",
    "*actions": "home"
  },
  initialize: function() {
    this.headerView = new HeaderView();
    this.headerView.render();
    this.footerView = new FooterView();
    this.footerView.render();
  },
  home: function() {
    this.homeView = new HomeView();
```

[1] http://christianheilmann.com/

[2] http://backbonejs.org

```
    this.homeView.render();
  },
  signup: function() {
    ...
  }
});
```

View, Models и Collections создаются так же

```
HeaderView = Backbone.View.extend({
  el: "#header",
  template: '<div>...</div>',
  events: {
    "click #save": "saveMessage"
  },
  initialize: function() {
    this.collection = new Collection();
    this.collection.bind("update", this.render, this);
    },
  saveMessage: functiton() {
    ...
  },
  render: function() {
    $(this.el).html(_.template(this.template));
  }
});

Model = Backbone.Model.extend({
  url: "/api/item"
  ...
)};

Collection = Backbone.Collection.extend({
  ...
});
```

Для более подробной информации о Backbone.js обратитесь к главе *Введение в Backbone.js*.

5.1 Chat с Parse.com: версия с JavaScript SDK и Backbone.js

Легко увидеть, что если мы будем продолжать добавлять все больше и больше кнопок типа “DELETE”, “UPDATE” и другие функциональные возможности, наша система асинхронных коллбэков будет расти. И нам надо будет значть когда для обновить представление, например, список сообщений, в зависимости от того, были ли изменения данных или нет.

Model-View-Controller (MVC) фреймворк Backbone.js может быть использован для создания сложных приложений легких в управлении и обслуживании.

Если вам было комфортно с предыдущим примером, давайте создавать поверх него с использованием фреймворка Backbone.js. Здесь мы будем шаг за шагом создавать приложение Chat с помощью Backbone.js и Parse.com JavaScript SDK. Если вам все это уже знакомо, вы можете скачать Super Simple Backbone Starter Kit github.com/azat-co/super-simple-backbone-starter-kit[3]. Интеграция с Backbone.js позволит простую реализацию обработки действий пользователя путем привязки их к асинхронным обновлениям коллекции.

Приложение доступно по адресу rpjs/sdk[4], но опять же, вам предлагается начать с нуля и попробовать написать свой собственный код, используя пример только в качестве справки.

Вот структура версии Chat с Parse.com, JavaScript SDK и Backbone.js

```
/sdk
  -index.html
  -home.html
  -footer.html
  -header.html
  -app.js
  /css
    -bootstrap-responsive.css
    -bootstrap-responsive.min.css
    -bootstrap.css
    -bootstrap.min.css
  /img
    -glyphicons-halflings-white.png
    -glyphicons-halflings.png
  /js
    -bootstrap.js
    -bootstrap.min.js
  /libs
    -require.min.js
    -text.js
```

Создайте папку, затем в ней создайте файл **index.html** со следующим содержимым:

[3] http://github.com/azat-co/super-simple-backbone-starter-kit

[4] https://github.com/azat-co/rpjs/tree/master/sdk

```
<!DOCTYPE html>
<html lang="en">
  <head>
  ...
  </head>
  <body>
  ...
  </body>
</html>
```

Скачайте нужные библиотеки или подключити их из Google API. Теперь подключите библиотеки JavaScript и таблицы стилей Twitter Bootstrap в head-элемент, наряду с другими важными, но не обязательными *meta*-элементами.

```
<head>
  <meta charset="utf-8" />
  <title></title>
  <meta name="description" content="" />
  <meta name="author" content="" />
```

Это нужно для адаптивного поведения:

```
  <meta name="viewport"
    content="width=device-width, initial-scale=1.0" />
```

Горячая привязка jQuery из Google API:

```
  <script type="text/javascript"
src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js">
  </script>
```

Приятно иметь плагин Twitter Bootstrap:

```
  <script type="text/javascript" src="js/bootstrap.min.js"></script>
```

Parse JavaScript SDK напряму подключается с Parse.com CDN:

```
  <script type="text/javascript"
    src="http://www.parsecdn.com/js/parse-1.0.5.min.js">
  </script>
```

Подключение Twitter Bootstrap CSS:

```
  <link type="text/css"
    rel="stylesheet"
    href="css/bootstrap.min.css" />
  <link type="text/css"
    rel="stylesheet"
    href="css/bootstrap-responsive.min.css" />
```

Подключение нашего JS-приложения:

```
  <script type="text/javascript" src="app.js"></script>
  </head>
```

Заполните элемент **body** каркасом Twitter Bootstrap (подробнее об этом в главе *Основы*):

```
  <body>
  <div class="container-fluid">
    <div class="row-fluid">
      <div class="span12">
        <div id="header">
        </div>
      </div>
    </div>
    <div class="row-fluid">
      <div class="span12">
        <div id="content">
        </div>
      </div>
    </div>
    <div class="row-fluid">
      <div class="span12">
        <div id="footer">
        </div>
      </div>
    </div>
  </div>
  </body>
```

Создайте файл **app.js** и поместите представления Backbone.js внутрь:

- headerView: menu и общая информация о приложении
- footerView: копирайты и контактные ссылки
- homeView: содержимое главной страницы

Мы используем синтаксис Require.js и shim-плагин для HTML-шаблонов:

```
require([
'libs/text!header.html',
'libs/text!home.html',
'libs/text!footer.html'], function (
 headerTpl,
 homeTpl,
 footerTpl) {
```

Маршрутизатор приложения с одним маршрутом:

```
 var ApplicationRouter = Backbone.Router.extend({
   routes: {
     "": "home",
     "*actions": "home"
   },
```

Прежде, чем мы что-либо сделаем, мы можем инициализировать представления, которые будут использованы в рамках приложения:

```
   initialize: function() {
     this.headerView = new HeaderView();
     this.headerView.render();
     this.footerView = new FooterView();
     this.footerView.render();
   },
```

Этот код заботится о маршруте главной страницы:

```
   home: function() {
     this.homeView = new HomeView();
     this.homeView.render();
   }
 });
```

Backbone-представление заголовка привязано к элементу `#header` и использует шаблон **headerTpl**:

```
HeaderView = Backbone.View.extend({
  el: "#header",
  templateFileName: "header.html",
  template: headerTpl,
  initialize: function() {
  },
  render: function() {
    console.log(this.template)
    $(this.el).html(_.template(this.template));
  }
});
```

Для рендеринга HTML мы используем функцию `jQuery.html()`:

```
FooterView = Backbone.View.extend({
  el: "#footer",
  template: footerTpl,
  render: function() {
    this.$el.html(_.template(this.template));
  }
});
```

Backbone-представление главной страницы использует DOM-элемент `#content`:

```
HomeView = Backbone.View.extend({
  el: "#content",
  // template: "home.html",
  template: homeTpl,
  initialize: function() {
  },
  render: function() {
    $(this.el).html(_.template(this.template));
  }
});
```

Чтобы запустить приложение, нам необходимо создать новый экземпляр и вызвать `Backbone.history.start()`:

```
  app = new ApplicationRouter();
  Backbone.history.start();
});
```

Полный код файла **app.js**:

```
require([
'libs/text!header.html',
//пример использования shim-плагина
'libs/text!home.html',
'libs/text!footer.html'], function (
 headerTpl,
 homeTpl,
 footerTpl) {
  var ApplicationRouter = Backbone.Router.extend({
    routes: {
      "": "home",
      "*actions": "home"
    },
    initialize: function() {
      this.headerView = new HeaderView();
      this.headerView.render();
      this.footerView = new FooterView();
      this.footerView.render();
    },
    home: function() {
      this.homeView = new HomeView();
      this.homeView.render();
    }
  });
  HeaderView = Backbone.View.extend({
    el: "#header",
    templateFileName: "header.html",
    template: headerTpl,
    initialize: function() {
    },
    render: function() {
      console.log(this.template)
      $(this.el).html(_.template(this.template));
    }
  });

  FooterView = Backbone.View.extend({
    el: "#footer",
    template: footerTpl,
    render: function() {
      this.$el.html(_.template(this.template));
    }
  })
  HomeView = Backbone.View.extend({
    el: "#content",
```

```
46      // шаблон: "home.html",
47      template: homeTpl,
48      initialize: function() {
49      },
50      render: function() {
51        $(this.el).html(_.template(this.template));
52      }
53    });
54    app = new ApplicationRouter();
55    Backbone.history.start();
56  });
```

Код, приведенный выше, показывает шаблоны. Все представления и маршрутизаторы внутри, требуя модули, чтобы убедиться, что шаблоны загруженны до того, как мы начинаем их обрабатывать.

Вот так выглядит **home.html**:

- Таблица сообщений
- Логика Underscore.js для вывода строк таблицы
- Форма нового сообщения

Давайте воспользуемся структурой библиотеки Twitter Bootstrap (с адаптивными компонентами) путем назначения классов `row-fluid` и `span12`:

```
<div class="row-fluid"  id="message-board">
<div class="span12">
  <table class="table table-bordered table-striped">
    <caption>Chat</caption>
    <thead>
      <tr>
        <th class="span2">Username</th>
        <th>Message</th>
      </tr>
    </thead>
    <tbody>
```

Эта часть содержит инструкции шаблона Underscore.js, который является просто некоторым JS-кодом, завернутый в метки `<%` и `%>`. `_.each()` — это функция итерации из библиотеки UnderscoreJS (underscorejs.org/#each[5]), которая перебирает элементы объекта/массива:

[5] http://underscorejs.org/#each

```
      <% if (models.length>0) {
        _.each(models, function (value,key, list) { %>
          <tr>
            <td><%= value.attributes.username %></td>
            <td><%= value.attributes.message %></td>
          </tr>
        <% });
      }
      else { %>
      <tr>
        <td colspan="2">No messages yet</td>
      </tr>
      <%}%>
    </tbody>
  </table>
</div>
</div>
```

Для формы нового сообщения, мы также используем класс `row-fluid`, а затем добавляем элементы `input`:

```
<div class="row-fluid"  id="new-message">
  <div class="span12">
    <form class="well form-inline">
      <input type="text"
        name="username"
        class="input-small"
        placeholder="Username" />
      <input type="text" name="message"
        class="input-small"
        placeholder="Message Text" />
      <a id="send" class="btn btn-primary">SEND</a>
    </form>
  </div>
</div>
```

Полный код файла шаблона **home.html**:

```
<div class="row-fluid"  id="message-board">
<div class="span12">
  <table class="table table-bordered table-striped">
    <caption>Chat</caption>
    <thead>
      <tr>
        <th class="span2">Username</th>
        <th>Message</th>
      </tr>
    </thead>
    <tbody>
      <% if (models.length>0) {
        _.each(models, function (value,key, list) { %>
          <tr>
            <td><%= value.attributes.username %></td>
            <td><%= value.attributes.message %></td>
          </tr>
        <% });
      }
      else { %>
      <tr>
        <td colspan="2">No messages yet</td>
      </tr>
      <%}%>
    </tbody>
  </table>
</div>
</div>
<div class="row-fluid"  id="new-message">
  <div class="span12">
    <form class="well form-inline">
      <input type="text"
        name="username"
        class="input-small"
        placeholder="Username" />
      <input type="text" name="message"
        class="input-small"
        placeholder="Message Text" />
      <a id="send" class="btn btn-primary">SEND</a>
    </form>
  </div>
</div>
```

Теперь мы можем добавить следующие компоненты:

- Коллекцию Parse.com,

- Модель Parse.com,
- События отправки/добавления сообщения,
- Функция получение/отображение сообщений.

Backbone-совместимая объектов/классов модели Parse.com JS SDK с обязательным атрибутом **className** (это название коллекции, которое появится во вкладке просмотра данных (Data Browser) веб-интерфейса Parse.com):

```
Message = Parse.Object.extend({
        className: "MessageBoard"
});
```

Backbone-совместимый объект коллекции Parse.com JavaScript SDK, который указывает на модель

```
MessageBoard = Parse.Collection.extend ({
        model: Message
});
```

Представление главной страницы должно иметь слушатель события click на кнопке “SEND”:

```
HomeView = Backbone.View.extend({
        el: "#content",
        template: homeTpl,
        events: {
                "click #send": "saveMessage"
        },
```

Давайте создавать коллекцию и привязывать слушатели событий, когда мы создаем homeView:

```
        initialize: function() {
                this.collection = new MessageBoard();
                this.collection.bind("all", this.render, this);
                this.collection.fetch();
                this.collection.on("add", function(message) {
                        message.save(null, {
                                success: function(message) {
                                        console.log('saved ' + message);
                                },
                                error: function(message) {
                                        console.log('error');
                                }
                        });
                        console.log('saved' + message);
                })
        },
```

Определение вызовов `saveMessage()` для события click кнопки “SEND”:

```
        saveMessage: function(){
                var newMessageForm = $("#new-message");
                var username = newMessageForm.
      find('[name="username"]').
      attr('value');
                var message = newMessageForm.
      find('[name="message"]').
      attr('value');
                this.collection.add({
                        "username": username,
                        "message": message
                        });
        },
        render: function() {
                console.log(this.collection);
                $(this.el).html(_.template(
      this.template,
      this.collection
    ));
        }
```

Конечный результат наших манипуляций в **app.js** может выглядеть примерно так:

```
/*
Rapid Prototyping with JS is a JavaScript
and Node.js book that will teach you how to build mobile
and web apps fast. — Read more at
http://rapidprototypingwithjs.com.
*/

require([
  'libs/text!header.html',
  'libs/text!home.html',
  'libs/text!footer.html'],
  function (
    headerTpl,
    homeTpl,
    footerTpl) {
  Parse.initialize(
    "your-parse-app-id",
    "your-parse-js-sdk-key");
  var ApplicationRouter = Backbone.Router.extend({
    routes: {
      "": "home",
      "*actions": "home"
    },
```

```
24      initialize: function() {
25        this.headerView = new HeaderView();
26        this.headerView.render();
27        this.footerView = new FooterView();
28        this.footerView.render();
29      },
30      home: function() {
31        this.homeView = new HomeView();
32        this.homeView.render();
33      }
34    });
35
36    HeaderView = Backbone.View.extend({
37      el: "#header",
38      templateFileName: "header.html",
39      template: headerTpl,
40      initialize: function() {
41      },
42      render: function() {
43        $(this.el).html(_.template(this.template));
44      }
45    });
46
47    FooterView = Backbone.View.extend({
48      el: "#footer",
49      template: footerTpl,
50      render: function() {
51        this.$el.html(_.template(this.template));
52      }
53    });
54    Message = Parse.Object.extend({
55      className: "MessageBoard"
56    });
57    MessageBoard = Parse.Collection.extend ({
58      model: Message
59    });
60
61    HomeView = Backbone.View.extend({
62      el: "#content",
63      template: homeTpl,
64      events: {
65        "click #send": "saveMessage"
66      },
67
68      initialize: function() {
```

```
69        this.collection = new MessageBoard();
70        this.collection.bind("all", this.render, this);
71        this.collection.fetch();
72        this.collection.on("add", function(message) {
73          message.save(null, {
74            success: function(message) {
75              console.log('saved '+message);
76            },
77            error: function(message) {
78              console.log('error');
79            }
80          });
81          console.log('saved'+message);
82        })
83      },
84      saveMessage: function(){
85        var newMessageForm=$("#new-message");
86        var username =
87          newMessageForm
88            .find('[name="username"]')
89            .attr('value');
90        var message = newMessageForm
91          .find('[name="message"]')
92          .attr('value');
93        this.collection.add({
94          "username": username,
95          "message": message
96          });
97      },
98      render: function() {
99        console.log(this.collection)
100       $(this.el).html(_.template(
101         this.template,
102         this.collection
103       ));
104     }
105   });
106
107   app = new ApplicationRouter();
108   Backbone.history.start();
109 });
```

Полный исходный код приложения Chat с Backbone.js и Parse.com доступно в rpjs/sdk[6].

[6] https://github.com/azat-co/rpjs/tree/master/sdk

5.2 Развертка Chat на PaaS

Как только вы поймете, что front-end приложение хорошо работает локально, с или без локального HTTP-сервера типа MAMP или XAMPP, выложите его на Windows Azure или Heroku. Подробные инструкции по развертке описаны в главе *jQuery и Parse.com*.

5.3 Усовершенствование Chat

В последних двух примерах, Chat имел базовую функциональность. Вы могли бы расширить возможности приложения, добавив дополнительные функции.

Дополнительные функции для разработчиков **среднего** уровня:

- Сортировка списка сообщений через атрибут *updateAt* перед их отображением
- Добавление кнопки “Refresh” для обновления списка сообщений
- Сохранение имя пользователя после первого сообщения во временную память или в сессию
- Добавление кнопки голосования “вверх”, расположенную рядом с каждым сообщением, и сохранение голосов
- Добавление кнопки голосования “вниз”, расположенную рядом с каждым сообщением, и сохранение голосов

Дополнительные функции для разработчиков **продвинутого** уровня:

- Добавление коллекции User
- Предотвратить голосование одним пользователем несколько раз
- Добавление возможность регистрации и логина, используя функции Parse.com
- Добавление кнопки “Delete Message” рядом с каждым сообщением, созданым пользователем
- Добавление кнопки “Edit Message” рядом с каждым сообщением, созданым пользователем

III Back-end прототипирование

6. Node.js и MongoDB

Резюме: приложение “Hello World” на Node.js, список некоторых из наиболее важных основных модулей Node.js, NPM workflow, подробные команды для развертывания приложения Node.js на Heroku и Windows Azure, MongoDB и его оболочка, приложения Chat в среде выполнения (run-time) и с базой данных, пример разработки через тестирование.

> *“Любой дурак может написать код, который понимает компьютер. Хорошие программисты пишут код, который могут понять люди.”* — Мартин Фаулер (Мартин Фаулер[1]

6.1 Node.js

6.1.1 Создание “Hello World” на Node.js

Чтобы проверить наличие Node.js на вашем компьютере, введите и выполните следующую команду в вашем терминале:

```
$ node -v
```

На момент написания этой статьи последняя версия 0.8.1. Если у вас нет установленного Node.js или у вас поздняя версия, вы всегда можете скачать последнюю версию nodejs.org/#download[2].

Как обычно, вы можете скопировать пример кода из rpjs/hello[3] или написать собственную программу с нуля. Если вы хотите сделать последнее, то создайте папку **hello** для вашего Node.js-приложения “Hello World”. Затем создайте файл **server.js** и строка за строкой введите код ниже.

Это позволит загрузить ядро модуля **http** для сервера (подробнее о модулях позже):

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

Нам понадобится номер порта для нашего Node.js server. Чтобы получить его из среды исполнения или назначить 1337, если среда не задана, используйте:

[1] https://ru.wikipedia.org/wiki/Фаулер,_Мартин
[2] http://nodejs.org/#download
[3] https://github.com/azat-co/rpjs/tree/master/hello

```
var port = process.env.PORT || 1337;
```

Это создаст сервер, функция обратного вызова будет содержать ответ кода обработчика:

```
var server = http.createServer(function (req, res) {
```

Чтобы задать правильный заголовок и код состояния, используйте:

```
  res.writeHead(200, {'Content-Type': 'text/plain'});
```

Для вывода “Hello World” с символом конца строки используйте:

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

Чтобы задать порт и показать адрес сервера и номер порта, используйте:

```
server.listen(port, function() {
  console.log('Server is running at %s:%s ',
    server.address().address, server.address().port);
});
```

Из папки, в которой у вас есть *server.js*, запустите в терминале следующую команду:

```
$ node server.js
```

Открыв localhost:1337[4] или 127.0.0.1:1337[5] или любой другой адрес, который вы увидите в терминале в результате работы функции `console.log()`, вы должны увидеть “Hello World” в браузере. Чтобы выключить сервер, нажмите Control + C.

Примечание

Имя основного файла может отличаться от server.js, например, index.js или app.js. В случае, если вам необходимо запустить файл **app.js**, просто используйте `$ node app.js`.

[4] http://localhost:1337/

[5] http://127.0.0.1:1337/

6.1.2 Node.js основные модули

В отличие от других технологий программирования, Node.js не поставляется со стандартной тяжелой библиотекой. Основные модули node.js являются необходимым минимумом, а остальное могут быть отобраны через реестр Node Package Manager (NPM). Основные модули, классы, методы и события включают в себя:

- http[6]
- util[7]
- querystring[8]
- url[9]
- fs[10]

http[11]

Это основной модуль, отвечающий за HTTP-сервер Node.js. Вот основные методы:

- `http.createServer()`: возвращает новый объект веб-сервера
- `http.listen()`: запускает приём соединений на указанный порт и имя хоста
- `http.createClient()`: приложение node может быть клиентом и делать запросы на другие серверы
- `http.ServerRequest()`: входящие запросы передаются в обработчики запросов
  - **data**: устанавливается, когда получен кусок тела сообщения
  - **end**: устанавливается один раз для каждого запроса
  - `request.method()`: метод запроса в виде строки
  - `request.url()`: строка URL запроса
- `http.ServerResponse()`: этот объект создается внутри HTTP-сервера, а не пользователем и используется в качестве выходного параметра обработчиков запросов
  - `response.writeHead()`: отправляет заголовок ответа на запрос
  - `response.write()`: отправляет тело ответа * `response.end()`: отправляет и заканчивает тело ответа

util[12]

Этот модуль предоставляет утилиты для отладки. Некоторые методы включают в себя:

- `util.inspect()`: возвращает строковое представление объекта, что полезно для отладки

[6] http://nodejs.org/api/http.html#http_http
[7] http://nodejs.org/api/util.html
[8] http://nodejs.org/api/querystring.html
[9] http://nodejs.org/api/url.html
[10] http://nodejs.org/api/fs.html
[11] **http://nodejs.org/api/http.html#http_http**
[12] **http://nodejs.org/api/util.html**

querystring[13]

Этот модуль предоставляет утилиты для работы со строками запроса. Некоторые методы включают в себя:

- `querystring.stringify()`: сериализует объект в строку запроса
- `querystring.parse()`: десериализует строку запроса в объект

url[14]

Этот модуль имеет утилиты для разбора и парсинга URL. Некоторые методы включают в себя:

- `parse()`: берет строку URL и возвращает объект

fs[15]

fs управляет операциями с файловой системой, такими как чтение и запись в/из файлов. Существуют синхронные и асинхронные методы в библиотеке. Некоторые методы включают в себя:

- `fs.readFile()`: читает файла асинхронно
- `fs.writeFile()`: записывает данные в файл асинхронно

Нет необходимости устанавливать или загружать модули ядра. Все что вам нужно, чтобы включить их в приложение — это следовать следующему синтаксис:

```
1 var http = require('http');
```

Список неосновных модулей могут быть найдены в:

- npmjs.org[16]: реестр Node Package Manager
- GitHub hosted list[17]: список модулей Node.js, поддерживаемые Joyent
- nodetoolbox.com[18]: реестр на основе статистики
- Nipster[19]: поисковой инструмент NPM для Node.js
- Node Tracking[20]: реестр на основе статистики GitHub

Если вы хотите узнать, как создавать собственные модули, взгляните на статью Your first Node.js module[21].

[13] **http://nodejs.org/api/querystring.html**
[14] **http://nodejs.org/api/url.html**
[15] **http://nodejs.org/api/fs.html**
[16] https://npmjs.org
[17] https://github.com/joyent/node/wiki/Modules
[18] http://nodetoolbox.com/
[19] http://eirikb.github.com/nipster/
[20] http://nodejsmodules.org
[21] http://cnnr.me/blog/2012/05/27/your-first-node-dot-js-module/

6.1.3 Node Package Manager

Node Package Manager, или NPM, управляет зависимостями и устанавливает модули для вас. Node.js поставляется с NPM, их сайт npmjs.org[22].

package.json содержит метаинформацию о нашем Node.js-приложениях, такую как номер версии, имя автора и, самое главное, зависимости, которые мы используем в приложении. Вся эта информация находится в JSON-объекте, которую читает NPM.

Если вы хотите установить пакеты и зависимости, указанные в **package.json**, введите:

```
$ npm install
```

Типичный **package.json** файл может выглядеть так:

```
{
   "name": "Blerg",
   "description": "Blerg blerg blerg.",
   "version": "0.0.1",
   "author": {
      "name" : "John Doe",
      "email" : "john.doe@gmail.com"
   },
   "repository": {
       "type": "git",
       "url": "http://github.com/johndoe/blerg.git"
   },
   "engines": [
       "node >= 0.6.2"
   ],
   "license" : "MIT",
   "dependencies": {
       "express": ">= 2.5.6",
       "mustache": "0.4.0",
       "commander": "0.5.2"
   },
   "bin" : {
       "blerg" : "./cli.js"
   }
}
```

Чтобы обновить пакет до его текущей последней версии или до последней версии, которая допустима по спецификации версий, определенной в **package.json**, используйте:

[22] http://npmjs.org/

```
$ npm update name-of-the-package
```

Или для одиночной установки модуля:

```
$ npm install name-of-the-package
```

Единственный модуль, используемый в примерах и который не принадлежит основному пакету Node.js — это **mongodb**. Мы установим его позже.

Heroku понадобится *package.json* чтобы запустить NPM на сервере.

Для получения дополнительных сведений о НПМ, взгляните на статью Tour of NPM[23].

6.1.4 Развертка “Hello World” на PaaS

Для развертки на Heroku и Windows Azure нам понадобится Git-репозиторий. Чтобы создать его из корневой директории проекта, введите следующую команду в вашем терминале:

```
$ git init
```

Git создаст скрытую папку **.git**. Теперь мы можем добавить файлы и сделать первый коммит:

```
$ git add .
$ git commit -am "first commit"
```

Совет

Для просмотра скрытых файлов в приложении Mac OS X Finder выполните эту команду в окне терминала: `defaults write com.apple.finder AppleShowAllFiles -bool true`. Чтбоы поменять флаг обратно на скрытый: `defaults write com.apple.finder AppleShowAllFiles -bool false`.

6.1.5 Развертка на Windows Azure

Для того, чтобы развернуть наше приложение “Hello World” на Windows Azure, мы должны добавить **удаленный репозиторий** Git. Вы можете скопировать URL с портала Windows Azure в разделе Web Site и использовать его с помощью этой команды:

```
$ git remote add azure yourURL
```

Теперь мы должны быть в состоянии пушить этой команды:

[23] http://tobyho.com/2012/02/09/tour-of-npm/

```
$ git push azure master
```

Если все прошло нормально, то вы должны увидеть в терминале логи об успехе и “Hello World” в браузере по URL из Windows Azure Web Site.

Для выгрузки изменений просто выполните:

```
$ git add .
$ git commit -m "changing to hello azure"
$ git push azure master
```

Более подробное руководство можно найти в туториале Build and deploy a Node.js web site to Windows Azure[24].

6.1.6 Развертка на Heroku

Для развертки на Heroku мы должны создать 2 дополнительных файла: **Procfile** и **package.json**. Вы может найти исходный код в папке rpjs/hello[25] или написать свой собственный.

Структура приложения “Hello World” выглядит так:

```
/hello
  -package.json
  -Procfile
  -server.js
```

Procfile является механизмом для декларирования команд, которые выполняются с помощью dyno вашего приложения на платформе Heroku. По сути, он рассказывает Heroku, какие процессы запускать. Procfile имеет только одну строку в этом случае:

```
web: node server.js
```

Для этого примера мы оставим **package.json** простым:

```
{
  "name": "node-example",
  "version": "0.0.1",
  "dependencies": {
  },
  "engines": {
    "node": ">=0.6.x"
  }
}
```

Как только у нас появились все файлы в папке проекта, мы можем использовать Git для развертывания приложения. Команды практически одинаковы, что и в Windows Azure, за исключением того, что мы должны добавить удаленный репозиторий Git и создать стек Cedar командой:

[24] http://www.windowsazure.com/en-us/develop/nodejs/tutorials/create-a-website-(mac)/

[25] https://github.com/azat-co/rpjs/tree/master/hello

```
$ heroku create
```

После этого мы пушим и обновляем:

```
$ git push heroku master
$ git add .
$ git commit -am "changes"
$ git push heroku master
```

Если все прошло нормально, то вы должны увидеть в терминале логи об успехе и “Hello World” в браузере по URL вашего Heroku-приложения.

6.2 Chat: run-time версия

Первая версия чата back-end приложение Chat будет хранить сообщения только в памяти среды выполнения (run-time memory) KISS[26]. Это означает, что каждый раз при запуске/перезагрузки сервера, данные будут потеряны.

Сначала мы начнем с простого тест-кейса для иллюстрации подхода Test-Driven Development. Полный код доступен в папке rpjs/test[27].

6.3 Тесты для Chat

У нас должно быть два метода:

1. Получить все сообщения в ввиде массива из объектов JSON для конечной точки GET `/message` используя метод `getMessages()`
2. Добавить новое сообщение со свойствами **name** и **message** для маршрута POST `/messages` через функцию `addMessage()`

Мы начнем с создания пустого файла **mb-server.js**. После этого давайте перейдем к тестам и создадим файл **test.js** со следующим содержимым:

[26] https://ru.wikipedia.org/wiki/KISS_(%D0%BF%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF)

[27] https://github.com/azat-co/rpjs/tree/master/test

```
var http = require('http');
var assert = require('assert');
var querystring = require('querystring');
var util = require('util');

var messageBoard = require('./mb-server');

assert.deepEqual('[{"name":"John","message":"hi"}]',
  messageBoard.getMessages());
assert.deepEqual ('{"name":"Jake","message":"gogo"}',
  messageBoard.addMessage ("name=Jake&message=gogo"));
assert.deepEqual('[{"name":"John","message":"hi"},{"name":"Jake","message":"gogo"}]',
  messageBoard.getMessages("name=Jake&message=gogo"));
```

Пожалуйста, имейте в виду, что это очень упрощенное сравнение строк, а не JavaScript-объектов. Поэтому каждый пробел, кавычка и заглавные буквы имееют значение. Вы могли бы сделать сравнение “умнее”, переведя строку в JSON-объект:

```
JSON.parse(str);
```

Для тестирования наших предположений, мы используем модуль ядра Node.js assert[28]. Он предоставляет ряд полезных методов, таких как `equal()`, `deepEqual()` и др.

Более продвинутые библиотеки включают в себя альтернативные интерфейсы с TDD и/или BDD-подходами:

- Should[29]
- Expect[30]

Так же вы можете использовать следующие библиотеки и модули:

- Mocha[31]
- NodeUnit[32]
- Jasmine[33]
- Vows[34]
- Chai[35]

Вы можете сейчас скопировать скрипт “Hello World” в файл **mb-server.js** или даже держать его пустым. Если мы запустим **test.js** командой терминала:

[28] http://nodejs.org/api/assert.html
[29] https://github.com/visionmedia/should.js/
[30] https://github.com/LearnBoost/expect.js/
[31] http://visionmedia.github.com/mocha/
[32] https://github.com/caolan/nodeunit
[33] http://pivotal.github.com/jasmine/
[34] http://vowsjs.org/
[35] http://chaijs.com/

```
$ node test.js
```

мы должны увидеть сообщение об ошибке. Возможно, похожее на эту:

```
TypeError: Object #<Object> has no method 'getMessages'
```

Это совершенно нормально, потому что мы еще не написали `getMessages()`. Так давайте сделаем это и сделаем наше приложение более полезным, добавив два новых метода: получить список сообщений для чата и добавить новое сообщение в коллекцию.

Файл **mb-server.js** с глобальным объектом `exports`:

```
exports.getMessages = function() {
  return JSON.stringify(messages);
};
exports.addMessage = function (data){
  messages.push(querystring.parse(data));
  return JSON.stringify(querystring.parse(data));
};
```

Пока ничего сложного, верно? Для хранения списка сообщений мы будем использовать массив:

```
var messages=[];
//этот массив будет содержать наши сообщения
messages.push({
  "name":"John",
  "message":"hi"
});
//образец сообщение для проверки метода list
```

Совет

Как правило, оснастка, типа болванки данных, относится к test/spec-файлам, а не к основной кодовой базе приложения.

Наш серверный код будет выглядеть намного интереснее. Для получения списка сообщений, в соответсвии с методологией REST, мы должны сделать GET-запрос. Для создание/добавления нового сообщения должен быть POST-запрос. Поэтому в нашем объекте createServer, мы должны добавить `req.method()` и `req.url()` для проверки типа HTTP-запроса и пути URL-адреса.

Давайте загрузим модуль http:

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

Нам понадобится несколько удобных функций из модулей util и querysting (для сериализации и десериализации объектов и строк запросов):

```
var util = require('util');
var querystring = require('querystring');
```

Чтобы создать сервер и сделать его доступным другим модулям (например test.js):

```
exports.server=http.createServer(function (req, res) {
```

Внутри обратного вызова обработчика запроса мы должны проверить, является ли метод запроса POST-ом и равен ли URL `messages/create.json`:

```
  if (req.method == "POST" &&
    req.url == "/messages/create.json") {
```

Если условие выше истинно, то добавляем сообщение в массив. Однако, **data** должно быть преобразовано в строковый тип (с Кодировкой UTF-8) до добавления, потому что это тип Buffer:

```
    var message = '';
    req.on('data', function(data, message){
      console.log(data.toString('utf-8'));
      message = exports.addMessage(data.toString('utf-8'));
```

Эти логи помогут нам отслеживать в терминале активность сервера:

```
    })
    console.log(util.inspect(message, true, null));
    console.log(util.inspect(messages, true, null));
```

Выходные данные должны быть в текстовом формате со статусом 200 (ок):

```
    res.writeHead(200, {
      'Content-Type': 'text/plain'
     });
```

Мы выводим сообщение с ID вновь созданного объекта:

```
    res.end(message);
  }
```

Если метод равен GET, а URL равен `/messages/list.json`, выводим список сообщений:

```
  if (req.method == "GET" &&
    req.url == "/messages/list.json") {
```

Получение списка сообщений:

```
    var body = exports.getMessages();
```

Тело ответа будет содержать результат:

```
    res.writeHead(200, {
      'Content-Length': body.length,
      'Content-Type': 'text/plain'
    });
    res.end(body);
  }
  else {
```

Это задаст нужный заголовок и код состояния:

```
    res.writeHead(200, {
      'Content-Type': 'text/plain'
    });
```

В случае, если это ни один из двух описаных выше конечных точек, мы выводим строку с символом конца строки:

```
    res.end('Hello World\n');
  };
  console.log(req.method);


}).listen(1337, "127.0.0.1");
```

Теперь мы должны задать IP-адрес и порт сервера:

```
console.log('Server running at http://127.0.0.1:1337/');
```

Мы помещаем методы для модульного тестирования в **test.js** (ключевое слово exports). Эта функция возвращает массив сообщений в виде строки/текста:

```
exports.getMessages = function() {
  return JSON.stringify(messages);
};
```

`addMessage()` преобразует строку в объект JavaScript методом parse/deserializer из querystring:

```
exports.addMessage = function (data){
  messages.push(querystring.parse(data));
```

Кроме того, возвращает новое сообщение как JSON в виде строки:

```
  return JSON.stringify(querystring.parse(data));
};
```

Вот полный код **mb-server.js**. Он также доступен в папке rpjs/test[36]:

```
var http = require('http');
//загружает http-модуль
var util = require('util');
//полезные функции
var querystring = require('querystring');
//загружает модуль querystring,
//нужен для сериализации и
//десериализации объектов и строк запроса

var messages=[];
//этот массив будет содержать наши сообщения
messages.push({
  "name": "John",
  "message": "hi"
});
//образец сообщения для проверки метода list

exports.server=http.createServer(function (req, res) {
//создает сервер
  if (req.method == "POST" &&
    req.url == "/messages/create.json") {
    //если метод равен POST и
    //URL равен messages/, то добавляем сообщение в массив
    var message = '';
    req.on('data', function(data, message){
      console.log(data.toString('utf-8'));
      message = exports.addMessage(data.toString('utf-8'));
      //данные типа Buffer
      //должны быть конвертированы в строку
      //с кодированием в UTF-8 спервa
      //добавляет сообщение в массив
    })
    console.log(util.inspect(message, true, null));
    console.log(util.inspect(messages, true, null));
```

[36] https://github.com/azat-co/rpjs/tree/master/test

```
35          //вывод отладки в терминал
36      res.writeHead(200, {
37        'Content-Type': 'text/plain'
38       });
39      //устанавливает правильный заголовок и код состояния
40      res.end(message);
41      //выводим сообщение, следует добавить id объекта
42    }
43    if (req.method == "GET" &&
44      req.url == "/messages/list.json") {
45    //если метод GET и
46    //URL /messages выводит список сообщений
47      var body = exports.getMessages();
48      //body будет содержать наш вывод
49      res.writeHead(200, {
50        'Content-Length': body.length,
51        'Content-Type': 'text/plain'
52      });
53      res.end(body);
54    }
55    else {
56      res.writeHead(200, {
57        'Content-Type': 'text/plain'
58      });
59      //устанавливает правильный заголовок и код состояния
60      res.end('Hello World\n');
61    };
62    console.log(req.method);
63    //выводит строку с символом конца строки
64  }).listen(1337, "127.0.0.1");
65  //устанавливает порт и IP-адрес сервера
66  console.log('Server running at http://127.0.0.1:1337/');
67
68  exports.getMessages = function() {
69    return JSON.stringify(messages);
70    //выходной массив сообщений в виде строки/текста
71  };
72  exports.addMessage = function (data){
73    messages.push(querystring.parse(data));
74    //для конвертации строки в
75    //объект JavaScript мы используем parse/deserializer
76    return JSON.stringify(querystring.parse(data));
77    //вывод нового сообщения в формате JSON в виде строки
78  };
```

Чтобы проверить это, перейдите в localhost:1337/messages/list.json[37]. Вы должны увидеть пример сообщения. Кроме того, можно использовать терминальную команду:

```
$ curl http://127.0.0.1:1337/messages/list.json
```

POST-запрос, используя интерфейс командной строки:

```
curl -d "name=BOB&message=test" http://127.0.0.1:1337/messages/create.json
```

Вы должны получить вывод в окно терминала сервера и новое сообщения “test” при обновлении localhost:1337/messages/list.json[38]. Разумеется, все три теста должны пройти.

Ваше приложение в последствии может сильно разрастись с большим количеством методов и URL-путей. Вот где могут пригодиться фреймворки. Они предоставляют помощников для обработки запросов и другие приятные вещи типа поддержки статических файлов, сессий и т.д. В этом примере мы намеренно не использовали никаких фреймворков типа Express (http://expressjs.com/) или Restify (http://mcavage.github.com/node-restify/). Другие Node.js фреймворки:

- Derby[39]: MVC-фреймворк, делающий легким написание real-time приложений, приложений для совместной работы, которые работают и на Node.js и в браузере
- Express.js[40]: наиболее надежный, проверенный и используемый Node.js-фреймворк
- Restify[41]: легковесный фреймворк для REST API серверов
- Sails.js[42]: MVC Node.js-фреймворк
- hapi[43]: Node.js-фреймворк, надстройка над Express.js
- Connect[44]: middleware-фреймворк для node, поставляющийся в комплекте с более чем 18 middleware (плагинами) и богатым выбором сторонних middleware
- GeddyJS[45]: простой, структурированный MVC-фреймворк для Node
- CompoundJS[46] (ex-RailswayJS): Node.JS MVC-фреймворк, основанный на ExpressJS
- Tower.js[47]: фреймворк для Node.js и для браузера
- Meteor[48]: open-source платформа для создания высококачественных веб-приложений в короткое время

Пути совершенствования приложения:

[37] http://localhost:1337/messages/list.json
[38] http://localhost:1337/messages/list.json
[39] http://derbyjs.com/
[40] http://expressjs.com
[41] http://mcavage.github.com/node-restify/
[42] http://sailsjs.org/
[43] http://spumko.github.io/
[44] http://www.senchalabs.org/connect/
[45] http://geddyjs.org
[46] http://compoundjs.com/
[47] http://towerjs.org
[48] http://meteor.com

- Улучшить существующие тестовые случае с помощью добавления сравнения объектов вместо строк
- Переместить тестовые данные из **mb-server.js** в **test.js**
- Добавить тестовые случаи для поддержки вашего front-end, например, голосования или логина пользователя
- Добавить методы для голосования, логина пользователя
- Генерировать уникальные id для каждого сообщения и хранить их в хэше вместо массива
- Установить Mocha и отрефакторить test.js под эту библиотеку

До сих пор мы пользовались хранением сообщений в памяти приложения (run-time memory), так что каждый раз, после перезапуска приложения, мы их теряем. Чтобы исправить это, мы должны добавить сохранение и одним из способом является использование базы данных, такой как MongoDB.

6.4 MongoDB

6.4.1 Оболочка MongoDB

Если вы еще не сделали этого, пожалуйста, установите последнюю версию MongoDB из mongodb.org/downloads[49]. Для получения более подробных инструкций, пожалуйста, обратитесь к разделу *Установка, база дынных: MongoDB*.

Теперь из папки, куда вы распаковали архив, запустите службу **mongod**:

```
$ ./bin/mongod
```

Вы должны быть в состоянии видеть информацию в вашем терминале и в браузере по адресу localhost:28017[50].

Для доступа к оболочке MongoDB (или **mongo**) запустите в новом окне терминала (**важно!**) в той же папке эту команду:

```
$ ./bin/mongo
```

Вы должны увидеть нечто вроде этого, в зависимости от вашей версии оболочки MongoDB:

```
MongoDB shell version: 2.0.6
connecting to: test
```

Для проверки базы данных используйте JavaScript-подобный интерфейс и команды **save** и **find**:

[49] http://www.mongodb.org/downloads
[50] http://localhost:28017

```
> db.test.save( { a: 1 } )
> db.test.find()
```

Более подробные пошаговые инструкции доступны в *Установка, база дынных: MongoDB*.
Некоторые другие полезные команды оболочки MongoDB:

```
> help
> show dbs
> use board
> show collections
> db.messages.remove();
> var a = db.messages.findOne();
> printjson(a);
> a.message = "hi";
> db.messages.save(a);
> db.messages.find({});
> db.messages.update({name: "John"},{$set: {message: "bye"}});
> db.messages.find({name: "John"});
> db.messages.remove({name: "John"});
```

Полный обзор интерактивной оболочки MongoDB доступен на mongodb.org: Overview — The MongoDB Interactive Shell[51].

6.4.2 MongoDB Native Driver

Мы будем использовать Node.js Native Driver для MongoDB (https://github.com/christkv/node-mongodb-native) для доступа к MongoDB из Node.js-приложений. Полная документация также доступна по адресу http://mongodb.github.com/node-mongodb-native/api-generated/db.html.

Для установки Node.js Native Driver используйте:

```
$ npm install mongodb
```

Более подробная информация по адресу http://www.mongodb.org/display/DOCS/node.JS.
Не забудьте включить зависимость в файл **package.json**:

[51] http://www.mongodb.org/display/DOCS/Overview+-+The+MongoDB+Interactive+Shell

```
{
  "name": "node-example",
  "version": "0.0.1",
  "dependencies": {
        "mongodb":"",
        ...
  },
  "engines": {
    "node": ">=0.6.x"
  }
}
```

Кроме того, для собственного развития можно использовать и другие мапперы, которые доступны как расширение Native Driver:

- Mongoskin[52]: the future layer for node-mongodb-native
- Mongoose[53]: асинхронный JavaScript-драйвер с дополнительной поддержкой моделирования
- Mongolia[54]: легковесный MongoDB ORM/драйвер
- Monk[55]: Monk — это небольшая прослойка, которая обеспечивает простое, но существенное улучшения юзабилити использования MongoDB в пределах Node.js

Этот небольшой пример покажет, можем ли мы подключиться к локальной MongoDB из скрипта Node.js.

После того, как мы установили библиотеку, мы можем включить библиотеку **mongodb** в наш ƒайла **db.js**:

```
var util = require('util');
var mongodb = require ('mongodb');
```

Это один из способов подключения к серверу MongoDB. Переменная db будет хранить ссылку на базу данных с указанным хостом и портом:

[52] https://github.com/guileen/node-mongoskin
[53] http://mongoosejs.com/
[54] https://github.com/masylum/mongolia
[55] https://github.com/LearnBoost/monk

```
var Db = mongodb.Db;
var Connection = mongodb.Connection;
var Server = mongodb.Server;
var host = '127.0.0.1';
var port = 27017;

var db=new Db ('test', new Server(host,port, {}));
```

Чтобы открыть соединение:

```
db.open(function(e,c){
        //сделать что-нибудь с базой данных здесь
  // console.log (util.inspect(db));
  console.log(db._state);
  db.close();
});
```

Этот фрагмент кода доступен в папке rpjs/db/db.js[56]. Если мы запустим его, он должен вывести в терминале “**connected**”. Если у вас есть сомнения и требуется проверка свойств объекта, есть полезный метод в модуле **util**:

```
console.log(util.inspect(db));
```

6.4.3 MongoDB на Heroku: MongoHQ

После того, как вы сделали приложение, которое локально показывает ‘connected’, пришло время немного изменить его и развернуть на платформе-как-услуге, т.е., Heroku.

Мы рекомендуем использовать MongoHQ add-on[57], который является частью технологии MongoHQ[58]. Он предоставляет веб-интерфейс для просмотра и управления данными и коллекциями. Более подробная информация доступна на https://devcenter.heroku.com/articles/mongohq.

Примечание

Возможно вам придется предоставить данные вашей кредитной карты, чтобы использовать MongoHQ, даже если вы выбрали бесплатную версию. Денег снять не должны.

Для того, чтобы подключиться к серверу базы данных, существует URL подключения базы данных (так же известный, как MongoHQ URL/URI), который представляет собой способ передачи всей необходимой информации для подключения к базе данных в одну строку.

Строка подключения к базе данных MONGOHQ_URL имеет следующий Формат:

[56] https://github.com/azat-co/rpjs/blob/master/db/db.js

[57] https://addons.heroku.com/mongohq

[58] https://www.mongohq.com/home

```
mongodb://user:pass@server.mongohq.com/db_name
```

Вы можете либо скопировать строку MongoHQ URL с сайта Heroku (и захардкодить его) либо получить строку из объекта **process.env** Node.js:

```
process.env.MONGOHQ_URL;
```

или

```
var connectionUri = url.parse(process.env.MONGOHQ_URL);
```

Совет

Глобальный объект процесса дает доступ к переменным среды через process.env. Эти переменные традиционно используются для передачи имен хостов и портов базы данных, паролей, API-ключей, номеров портов и других сведений о системе, которые не должны быть захардкоженны в основную логику.

Чтобы заставить наш код работать как локально, так и на Heroku, мы можем использовать логическую операцию ИЛИ `||` и назначить локальный хост и порт, если переменные среды равны undefined:

```
var port = process.env.PORT || 1337;
var dbConnUrl = process.env.MONGOHQ_URL ||
  'mongodb://@127.0.0.1:27017';
```

Вот обновленный кроссредный файл **db.js**:

```
var url = require('url')
var util = require('util');
var mongodb = require ('mongodb');
var Db = mongodb.Db;
var Connection = mongodb.Connection;
var Server = mongodb.Server;

var dbConnUrl = process.env.MONGOHQ_URL ||
'mongodb://127.0.0.1:27017';
var host = url.parse(dbConnUrl).hostname;
var port = new Number(url.parse(dbConnUrl).port);

var db=new Db ('test', new Server(host,port, {}));
db.open(function(e,c){
  // console.log (util.inspect(db));
  console.log(db._state);
  db.close();
});
```

После добавления **MONGOHQ_URL** в **db.js** мы можем инициализировать Git-репозиторий, создать Heroku-приложение, добавить MongoHQ add-on к нему и развернуть приложение с помощью Git.

Используйте те же шаги, что и в предыдущих примерах, чтобы создать новый git-репозиторий:

```
git init
git add .
git commit -am 'initial commit'
```

Создайте стек Cedar Heroku-приложения:

```
$ heroku create
```

Если все прошло хорошо, вы должны увидеть сообщение, с именем нового Heroku-приложения (и URL) вместе с сообщением, что удаленный репозиторий был добавлен. Наличие удаленного репозитория в вашем локальном git важно. Вы можете проверить список удаленных репозиториев командой:

```
git remote show
```

Чтобы установить бесплатно MongoHQ на существующем Heroku-приложении (работает по принципу ‘один add-on — одно приложение’), используйте:

```
$ heroku addons:add mongohq:sandbox
```

Или войдите в addons.heroku.com/mongohq[59] с учетными данными Heroku и выберите MongoHQ Free для этого конкретного Heroku-приложения, если вам известно имя этого приложения.

Если **db.js** и модифицированный **db.js** у вас работают, давайте расширим его путем добавления HTTP-сервера, так чтобы сообщение ‘connected’ отображалось в браузере, а не в окне терминала. Чтобы сделать это, мы обернем создание экземпляра объекта сервера в коллбэк соединения с базой данных:

[59] https://addons.heroku.com/mongohq

```
...
db.open(function(e,c){
        // console.log (util.inspect(db));
        var server = http.createServer(function (req, res) {
    //создает сервер
          res.writeHead(200, {'Content-Type': 'text/plain'});
    //устанавливает правильные заголовки и коды статусов
          res.end(db._state);
    //выводит строку с символом конца строки
        });
        server.listen(port, function() {
                console.log('Server is running at %s:%s '
    , server.address().address
    , server.address().port);
    //устанавливает порта и IP-адрес сервера
        });
        db.close();
});
...
```

Окончательный готовый для развертки файл **app.js** из rpjs/db[60]:

```
/*
Rapid Prototyping with JS is a JavaScript
and Node.js book that will teach you how to build mobile
and web apps fast. — Read more at
http://rapidprototypingwithjs.com.
*/
var util = require('util');
var url = require('url');
var http = require('http');
var mongodb = require('mongodb');
var Db = mongodb.Db;
var Connection = mongodb.Connection;
var Server = mongodb.Server;
var port = process.env.PORT || 1337;
var dbConnUrl = process.env.MONGOHQ_URL ||
  'mongodb://@127.0.0.1:27017';
var dbHost = url.parse(dbConnUrl).hostname;
var dbPort = new Number(url.parse(dbConnUrl).port);
console.log(dbHost + dbPort)
var db = new Db('test', new Server(dbHost, dbPort, {}));
db.open(function(e, c) {
  // console.log (util.inspect(db));
```

[60] https://github.com/azat-co/rpjs/blob/master/db

```
  // creates server
  var server = http.createServer(function(req, res) {
    //sets the right header and status code
    res.writeHead(200, {
      'Content-Type': 'text/plain'
    });
    //outputs string with line end symbol
    res.end(db._state);
  });
  //sets port and IP address of the server
  server.listen(port, function() {
    console.log(
      'Server is running at %s:%s ',
    server.address().address,
    server.address().port);
  });
  db.close();
});
```

После развертывания, вы должны быть в состоянии открыть предоставленный Heroku URL и увидеть сообщение ‘connected’.

Вот инструкции, как использовать MongoDB из кода Node.js: mongodb.github.com/node-mongodb-native/api-articles/nodekoarticle1.html[61].

Другой подход заключается в использовании модуля MongoHQ, который доступен по адресу github.com/MongoHQ/mongohq-nodejs[62].

Этот пример иллюстрирует другое использование **mongodb**-библиотеки через вывод коллекций и количества документов. Полный исходный код rpjs/db/collections.js[63]:

```
var mongodb = require('mongodb');
var url = require('url');
var log = console.log;
var dbUri = process.env.MONGOHQ_URL || 'mongodb://localhost:27017/test';

var connectionUri = url.parse(dbUri);
var dbName = connectionUri.pathname.replace(/^\//, '');

mongodb.Db.connect(dbUri, function(error, client) {
  if (error) throw error;

  client.collectionNames(function(error, names){
    if(error) throw error;

```

[61] http://mongodb.github.com/node-mongodb-native/api-articles/nodekoarticle1.html
[62] https://github.com/MongoHQ/mongohq-nodejs
[63] https://github.com/azat-co/rpjs/blob/master/db/collections.js

```
    //вывод всех имен коллекций
    log("Collections");
    log("===========");
    var lastCollection = null;
    names.forEach(function(colData){
      var colName = colData.name.replace(dbName + ".", '')
      log(colName);
      lastCollection = colName;
    });
    if (!lastCollection) return;
    var collection = new mongodb.Collection(client, lastCollection);
    log("\nDocuments in " + lastCollection);
    var documents = collection.find({}, {limit:5});

    //вывод количества всех найденных документах
    documents.count(function(error, count){
      log("   " + count + " documents(s) found");
      log("====================");

      // вывод первых 5 документов
      documents.toArray(function(error, docs) {
        if(error) throw error;

        docs.forEach(function(doc){
          log(doc);
        });

        // закрываем соединение
        client.close();
      });
    });
  });
});
```

Мы использовали сокращение для `console.log()` в `var log = console.log;` и `return` для управления потоком через `if (!lastCollection) return;`.

6.4.4 BSON

Бинарный JSON, или BSON — это специальный тип данных, который использует MongoDB. Это JSON, который имеет поддержку дополнительных, более сложных типов данных.

Предупреждение

Слово предостережения о BSON: ObjectId в MongoDB эквивалентна ObjectID в MongoDB Node.js Native Driver, т.е., убедитесь, что используется правильный регистр. В противном случае вы получите ошибку. Больше о типах: ObjectId in MongoDB[64] vs Data Types in MongoDB Native Node.js Driver[65]. Пример кода Node.js с `mongodb.ObjectID()`: `collection.findOne({_id: new ObjectID(idString)}, console.log) // ok`. С другой стороны, в оболочке MongoDB мы используем: `db.messages.findOne({_id:ObjectId(idStr)});`.

6.5 Chat: MongoDB-версия

Мы должны были все настроить для написания Node.js-приложения, которое будет работать как локально, так и на Heroku. Исходный код доступен в rpjs/mongo[66]. Структура приложения проста:

```
/mongo
  -web.js
  -Procfile
  -package.json
```

Это то, как выглядит *web.js*. Первым делом мы включаем наши библиотеки:

```
var http = require('http');
var util = require('util');
var querystring = require('querystring');
var mongo = require('mongodb');
```

Затем вставляем волшебную строку для подключения к MongoDB:

```
var host = process.env.MONGOHQ_URL || "mongodb://@127.0.0.1:27017/twitter-clone";
//MONGOHQ_URL=mongodb://user:pass@server.mongohq.com/db_name
```

Примечание

Формат URI/URL содержит необязательное имя базы данных, в которой наша коллекция будет храниться. Не стесняйтесь менять его на что-то другое, например ‘rpjs’ или ‘test’.

Мы вкладываем всю логику внутрь открытого соединения в виде функции обратного вызова:

[64] http://www.mongodb.org/display/DOCS/Object+IDs
[65] https://github.com/mongodb/node-mongodb-native/#data-types
[66] https://github.com/azat-co/rpjs/tree/master/mongo

```
mongo.Db.connect(host, function(error, client) {
  if (error) throw error;
  var collection = new mongo.Collection(
    client,
    'messages');
  var app = http.createServer(function(request, response) {
    if (request.method === "GET" &&
      request.url === "/messages/list.json") {
      collection.
        find().
        toArray(function(error,results) {
        response.writeHead(200,{
          'Content-Type':'text/plain'
        });
        console.dir(results);
        response.end(JSON.stringify(results));
      });
    };
    if (request.method === "POST" &&
      request.url === "/messages/create.json") {
      request.on('data', function(data) {
        collection.insert(
          querystring.parse(data.toString('utf-8')),
          {safe:true},
          function(error, obj) {
            if (error) throw error;
            response.end(JSON.stringify(obj));
          }
        )
      })
    };
  });
  var port = process.env.PORT || 5000;
  app.listen(port);
})
```

Примечание

Мы не обязаны использовать дополнительные слова после имени коллекции/сущности, например, вместо `/messages/list.json` и `/messages/create.json` прекрасно иметь просто `/messages` для всех HTTP-методов типа GET, POST, PUT, DELETE. Если вы их измените в коде приложения, убедитесь в использовании обновленных команд CURL и front-end кода.

Для тестирования с помощью CURL запустите:

```
curl http://localhost:5000/messages/list.json
```

Или откройте в своем браузере http://locahost:5000/messages/list.json.

Он должен выдать вам пустой массив: `[]`. Теперь опубликуйте новое сообщение:

```
curl  -d "username=BOB&message=test" http://localhost:5000/messages/create.json
```

Теперь мы должны увидеть ответ, содержащий ObjectID вновь созданного элемента, например: `[{"username":"BOB","message":"test","_id":"51edcad45862430000000001"}]`. Ваш ObjectId может отличаться.

Если локально все работает как надо, попробуйте развернуть его на Heroku.

Для тестирования приложения на Heroku, можно использовать те же команды CURL[67], заменив `http://localhost/` или "http://127.0.0.1" вашим уникальным хостом/URL Heroku-приложения:

```
$ curl http://your-app-name.herokuapp.com/messages/list.json
$ curl -d "username=BOB&message=test"
  http://your-app-name.herokuapp.com/messages/create.json
```

Также было бы хорошо перепроверить базу данных либо через оболочку Mongo: команда `$ mongo`, затем `use twitter-clone` и `db.messages.find()`, либо через MongoHub[68], mongoui[69], mongo-express[70] или в случае с MongoHQ через веб-интерфейс, доступный на heroku.com веб-сайте.

Если вы хотите использовать другое имя домена вместо http://your-app-name.herokuapp.com, вам потребуется сделать две вещи:

1. Скажите Heroku имя вашего домена:

```
$ heroku domains:add www.your-domain-name.com
```

2. Добавьте в DNS manager запись CNAME http://your-app-name.herokuapp.com.

Больше информации о пользовательских доменах можно найти тут devcenter.heroku.com/articles/custom domains[71].

Совет

Для более продуктивной и эффективной разработки мы должны автоматизировать как можно больше, например, использовать тесты, вместо CURL команды. Есть статья о библиотеке Mocha в БОНУСНОЙ главе, которая, наряду с библиотеками superagent или request экономит время для решения подобных задач.

[67] http://curl.haxx.se/docs/manpage.html
[68] https://github.com/bububa/MongoHub-Mac
[69] https://github.com/azat-co/mongoui
[70] https://github.com/andzdroid/mongo-express
[71] https://devcenter.heroku.com/articles/custom-domains

7. Собираем все вместе

Резюме: описания различных подходов развертывания, финальная версия приложения Chat и его развертывание.

> *“Отладка в 2 раза сложнее написания кода. Поэтому, если вы пишете код на столько искусно, на сколько это возможно, то вы по определению не достаточно умны, чтобы отладить его.”* — Брайан Уилсон Керниган[1]

Теперь было бы хорошо, если бы мы могли разместить наши front-end и back-end приложения вместе для совместной работы. Есть несколько способов сделать это:

- Различные домены (Heroku-приложения) для front-end и back-end приложений: убедитесь, что нет кроссдоменных проблем через использование CORS или JSONP. Этот подход подробно описан позже.
- Один и тот же домен развертывания: убедитесь, что Node.js обрабатывает статические ресурсы и активы для front-end приложения — не рекомендуется для серьезного боевого приложения.

7.1 Разные домены развертывания

Это пока наилучшее решение для боевой среды. Back-end приложения обычно развертываются на *http://app.* или *http://api.* поддоменах.

Один из способов заставить другой домен развертывания работать, это преодолеть ограничение технологии AJAX в один домен с помощью JSONP:

[1] https://ru.wikipedia.org/wiki/%D0%9A%D0%B5%D1%80%D0%BD%D0%B8%D0%B3%D0%B0%D0%BD,_%D0%91%D1%80%D0%B0%D0%B9%D0%B0%D0%BD

```
var request = $.ajax({
  url: url,
  dataType: "jsonp",
  data: {...},
  jsonpCallback: "fetchData",
  type: "GET"
});
```

Другой, и более качественный способ, это добавить метод OPTIONS и специальные заголовки, которые называются CORS, в серверное приложение Node.js перед выводом:

```
  ...
  response.writeHead(200,{
    'Access-Control-Allow-Origin': origin,
    'Content-Type':'text/plain',
    'Content-Length':body.length
  });
  ...
```

или

```
  ...
  res.writeHead(200, {
    'Access-Control-Allow-Origin', 'your-domain-name',
    ...
  });
  ...
```

Необходимость метода OPTIONS изложена в HTTP access control (CORS)[2]. Запрос OPTIONS может быть рассмотрен следующим образом:

```
  ...
  if (request.method=="OPTIONS") {
    response.writeHead("204", "No Content", {
      "Access-Control-Allow-Origin": origin,
      "Access-Control-Allow-Methods":
        "GET, POST, PUT, DELETE, OPTIONS",
      "Access-Control-Allow-Headers": "content-type, accept",
      "Access-Control-Max-Age": 10, // Секунды.
      "Content-Length": 0
    });
    response.end();
  };
  ...
```

[2] https://developer.mozilla.org/en-US/docs/HTTP_access_control

7.2 Изменение конечных точек

Наше front-end приложение использует Parse.com в качестве замены back-end приложения. Теперь мы можем переключиться на наш собственный back-end, заменив конечные точки (да, это безболезненно!). Исходный код front-end приложения находится в GitHub-папке rpjs/board[3].

В начале файла **app.js** раскомментируйте первую строку для запуска локально или замените значения URL-адресов на значения публичных URL-адресов back-end приложений Heroku или Windows Azure:

```
// var URL = "http://localhost:5000/";
var URL ="http://your-app-name.herokuapp.com/";
```

Как вы можете видеть, большая часть кода в **app.js** и структура папок осталась нетронутой, за исключением замены моделей и коллекций Parse.com на оригинальные из Backbone.js:

```
Message = Backbone.Model.extend({
  url: URL + "messages/create.json"
})
MessageBoard = Backbone.Collection.extend ({
  model: Message,
  url: URL + "messages/list.json"
});
```

Это те места, где Backbone.js ищет REST API URL-адреса, соответствующие конкретной коллекции и модели.

Вот полный исходный код файла rpjs/board/app.js[4]:

```
/*
Rapid Prototyping with JS is a JavaScript
and Node.js book that will teach you how to
build mobile and web apps fast. — Read more at
http://rapidprototypingwithjs.com.
*/

// var URL = "http://localhost:5000/";
var URL = "http://your-app-name.herokuapp.com/";

require([
  'libs/text!header.html',
  'libs/text!home.html',
  'libs/text!footer.html'],

```

[3] https://github.com/azat-co/rpjs/tree/master/board

[4] https://github.com/azat-co/rpjs/blob/master/board/app.js

```
16  function(
17  headerTpl,
18  homeTpl,
19  footerTpl) {
20
21    var ApplicationRouter = Backbone.Router.extend({
22      routes: {
23        "": "home",
24        "*actions": "home"
25      },
26      initialize: function() {
27        this.headerView = new HeaderView();
28        this.headerView.render();
29        this.footerView = new FooterView();
30        this.footerView.render();
31      },
32      home: function() {
33        this.homeView = new HomeView();
34        this.homeView.render();
35      }
36    });
37
38    HeaderView = Backbone.View.extend({
39      el: "#header",
40      templateFileName: "header.html",
41      template: headerTpl,
42      initialize: function() {},
43      render: function() {
44        $(this.el).html(_.template(this.template));
45      }
46    });
47
48    FooterView = Backbone.View.extend({
49      el: "#footer",
50      template: footerTpl,
51      render: function() {
52        this.$el.html(_.template(this.template));
53      }
54    });
55    Message = Backbone.Model.extend({
56      url: URL + "messages/create.json"
57    })
58    MessageBoard = Backbone.Collection.extend({
59      model: Message,
60      url: URL + "messages/list.json"
```

```
61    });
62
63    HomeView = Backbone.View.extend({
64      el: "#content",
65      template: homeTpl,
66      events: {
67        "click #send": "saveMessage"
68      },
69
70      initialize: function() {
71        this.collection = new MessageBoard();
72        this.collection.bind("all", this.render, this);
73        this.collection.fetch();
74        this.collection.on("add", function(message) {
75          message.save(null, {
76            success: function(message) {
77              console.log('saved ' + message);
78            },
79            error: function(message) {
80              console.log('error');
81            }
82          });
83          console.log('saved' + message);
84        })
85      },
86      saveMessage: function() {
87        var newMessageForm = $("#new-message");
88        var username = newMessageForm.find('[name="username"]')
89          .attr('value');
90        var message = newMessageForm.find('[name="message"]')
91          .attr('value');
92        this.collection.add({
93          "username": username,
94          "message": message
95        });
96      },
97      render: function() {
98        console.log(this.collection)
99        $(this.el).html(_.template(
100         this.template,
101         this.collection
102       ));
103     }
104   });
105
```

```
  app = new ApplicationRouter();
  Backbone.history.start();
});
```

7.3 Приложение Chat

Исходный код Node.js back-end приложения в GitHub-папке rpjs/node[5] имеет такую структуру:

```
/node
        -web.js
        -Procfile
        -package.json
```

Вот исходный код web.js нашего Node.js-приложения с CORS заголовками:

```
/*
Rapid Prototyping with JS is a JavaScript
and Node.js book that will teach you how to build mobile
and web apps fast. — Read more at
http://rapidprototypingwithjs.com.
*/

var http = require('http');
var util = require('util');
var querystring = require('querystring');
var mongo = require('mongodb');

var host = process.env.MONGOHQ_URL ||
  "mongodb://localhost:27017/board";
//MONGOHQ_URL=mongodb://user:pass@server.mongohq.com/db_name


mongo.Db.connect(host, function(error, client) {
  if (error) throw error;
  var collection = new mongo.Collection(client, 'messages');
  var app = http.createServer( function (request, response) {
    var origin = (request.headers.origin || "*");
    if (request.method=="OPTIONS") {
      response.writeHead("204", "No Content", {
        "Access-Control-Allow-Origin": origin,
        "Access-Control-Allow-Methods": "GET, POST, PUT, DELETE,
        OPTIONS",
```

[5] https://github.com/azat-co/rpjs/tree/master/node

```
28          "Access-Control-Allow-Headers": "content-type, accept",
29          "Access-Control-Max-Age": 10, // Seconds.
30          "Content-Length": 0
31        });
32        response.end();
33      };
34      if (request.method==="GET"&&
35        request.url==="/messages/list.json") {
36        collection.find().toArray(function(error,results) {
37          var body = JSON.stringify(results);
38          response.writeHead(200,{
39            'Access-Control-Allow-Origin': origin,
40            'Content-Type':'text/plain',
41            'Content-Length':body.length
42          });
43          console.log("LIST OF OBJECTS: ");
44          console.dir(results);
45          response.end(body);
46        });
47      };
48      if (request.method === "POST" &&
49        request.url === "/messages/create.json") {
50        request.on('data', function(data) {
51          console.log("RECEIVED DATA:")
52          console.log(data.toString('utf-8'));
53          collection.insert(JSON.parse(data.toString('utf-8')),
54          {safe:true}, function(error, obj) {
55            if (error) throw error;
56            console.log("OBJECT IS SAVED: ")
57            console.log(JSON.stringify(obj))
58            var body = JSON.stringify(obj);
59            response.writeHead(200,{
60                  'Access-Control-Allow-Origin': origin,
61              'Content-Type':'text/plain',
62              'Content-Length':body.length
63            });
64            response.end(body);
65          })
66        })
67
68      };
69
70    });
71    var port = process.env.PORT || 5000;
72    app.listen(port);
```

```
73 })
```

7.4 Развертка

Для вашего удобства front-end приложение находится в папке rpjs/board[6], а back-end приложение с CORS в rpjs/node[7]. Теперь вы наверное знаете, что делать, но, чисто для справки, ниже перечислены шаги необходимые для развертывания этих примеров на Heroku.

В папке **node** выполните:

```
$ git init
$ git add .
$ git commit -am "first commit"
$ heroku create
$ heroku addons:add mongohq:sandbox
$ git push heroku master
```

Скопируйте URL-адрес и вставьте его в файл **board/app.js**, присвоив значение переменной URL. Затем в папке **board** выполните:

```
$ git init
$ git add .
$ git commit -am "first commit"
$ heroku create
$ git push heroku master
$ heroku open
```

7.5 Однодоменная развертка

Однодоменное развертывание *не рекомендуется* для серьезных боевых приложений, поскольку статические активы лучше обслуживаются с веб-серверов типа nginx (не движок ввода/вывода Node.js), к тому же разделение API делает менее сложным тестирование, повышая надежность, и быстрым диагностирование/мониторинг. Однако, подход “одно приложение — один домен” может быть использован для подготовки, тестирования, разработки сред и/или небольших приложений.

Это пример статического сервера Node.js:

[6] https://github.com/azat-co/rpjs/tree/master/board
[7] https://github.com/azat-co/rpjs/tree/master/node

```
var http = require("http"),
  url = require("url"),
  path = require("path"),
  fs = require("fs")
  port = process.argv[2] || 8888;

  http.createServer(function(request, response) {

  var uri = url.parse(request.url).pathname
    , filename = path.join(process.cwd(), uri);

  path.exists(filename, function(exists) {
    if(!exists) {
      response.writeHead(404, {
        "Content-Type": "text/plain"});
      response.write("404 Not Found\n");
      response.end();
      return;
    }

        if (fs.statSync(filename).isDirectory())
          filename += '/index.html';

    fs.readFile(filename, "binary",
      function(err, file) {
              if(err) {
                response.writeHead(500,
                  {"Content-Type": "text/plain"});
                response.write(err + "\n");
                response.end();
                return;
              }
        response.writeHead(200);
        response.write(file, "binary");
        response.end();
      }
    );
  });
}).listen(parseInt(port, 10));

console.log("Static file server running at\n "+
  " => http://localhost:" + port + "/\nCTRL + C to shutdown");
```

Примечание

Другой, более элегантный способ, заключается в использовании Node.js-фреймворков типа Connect (http://www.senchalabs.org/connect/static.html) или Express (http://expressjs.com/guide.html), потому что существует специальный **статичный** middleware (плагин) для JS и CSS активов.

8. BONUS: Статьи Webapplog

Резюме: статьи о сущности асинхронности в Node.js, TDD с Mocha; введение в фреймворки/библиотеки Express.js, Monk, Wintersmith, Derby.

> *"Не переживайте по поводу неудач, вам только один раз нужно принять правильное решение."* — Дрю Хьюстон[1]

Для вашего удобства мы включили в этой главе некоторые из постов о Node.js из Webapplog.com[2] — общедоступном блоге о веб-разработки.

8.1 Асинхронность в Node

8.1.1 Неблокирующий ввод/вывод

Одно из самых больших преимуществ использования Node.js перед Python или Ruby является то, что Node имеет неблокирующий механизм ввода/вывода. Чтобы проиллюстрировать это, позвольте мне использовать пример очереди в кофейне Starbucks. Давайте представим, что каждый человек, стоящий в очереди за напитком, является задачей, а все за прилавком — кассир, касса, бариста — это серверное приложение или сервер. Заказываем ли мы чашку обычного капельного кофе, типа Pike Place, или горячий чай, типа Earl Grey, бариста делает его. Вся очередь ждет, пока напиток сделается и с каждого человека взимается соответствующая плата.

Конечно мы знаем, что вышеупомянутые напитки (времязатратные узкие мест) можно легко сделать: просто залить жидкость и дело сделано. Но что делать тем причудливым шоко-мокко-фраппе-латте-соя-декафе? Что, если бы все в очереди решили заказать эти времязатратные напитки? Очередь будет задерживаться и становиться длиннее и длиннее. Менеджеру кафе придется добавить дополнительных касс и поставить больше барист на работу (или даже самому занять их место).

Это не очень хорошо, верно? Но так работают практически все серверные технологии за исключением Node.js, который, как настоящий Starbucks. Когда вы заказываете что-то, бариста кричит заказ другому сотруднику и вы покидаете кассу. Другой человек делает свой заказ, пока вы ждете ваше произведение искусства в бумажном стаканчике. Очередь движется, процессы выполняются асинхронно и без блокировки очереди.

[1] https://ru.wikipedia.org/wiki/%D0%A5%D1%8C%D1%8E%D1%81%D1%82%D0%BE%D0%BD,_%D0%94%D1%80%D1%8E

[2] http://webapplog.com

Вот почему Node.js бьет всех (за исключением, может быть, низкоуровневого C++) с точки зрения производительности и масштабируемости. С Node.js вам просто не нужно много процессоров и серверов, чтобы справиться с нагрузкой.

8.1.2 Асинхронный способ кодирования

Асинхронность требует иного способа мышления программистов знакомых с Python, PHP, C или Ruby. Легко привнести ошибку, непреднамеренно забыв закончить выполнение кода соответствующим выражением **return**.

Вот простой пример, иллюстрирующий этот сценарий:

```
var test = function (callback) {
  return callback();
  console.log('test') //не должен выводиться
}

var test2 = function(callback){
  callback();
  console.log('test2') //выводится третьим
}

test(function(){
  console.log('callback1') //выводится первым
  test2(function(){
  console.log('callback2') //выводится вторым
  })
});
```

Если мы не используем return callback() и просто используем callback() наша строка test2 будет выведена (test не выводится).

```
callback1
callback2
tes2
```

Ради забавы я добавил задержку `setTimeout()` для строки callback2 и теперь порядок изменился:

```
var test = function (callback) {
  return callback();
  console.log('test') //не должен выводиться
}

var test2 = function(callback){
  callback();
  console.log('test2') //выводится вторым
}

test(function(){
  console.log('callback1') //выводится первым
  test2(function(){
    setTimeout(function(){
      console.log('callback2') //выводится третьим
    },100)
  })
});
```

Выводит:

```
callback1
tes2
callback2
```

Последний пример показывает, что две функции являются независимыми друг от друга и работают параллельно. Быстрая функция закончится раньше, чем медленная. Возвращаясь к нашим примерам Starbucks, вы могли бы получить ваш напиток быстрее, чем другой человек, который был перед вами в очереди. Лучше для людей и лучше для программ! :-)

8.2 MongoDB миграция с Monk

Недавно один из наших топовых пользователей пожаловался, что его аккаунт Storify[3] был недоступен. Мы проверили боевую базу данных и было похоже, что учетная запись могла быть взломана и злостно удалена кем-то, использующим учетные данные пользователя. Благодаря отличному сервису MongoHQ, мы получили бэкап базы данных менее чем за 15 минут. Было два варианта для продолжения миграции:

1. Скрипт для Mongo оболочки
2. Node.js-программа

Из-за того, что удаления учетной записи пользователя Storify предполагает удаление всех связанных объектов — личной информации, отношений (фолловеров, подписок), лайков,

[3] http://storify.com

историй — мы решили использовать последний вариант. Он прекрасно сработал и вот упрощенный вариант, который можно использовать как шаблон для MongoDB-миграции (также в gist.github.com/4516139[4]).

Давайте загрузим все необходимые модули: Monk[5], Progress[6], Async[7] и MongoDB:

```
var async = require('async');
var ProgressBar = require('progress');
var monk = require('monk');
var ObjectId=require('mongodb').ObjectID;
```

Кстати, Monk, созданный LeanBoost[8], это “крошечная прослойка, которая обеспечивает простое, но существенное улучшения юзабилити для использования MongoDB в пределах Node.js”.

Monk принимает строку подключения в следующем формате:

```
username:password@dbhost:port/database
```

Так мы можем создать следующие объекты:

```
var dest = monk('localhost:27017/storify_localhost');
var backup = monk('localhost:27017/storify_backup');
```

Нам потребуется знать ID объекта, который мы хотим восстановить:

```
var userId = ObjectId(YOUR-OBJECT-ID);
```

Это удобная функция `restore()`, которую мы можем использовать, чтобы восстановить объекты из связанных коллекций, указав запрос (подробнее о запросах MongoDB Querying 20M-Record MongoDB Collection[9]). Чтобы вызвать его, просто передайте имя коллекции в виде строки, например, `"stories"` и запрос, который связывает объекты из этой коллекции с вашим основным объектом, например, `{userId:user.id}`. Прогресс бар нужен, чтобы показать нам красивые визуальные эффекты в терминале:

[4] https://gist.github.com/4516139
[5] https://github.com/LearnBoost/monk
[6] https://github.com/visionmedia/node-progress
[7] https://github.com/caolan/async
[8] https://www.learnboost.com/
[9] http://www.webapplog.com/querying-20m-record-mongodb-collection/

```
var restore = function(collection, query, callback){
  console.info('restoring from ' + collection);
  var q = query;
  backup.get(collection).count(q, function(e, n) {
    console.log('found '+n+' '+collection);
    if (e) console.error(e);
    var bar = new ProgressBar('[:bar] :current/:total'
      + ':percent :etas'
      , { total: n-1, width: 40 })
    var tick = function(e) {
      if (e) {
        console.error(e);
        bar.tick();
      }
      else {
        bar.tick();
      }
      if (bar.complete) {
        console.log();
        console.log('restoring '+collection+' is completed');
        callback();
      }
    };
    if (n>0){
      console.log('adding '+ n+ ' '+collection);
      backup.get(collection).find(q, {
        stream: true
      }).each(function(element) {
        dest.get(collection).insert(element, tick);
      });
    } else {
      callback();
    }
  });
}
```

Теперь мы можем использовать асинхронный вызов функции `restore()` упомянутой выше:

```
async.series({
  restoreUser: function(callback){   // импорт пользовательских элементов
    backup.get('users').find({_id:userId}, {
      stream: true, limit: 1
    }).each(function(user) {
      dest.get('users').insert(user, function(e){
        if (e) {
          console.log(e);
        }
        else {
          console.log('resored user: '+ user.username);
        }
        callback();
      });
    });
  },

  restoreIdentity: function(callback){
    restore('identities',{
      userid:userId
    }, callback);
  },

  restoreStories: function(callback){
    restore('stories', {authorid:userId}, callback);
  }

  }, function(e) {
  console.log();
  console.log('restoring is completed!');
  process.exit(1);
});
```

Полный код доступен по адресу gist.github.com/4516139[10] и здесь:

[10] https://gist.github.com/4516139

```
var async = require('async');
var ProgressBar = require('progress');
var monk = require('monk');
var ms = require('ms');
var ObjectId=require('mongodb').ObjectID;

var dest = monk('localhost:27017/storify_localhost');
var backup = monk('localhost:27017/storify_backup');

var userId = ObjectId(YOUR-OBJECT-ID);
// monk должен иметь авто срабатывание, но нам нужно это для запросов

var restore = function(collection, query, callback){
  console.info('restoring from ' + collection);
  var q = query;
  backup.get(collection).count(q, function(e, n) {
    console.log('found '+n+' '+collection);
    if (e) console.error(e);
    var bar = new ProgressBar(
      '[:bar] :current/:total :percent :etas',
      { total: n-1, width: 40 })
    var tick = function(e) {
      if (e) {
        console.error(e);
        bar.tick();
      }
      else {
        bar.tick();
      }
      if (bar.complete) {
        console.log();
        console.log('restoring '+collection+' is completed');
        callback();
      }
    };
    if (n>0){
      console.log('adding '+ n+ ' '+collection);
      backup.get(collection).find(q, { stream: true })
        .each(function(element) {
        dest.get(collection).insert(element, tick);
      });
    } else {
      callback();
    }
  });
```

```
}

async.series({
  restoreUser: function(callback){   // импорт пользовательских элементов
    backup.get('users').find({_id:userId}, {
      stream: true,
      limit: 1 })
      .each(function(user) {
      dest.get('users').insert(user, function(e){
        if (e) {
          console.log(e);
        }
        else {
          console.log('resored user: '+ user.username);
        }
        callback();
      });
    });
  },

  restoreIdentity: function(callback){
    restore('identities',{
      userid:userId
    }, callback);
  },

  restoreStories: function(callback){
    restore('stories', {authorid:userId}, callback);
  }

  }, function(e) {
  console.log();
  console.log('restoring is completed!');
  process.exit(1);
});
```

Чтобы запустить его, выполните `npm install`/`npm update` и измените захардкоженные значения в базе данных.

8.3 TDD в Node.js с Mocha

8.3.1 Кому нужна разработка через тестирование?

Представьте, что вам необходимо реализовать комплекс фич в существующем интерфейсе, например, кнопку ‘like’ в комментарии. Без тестов, вам придется вручную создать пользо-

вателя, войти в систему, создать пост, создать другого пользователя, войти в систему под другим пользователем и лайкнуть пост. Утомительно? Что, если вам придется делать это 10 или 20 раз, чтобы найти и исправить какую-то неприятную ошибку? Что делать, если ваша функция разбивает существующую функциональную возможность, но вы заметите это только через шесть месяцев после релиза, потому что не было теста?!

Не тратьте время на написание тестов для проходных скриптов, но, пожалуйста, возьмите за привычку писать их для основного кода приложения. Немного времени потраченного в начале позволит вам и вашей команде сэкономить время позднее и иметь уверенность при развертывании новых релизов. Разработка через тестирование действительно очень хорошая вещь.

8.3.2 Руководство по быстрому старту

Прочтите это краткое руководство, чтобы настроить ваш TDD-процесс в Node.js с Mocha[11].

Установите Mocha[12] глобально, выполнив эту команду:

```
$ sudo npm install -g mocha
```

Мы также будем использовать две библиотеки: Superagent[13] и expect.js[14] от LearnBoost[15]. Чтобы установить их, запустите NPM[16]-команду в папке проекта:

```
$ npm install superagent
$ npm install expect.js
```

Откройте новый файл с расширением `.js` и введите:

```
var request = require('superagent');
var expect = require('expect.js');
```

Пока мы включили две библиотеки. Структура набора тестов (test suite) будут выглядеть так:

[11] http://visionmedia.github.com/mocha/
[12] http://visionmedia.github.com/mocha/
[13] https://github.com/visionmedia/superagent
[14] https://github.com/LearnBoost/expect.js/
[15] https://github.com/LearnBoost
[16] https://npmjs.org/

```
describe('Suite one', function(){
  it(function(done){
  ...
  });
  it(function(done){
  ...
  });
});
describe('Suite two', function(){
  it(function(done){
  ...
  });
});
```

Внутри этого замыкания, мы можем написать запрос нашему серверу, который должен быть запущен по адресу localhost:8080[17]:

```
...
it (function(done){
  request.post('localhost:8080').end(function(res){
    //TODO проверить, что ответ в порядке
  });
});
...
```

Expect даст нам удобные функции для проверки любых условиях, которые мы сможем придумать:

```
...
expect(res).to.exist;
expect(res.status).to.equal(200);
expect(res.body).to.contain('world');
...
```

Наконец, мы должны добавить вызов **done()** для уведомления Mocha, что асинхронный тест завершил свою работу. Теперь полный код нашего первого теста выглядит так:

[17] http://localhost:8080

```
var request = require('superagent');
var expect = require('expect.js');

describe('Suite one', function(){
 it (function(done){
   request.post('localhost:8080').end(function(res){
    expect(res).to.exist;
    expect(res.status).to.equal(200);
    expect(res.body).to.contain('world');
    done();
  });
});
```

Если хотим, мы можем добавить хуки **before** и **beforeEach**, которые, соответственно их именам, выполнятся один раз перед тестом (или набором тестов) или каждый раз перед тестом (или набором):

```
before(function(){
  //TODO заполнения базы данных
});
describe('suite one ',function(){
  beforeEach(function(){
    //todo логин тестового юзера
  });
  it('test one', function(done){
  ...
  });
});
```

Обратите внимание, что before и beforeEach могут быть размещены внутри или снаружи конструкции describe.

Чтобы запустить наш тест, просто выполните:

```
$ mocha test.js
```

Для использования другого типа отчета:

```
$ mocha test.js -R list
$ mocah test.js -R spec
```

8.4 Wintersmith — генератор статических сайтов

Для одностраничного сайта этой книги —rapidprototypingwithjs.com[18] — я использовал Wintersmith[19], чтобы узнать что-то новое и для быстрой загрузки. Wintersmith — это Node.js генератор

[18] http://rapidprototypingwithjs.com

[19] http://jnordberg.github.com/wintersmith/

статических сайтов. Он сильно впечатлил меня своей гибкостью и простотой разработки. Кроме того, я мог придерживаться моих любимых инструментов, таких как Markdown[20], Jade и Underscore[21].

Почему генераторы статических сайтов

Вот хорошая статья о том, почему использование генератора ститических сайтов хорошая идея в целом: An Introduction to Static Site Generators[22]. В основном он сводится к нескольким основным вещам:

Шаблоны

Вы можете использовать шаблонизаторы, например, Jade[23]. Jade использует пробелы для структурирования вложенных элементов, и его синтаксис аналогичен Ruby on Rails разметки Haml.

Markdown

Я скопировал текст из главы "Введение" моей книги и использовал его без каких-либо модификаций. Wintersmith поставляется с парсером marked[24] по умолчанию. Подробнее о том, почему Markdown хорош в моем старом посте: Markdown Goodness[25].

Простое развертывание

Все, что мы имеем, это файлы HTML, CSS и JavaScript, так что вы просто загружаете их с помощью FTP-клиента, например, Transmit[26] (от Panic) или Cyberduck[27].

Простой хостинг

Благодаря тому, что любой статический веб-сервер будет замечательно работать, нет необходимости в Heroku или Nodejitsu PaaS-решениях или даже PHP/MySQL хостинге.

Производительность

Нет обращений к базе данных, нет server-side API вызовов и накладных расходов CPU/RAM.

Гибкость

Wintersmith позволяет использовать разные плагины для содержания и шаблонов и вы можете даже писать свои плагины[28].

8.4.1 Приступая к работе с Wintersmith

Есть краткое руководство по началу работы на github.com/jnordberg/wintersmith[29].

Чтобы установить Wintersmith глобально, запустите NPM с -g и sudo:

```
1 $ sudo npm install wintersmith -g
```

Затем запустите для использования стандартный шаблон блога:

[20] http://daringfireball.net/projects/markdown/
[21] http://underscorejs.org/
[22] http://www.mickgardner.com/2012/12/an-introduction-to-static-site.html
[23] https://github.com/visionmedia/jade
[24] https://github.com/chjj/marked
[25] http://www.webapplog.com/markdown-goodness/
[26] http://www.panic.com/transmit/
[27] http://cyberduck.ch/
[28] https://github.com/jnordberg/wintersmith#content-plugins
[29] https://github.com/jnordberg/wintersmith

```
$ wintersmith new <path>
```

или для пустого сайта:

```
$ wintersmith new <path> -template basic
```

или используйте сокращение:

```
$ wintersmith new <path> -T basic
```

Похожий на Ruby on Rails скаффолдинг, Wintersmith будет генерировать основной скелет с папками **contents** и **templates**. Чтобы просмотреть веб-сайт, выполните следующие команды:

```
$ cd <path>
$ wintersmith preview
$ open http://localhost:8080
```

Большинство изменений будет обновлятся автоматически в режиме предварительного просмотра, за исключением config.json file[30].

Изображения, CSS, JavaScript и другие файлы идут в папку **contents**. Генератор Wintersmith имеет следующую логику:

1. ищет *.md-файлы в папке contents
2. читает метаданные[31], такие как имя шаблона
3. обрабатывает *.jade-шаблоны[32] согласно метаданным в *.md-файлах

Когда вы закончите со своим статическим сайтом, просто запустите:

```
$ wintersmith build
```

8.4.2 Другие генераторы статических сайтов

Вот некоторые другие Node.js генераторы статических сайтов:

- DocPad[33]
- Blacksmith[34]
- Scotch[35]
- Wheat[36]

[30] https://github.com/jnordberg/wintersmith#config
[31] https://github.com/jnordberg/wintersmith#the-page-plugin
[32] https://github.com/jnordberg/wintersmith#templates
[33] https://github.com/bevry/docpad#readme
[34] https://github.com/flatiron/blacksmith
[35] https://github.com/techwraith/scotch
[36] https://github.com/creationix/wheat

- Petrify[37]

Более подробный обзор этих генераторов статических сайтов доступен в посте Node.js Based Static Site Generators[38].

Для других языков и фреймворков, типа Rails и PHP, посмотрите Static Site Generators by GitHub Watcher Count[39] и "мать всех генераторов статических сайтов[40]".

8.5 Введение в Express.js: простое REST API приложение с Monk и MongoDB

8.5.1 REST API приложение с Express.js и Monk

Это приложение является началом проекта mongoui[41] — аналог phpMyAdmin для MongoDB, написанный на Node.js. Цель — создать модуль с хорошим админским веб-интерфейсом пользователя. Это будет нечто вроде Parse.com[42], Firebase.com[43], MongoHQ[44] или MongoLab[45], но без привязки к какому-либо определенному сервису. Почему мы должны писать `db.users.findOne({'_id':ObjectId('...')})` каждый раз, когда мы хотим найти информацию о пользователе? Альтернатива Mac-приложение MongoHub[46] хороша (и бесплатна), но неуклюжа в использовании и не web-based.

Ruby-энтузиасты любят сравнивать Express и Sinatra[47]. Они аналогичны по своей гибкости. Маршруты приложений настраиваются аналогичным образом, т.е., `app.get('/products/:id', showProduct);`. В настоящее время Express.js имеет версию 3.1. В дополнение к Express мы будем использовать модуль Monk[48].

Мы будем использовать Node Package Manager[49], который обычно поставляется вместе с Node.js. Если вы его еще не установили, вы можете взять его на npmjs.org[50].

Создайте новую папку и NPM файл конфигурации **package.json** со следующим содержимым:

[37] https://github.com/caolan/petrify
[38] http://blog.bmannconsulting.com/node-static-site-generators/
[39] https://gist.github.com/2254924
[40] http://nanoc.stoneship.org/docs/1-introduction/#similar-projects
[41] http://gitbhub.com/azat-co/mongoui
[42] http://parse.com
[43] http://firebase.com
[44] http://mongohq.com
[45] http://mongolab.com
[46] http://mongohub.todayclose.com/
[47] http://www.sinatrarb.com/
[48] https://github.com/LearnBoost/monk
[49] http://npmjs.org
[50] http://npmjs.org

```
{
  "name": "mongoui",
  "version": "0.0.1",
  "engines": {
    "node": ">= v0.6"
  },
  "dependencies": {
    "mongodb":"1.2.14",
    "monk": "0.7.1",
    "express": "3.1.0"
  }
}
```

Теперь запустите `npm install` для загрузки и установки модулей в папку **node_modules**. Если все прошло хорошо, вы увидите кучу папок в папке **node_modules**. Весь код нашего приложения будет находиться в одном файле **index.js**, чтобы делать все как можно проще (keep it simple stupid):

```
var mongo = require('mongodb');
var express = require('express');
var monk = require('monk');
var db =  monk('localhost:27017/test');
var app = new express();

app.use(express.static(__dirname + '/public'));
app.get('/',function(req,res){
  db.driver.admin.listDatabases(function(e,dbs){
      res.json(dbs);
  });
});
app.get('/collections',function(req,res){
  db.driver.collectionNames(function(e,names){
    res.json(names);
  })
});
app.get('/collections/:name',function(req,res){
  var collection = db.get(req.params.name);
  collection.find({},{limit:20},function(e,docs){
    res.json(docs);
  })
});
app.listen(3000)
```

Давайте разберем этот код по частям. Объявления модуля:

```
var mongo = require('mongodb');
var express = require('express');
var monk = require('monk');
```

Создание экземпляров базы данных и Express-приложения:

```
var db =  monk('localhost:27017/test');
var app = new express();
```

Говорим Express-приложению загрузить серверные статические файлы (если таковые имеются) из папки public:

```
app.use(express.static(__dirname + '/public'));
```

Установка главной страницы, a.k.a. маршрут корня:

```
app.get('/',function(req,res){
  db.driver.admin.listDatabases(function(e,dbs){
      res.json(dbs);
  });
});
```

Функция `get()` принимает два параметра: строку и функцию. Строка может иметь слеши и двоеточия, например, `product/:id`. Эта функция должна иметь два параметра: request и response. Request включает в себя информацию, такую как параметры строки запроса, сессию и заголовки. Response — это объект, в который мы выводим результаты. В этом случае, мы делаем это путем вызова функции `res.json()`.

`db.driver.admin.listDatabases()`, как вы уже могли догадаться, выдает нам список баз данных в асинхронном режиме.

Два других маршрута настраиваются аналогично функцией `get()`:

```
app.get('/collections',function(req,res){
  db.driver.collectionNames(function(e,names){
    res.json(names);
  })
});
app.get('/collections/:name',function(req,res){
  var collection = db.get(req.params.name);
  collection.find({},{limit:20},function(e,docs){
    res.json(docs);
  })
});
```

Express удобно поддерживает другие методы HTTP, такие как post и update. В случае установки post-маршрута, мы пишем это:

```
app.post('product/:id',function(req,res) {...});
```

Express также имеет поддержку middleware. Middleware — это просто функция обработки запроса с тремя параметрами: `request`, `response` и `next`. Например:

```
app.post('product/:id',
  authenticateUser,
  validateProduct,
  addProduct
);

function authenticateUser(req,res, next) {
  //проверка req.session для аутентификации
  next();
}

function validateProduct (req, res, next) {
   //валидация данных
   next();
}

function addProduct (req, res) {
  //сохранение данных в базу
}
```

validateProduct и authenticateProduct являются middleware. Они обычно помещаются в отдельный файл (или файлы) в больших проектах.

Другой способ установки middleware в Express-приложение является использование функции `use()`. Например, раньше мы делали это для статических ресурсов:

```
app.use(express.static(__dirname + '/public'));
```

Мы также можем сделать это для обработчика ошибок:

```
app.use(errorHandler);
```

Предположим, что вы уже установили mongoDB, тогда это приложение будет подключаться к нему (localhost:27017[51]) и показывать имя коллекции и элементы в коллекции. Чтобы запустить сервер mongo:

```
$ mongod
```

чтобы запустить приложение (держите окно терминала **mongod** открытым):

[51] http://localhost:27017

```
$ node .
```

или

```
$ node index.js
```

Чтобы увидеть работающее приложение, откройте localhost:3000[52] в Chrome с расширением JSONViewer[53] (для удобного рендеринга JSON).

8.6 Введение в Express.js: параметры, обработка ошибок и другие middleware

8.6.1 Обработчики запросов

Express.js — это node.js фреймворк, который, среди прочего, дает возможность организовать маршруты. Каждый маршрут определяется с помощью вызова метода объекта приложения с шаблоном URL-адреса, определенным в качестве первого параметра (RegExp также поддерживается), например:

```
app.get('api/v1/stories/', function(res, req){
  ...
})
```

или для POST-метода:

```
app.post('/api/v1/stories'function(req,res){
  ...
})
```

Само собой разумеется, методы DELETE и PUT так же поддерживаются[54].

Коллбэки, которые мы передаем в методы `get()` или `post()`, вызываются обработчиками запросов, потому что они принимают запросы (`req`), обрабатывают их и пишут в объект ответа (`res`). Например:

```
app.get('/about', function(req,res){
  res.send('About Us: ...');
});
```

Мы можем иметь несколько обработчиков запросов — отсюда и название *middleware*. Они принимают третий параметр `next`, вызыв которого (`next()`) переключает поток исполнения на следующий обработчик:

[52] http://localhost:3000

[53] https://chrome.google.com/webstore/detail/jsonview/chklaanhfefbnpoihckbnefhakgolnmc?hl=en

[54] http://expressjs.com/api.html#app.VERB

```
app.get('/api/v1/stories/:id', function(req,res, next) {
  //делаем авторизацию
  //если не авторизован или есть ошибки
  // возвращаем next(error);
  //если авторизован и нет ошибок
  return next();
}), function(req,res, next) {
  //экстракт id и выборка объекта из базы данных
  //при условии отсутствия ошибок, сохранить историю в объекте запроса
  req.story = story;
  return next();
}), function(req,res) {
  //вывод результат поиска в базе данных
  res.send(res.story);
});
```

ID истории в шаблоне URL является параметром строки запроса, который нам нужен для нахождения подходящих элементов в базе данных.

8.6.2 Параметры middleware

Параметры — это значения, которые передаются в строке запроса URL-адреса запроса. Если бы у нас не было Express.js или аналогичной библиотеки и пришлось бы использовать только модули ядра Node.js, тогда мы бы извлекали параметры из объекта HTTP.request[55] с помощью какой-нибудь функции `require('querystring').parse(url)` или `require('url').parse(url, true)`.

Благодаря Connect framework[56] и людям из VisionMedia[57], Express.js уже имеет поддержку параметров, обработку ошибок и много других важных функций в виде middleware. Вот как мы можем подключить middleware param в нашем приложении:

```
app.param('id', function(req,res, next, id){
  //делаем что-то с id
  //сохраняем id или другую информацию в объекте req
  //вызываем next
  next();
});

app.get('/api/v1/stories/:id',function(req,res){
  //middleware param будет выполняться до и
  //мы ожидаем, что объект req уже имеют необходимую информацию
  //выводим что-нибудь
  res.send(data);
});
```

[55] http://nodejs.org/api/http.html#http_http_request_options_callback
[56] http://www.senchalabs.org/connect/
[57] https://github.com/visionmedia/express

Например:

```
app.param('id', function(req,res, next, id){
  req.db.get('stories').findOne({_id:id}, function (e, story){
    if (e) return next(e);
    if (!story) return next(new Error('Nothing is found'));
    req.story = story;
    next();
  });
});

app.get('/api/v1/stories/:id',function(req,res){
  res.send(req.story);
});
```

Или мы можем использовать несколько обработчиков запросов, но концепция остается той же: мы можем ожидать объект `req.story` или ошибку, выданную до выполнения этого кода, поэтому мы абстрагируем общий код/логику получения параметров и их объектов:

```
app.get('/api/v1/stories/:id', function(req,res, next) {
  //делаем авторизацию
  }),
  //у нас есть объект в req.story, поэтому нет необходимости делать что-то здесь
  function(req,res) {
  //вывод результата поиска из базы данных
  res.send(story);
});
```

Авторизация и очистка введенных данных также являются хорошими кандидатами для middleware.

Функция `param()` особенно хороша, потому что мы можем комбинировать различные ключи, напр.:

```
app.get('/api/v1/stories/:storyId/elements/:elementId',
  function(req,res){
    res.send(req.element);
  }
);
```

8.6.3 Обработка ошибок

Обработка ошибок обычно используется везде в приложении, поэтому лучше ее реализовать как middleware. Она имеет те же параметры, плюс еще один `error`:

```
app.use(function(err, req, res, next) {
  //логирование и показ user-friendly сообщения об ошибке
  res.send(500);
})
```

В самом деле, ответом может быть что угодно:

JSON-строка

```
app.use(function(err, req, res, next) {
  //логирование и показ user-friendly сообщения об ошибке
  res.send(500, {status:500,
    message: 'internal error',
    type:'internal'}
  );
})
```

Текстовое сообщение

```
app.use(function(err, req, res, next) {
  //логирование и показ user-friendly сообщения об ошибке
  res.send(500, 'internal server error');
})
```

Страница ошибки

```
app.use(function(err, req, res, next) {
  //логирование и показ user-friendly сообщения об ошибке
  //полагая, что шаблонизатор подключен
  res.render('500');
})
```

Редирект на страницу ошибки

```
app.use(function(err, req, res, next) {
  //логирование и показ user-friendly сообщения об ошибке
  res.redirect('/public/500.html');
})
```

Статус ошибки HTTP-ответа (401, 400, 500 и т.д.)

```
app.use(function(err, req, res, next) {
  //логирование и показ user-friendly сообщения об ошибке
  res.end(500);
})
```

Кстати, логирование также должно быть абстрагировано в middleware!

Для того чтобы сработала ошибка в ваших обработчиках запросов и middleware, вы можете просто вызвать:

```
next(error);
```

или

```
next(new Error('Something went wrong :-(');
```

Вы также можете иметь несколько обработчиков ошибок и пользоваться ими вместо анонимных функций, как это показано в Express.js Error handling guide[58].

8.6.4 Другие middleware

В дополнение к извлечению параметров, он может быть использован для многих вещей, таких как авторизация, обработка ошибок, сессии, вывод и другие.

`res.json()` является одним из них. Он удобно выводит JavaScript/Node.js объект в виде JSON. Например:

```
app.get('/api/v1/stories/:id', function(req,res){
  res.json(req.story);
});
```

эквивалентно (если `req.story` — это массив и объект):

```
app.get('/api/v1/stories/:id', function(req,res){
  res.send(req.story);
});
```

или

```
app.get('api/v1/stories/:id',function(req,res){
  res.set({
    'Content-Type': 'application/json'
  });
  res.send(req.story);
});
```

8.6.5 Абстракция

Middleware гибки. Вы можете использовать анонимные или именованные функции, но самое главное, абстрагировать обработчики запросов во внешние модули, основанные на функциональности:

[58] http://expressjs.com/guide.html#error-handling

```
var stories = require.('./routes/stories');
var elements = require.('./routes/elements');
var users = require.('./routes/users');
...
app.get('/stories/,stories.find);
app.get('/stories/:storyId/elements/:elementId', elements.find);
app.put('/users/:userId',users.update);
```

routes/stories.js:

```
module.exports.find = function(req,res, next) {
};
```

routes/elements.js:

```
module.exports.find = function(req,res,next){
};
```

routes/users.js:

```
module.exports.update = function(req,res,next){
};
```

Вы можете использовать некоторые приемы функционального программирования:

```
function requiredParamHandler(param){
  //делаем что-нибудь с param, например,
  //проверяем, что находится в строке запроса
  return function (req,res, next) {
    //используем param, напр., если token валидный продолжаем с next();
    next();
  });
}

app.get('/api/v1/stories/:id',
  requiredParamHandler('token'),
  story.show
);

var story  = {
  show: function (req, res, next) {
    //делаем какую-то логику, напр., ограничения полей для вывода
    return res.send();
  }
}
```

Как вы можете видеть, middleware — это мощная концепция для организации кода. Хорошей практикой является хранение маршрутизатора худым и тонким, перемещая всю логику в соответствующие внешние модули/файлы. Таким образом, важные параметры конфигурации сервера будут в одном месте, когда они вам понадобятся! :-)

8.7 JSON REST API сервер с Node.js и MongoDB с использованием Mongoskin и Express.js

Этот туториал проведет вас через написание тестов с помощью библиотек Mocha[59] и Super Agent[60], а затем использования их в test-driven development, для создания бесплатного Node.js[61] JSON REST API сервера, использующего фреймворк Express.js[62] и библиотеку Mongoskin[63] для MongoDB[64]. В этом REST API сервере, мы будем выполнять операции **create**, **update**, **remove и delete** (CRUD) и использовать концепт Express.js middleware[65] с методами `app.param()` и `app.use()`.

8.7.1 Покрытие тестами

Прежде чем что-либо делать, давайте напишем функциональные тесты, которые делают HTTP-запросы к нашему будущему REST API серверу. Если вы знаете, как использовать Mocha[66] или просто хотите сразу перейти к реализации Express.js-приложения, не стесняйтесь сделать это. Вы можете использовать CURL команды терминала для тестирования.

Предположим, что мы уже установили Node.js, NPM[67] и MongoDB, давайте создадим *новую* папку (или, если вы уже написали тесты, используйте эту папку):

```
mkdir rest-api
cd rest-api
```

Мы будем использовать библиотеки Mocha[68], Expect.js[69] и Super Agent[70]. Для их установки запустите следующие команды из папки проекта:

```
$ npm install mocha
$ npm install expect.js
$ npm install superagent
```

Теперь давайте создадим файл **express.test.js** в той же папке, которая будет иметь шесть наборов:

- создание нового объекта

[59] http://visionmedia.github.io/mocha/
[60] http://visionmedia.github.io/superagent/
[61] http://nodejs.org
[62] http://expressjs.com/
[63] https://github.com/kissjs/node-mongoskin
[64] http://www.mongodb.org/
[65] http://expressjs.com/api.html#middleware
[66] http://visionmedia.github.io/mocha/
[67] http://npmjs.org
[68] http://visionmedia.github.io/mocha/
[69] https://github.com/LearnBoost/expect.js/
[70] http://visionmedia.github.io/superagent/

- получение объекта по его ID
- получение всей коллекции
- обновление объекта по его ID
- проверка обновленного объекта по его ID
- удаление объекта по его ID

HTTP-запросы легки как бриз с функциями Super Agent, которые мы будем ставить внутри каждого набора тестов. Вот полный исходный код файла **express.test.js**:

```
var superagent = require('superagent')
var expect = require('expect.js')

describe('express rest api server', function(){
  var id

  it('post object', function(done){
    superagent.post('http://localhost:3000/collections/test')
      .send({ name: 'John'
        , email: 'john@rpjs.co'
      })
      .end(function(e,res){
        // console.log(res.body)
        expect(e).to.eql(null)
        expect(res.body.length).to.eql(1)
        expect(res.body[0]._id.length).to.eql(24)
        id = res.body[0]._id
        done()
      })
  })

  it('retrieves an object', function(done){
    superagent.get('http://localhost:3000/collections/test/'+id)
      .end(function(e, res){
        // console.log(res.body)
        expect(e).to.eql(null)
        expect(typeof res.body).to.eql('object')
        expect(res.body._id.length).to.eql(24)
        expect(res.body._id).to.eql(id)
        done()
      })
  })

  it('retrieves a collection', function(done){
    superagent.get('http://localhost:3000/collections/test')
      .end(function(e, res){
```

```
37          // console.log(res.body)
38          expect(e).to.eql(null)
39          expect(res.body.length).to.be.above(1)
40          expect(res.body.map(function (item){
41            return item._id
42          })).to.contain(id)
43          done()
44        })
45    })
46
47    it('updates an object', function(done){
48      superagent.put('http://localhost:3000/collections/test/'+id)
49        .send({name: 'Peter'
50          , email: 'peter@yahoo.com'})
51        .end(function(e, res){
52          // console.log(res.body)
53          expect(e).to.eql(null)
54          expect(typeof res.body).to.eql('object')
55          expect(res.body.msg).to.eql('success')
56          done()
57        })
58    })
59
60    it('checks an updated object', function(done){
61      superagent.get('http://localhost:3000/collections/test/'+id)
62        .end(function(e, res){
63          // console.log(res.body)
64          expect(e).to.eql(null)
65          expect(typeof res.body).to.eql('object')
66          expect(res.body._id.length).to.eql(24)
67          expect(res.body._id).to.eql(id)
68          expect(res.body.name).to.eql('Peter')
69          done()
70        })
71    })
72
73    it('removes an object', function(done){
74      superagent.del('http://localhost:3000/collections/test/'+id)
75        .end(function(e, res){
76          // console.log(res.body)
77          expect(e).to.eql(null)
78          expect(typeof res.body).to.eql('object')
79          expect(res.body.msg).to.eql('success')
80          done()
81        })
```

```
82    })
83  })
```

Для запуска тестов мы можем использовать команду `$ mocha express.test.js`.

8.7.2 Зависимости

В этом уроке мы будем использовать Mongoskin[71], MongoDB-библиотеку, которая является лучшей альтернативой для обычного старого доброго нативного драйвера MongoDB для Node.js[72]. В дополнение, Mongoskin более легковесен, чем Mongoose. Для большей информации обратитесь к Mongoskin comparison blurb[73].

Express.js[74] является оболочкой для объектов HTTP-модуля[75] ядра Node.js. Фреймворк Express.js — это надстройка над middleware Connect[76] и предоставляет тонну удобств. Некоторые люди сравнивают фреймворк с Sinatra в плане того, что он недубовый и настраиваемый.

Если вы создали папку `rest-api` в предыдущем разделе *Покрытие тестами*, просто выполните следующие команды, чтобы установить модули для приложения:

```
npm install express
npm install mongoskin
```

8.7.3 Реализация

Прежде всего, определим наши зависимости:

```
var express = require('express')
  , mongoskin = require('mongoskin')
```

После версии 3.x, Express упрощает создание его экземпляра приложения, таким образом, что данная строка даст нам объект сервера:

```
var app = express()
```

Для извлечения параметров из тел запросов мы будем использовать middleware `bodyParser()`, который больше похож на объявление конфигураций:

```
app.use(express.bodyParser())
```

[71] https://github.com/kissjs/node-mongoskin
[72] https://github.com/mongodb/node-mongodb-native
[73] https://github.com/kissjs/node-mongoskin#comparation
[74] http://expressjs.com/
[75] http://nodejs.org/api/http.html
[76] https://github.com/senchalabs/connect

Middleware (в этой[77] и других видах[78]) — это мощный и удобный паттерн в Express.js и Connect[79] для организации и повторного использования кода.

Как и в случае с методом `bodyParser()`, который спасает нас от проблем с парсингом тела объекта HTTP-запроса, Mongoskin делает возможным подключение к базе данных MongoDB в одну строку кода:

```
var db = mongoskin.db('localhost:27017/test', {safe:true});
```

Примечание: *если вы хотите подключиться к удаленной базе данных, напр., экземпляр MongoHQ[80], замените строку со значениями ваших имени пользователя, пароля, хоста и порта. Вот формат строки URI:* `mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:`

Метод `app.param()` — это еще один middleware Express.js. Он просто говорит: "*делай что-то каждый раз, когда появляется это значение в шаблоне URL-адреса обработчика запроса*". В нашем случае мы выбераем конкретную коллекцию, когда шаблон запроса содержит строку `collectionName` предваряющуюся двоеточием (вы увидите это позже в маршрутах):

```
app.param('collectionName',
  function(req, res, next, collectionName) {
    req.collection = db.collection(collectionName)
    return next()
  }
)
```

Просто, для удобства пользователя, создадим корневой маршрут с сообщением:

```
app.get('/', function(req, res) {
  res.send('please select a collection, e.g., /collections/messages')
})
```

Теперь начинается настоящая работа. Вот как мы получаем список элементов, отсортированный по `_id` и который имеет лимит 10:

[77] http://expressjs.com/api.html#app.use
[78] http://expressjs.com/api.html#middleware
[79] https://github.com/senchalabs/connect
[80] *https://www.mongohq.com/home*

```
app.get('/collections/:collectionName',
  function(req, res) {
    req.collection
      .find({},
        {limit:10, sort: [['_id',-1]]}
      ).toArray(function(e, results){
        if (e) return next(e)
        res.send(results)
      }
    )
  }
)
```

Вы заметили строку `:collectionName` в параметре URL-шаблона? Этот и предыдущий middleware `app.param()` это то, что дает нам объект `req.collection`, который указывает на указанную коллекцию в нашей базе данных.

Объект создания конечной точки немного легче понять, так как мы просто передаем всю полезную нагрузку в MongoDB (метод, так же известный как free JSON REST API):

```
app.post('/collections/:collectionName', function(req, res) {
  req.collection.insert(req.body, {}, function(e, results){
    if (e) return next(e)
    res.send(results)
  })
})
```

Отдельные функции поисковых объектов быстрее, чем `find()`, но они используют другой интерфейс (они возвращают объект напрямую, вместо курсора), поэтому, пожалуйста, будьте осторожны. Кроме того, мы извлекаем ID из части пути с `:id` с помощью `req.params.id` — магии Express.js:

```
app.get('/collections/:collectionName/:id', function(req, res) {
  req.collection.findOne({_id: req.collection.id(req.params.id)},
    function(e, result){
      if (e) return next(e)
      res.send(result)
    }
  )
})
```

Обработчик запросов PUT становится более интересным, потому что `update()` не возвращает дополненный объект, вместо этого он возвращает нам количество затронутых объектов.

Также `{$set:req.body}` является специальным MongoDB-оператором (операторы, как правило, начинаются со знака доллара), устанавливающий значения.

Второй параметр `{safe:true, multi:false}` — это объект с опциями, который говорит MongoDB ждать выполнения перед запуском коллбэка и обрабатывать только один (первый) элемент.

```
app.put('/collections/:collectionName/:id', function(req, res) {
  req.collection.update({_id: req.collection.id(req.params.id)},
    {$set:req.body},
    {safe:true, multi:false},
    function(e, result){
      if (e) return next(e)
      res.send((result===1)?{msg:'success'}:{msg:'error'})
    }
  )
})
```

И, наконец, метод DELETE, который также выводит настраиваемое JSON-сообщение:

```
app.del('/collections/:collectionName/:id', function(req, res) {
  req.collection.remove({_id: req.collection.id(req.params.id)},
    function(e, result){
      if (e) return next(e)
      res.send((result===1)?{msg:'success'}:{msg:'error'})
    }
  )
})
```

Примечание: *`delete` — это оператор в JavaScript, поэтому Express.js использует `app.del` вместо этого.*

Последняя строка, которая собственно запускает сервер на порте 3000 в данном случае:

```
app.listen(3000)
```

На всякий случай, если что-то не работает, вот полный код файла **express.js**:

```
var express = require('express')
  , mongoskin = require('mongoskin')

var app = express()
app.use(express.bodyParser())

var db = mongoskin.db('localhost:27017/test', {safe:true});

app.param('collectionName',
  function(req, res, next, collectionName){
    req.collection = db.collection(collectionName)
    return next()
```

```
  }
)
app.get('/', function(req, res) {
  res.send('please select a collection, '
    + 'e.g., /collections/messages')
})
app.get('/collections/:collectionName', function(req, res) {
  req.collection.find({},{limit:10, sort: [['_id',-1]]})
    .toArray(function(e, results){
      if (e) return next(e)
      res.send(results)
    }
  )
})

app.post('/collections/:collectionName', function(req, res) {
  req.collection.insert(req.body, {}, function(e, results){
    if (e) return next(e)
    res.send(results)
  })
})
app.get('/collections/:collectionName/:id', function(req, res) {
  req.collection.findOne({_id: req.collection.id(req.params.id)},
    function(e, result){
      if (e) return next(e)
      res.send(result)
    }
  )
})
app.put('/collections/:collectionName/:id', function(req, res) {
  req.collection.update({_id: req.collection.id(req.params.id)},
    {$set:req.body},
    {safe:true, multi:false},
    function(e, result){
      if (e) return next(e)
      res.send((result===1)?{msg:'success'}:{msg:'error'})
    }
  )
})
app.del('/collections/:collectionName/:id', function(req, res) {
  req.collection.remove({_id: req.collection.id(req.params.id)},
    function(e, result){
      if (e) return next(e)
      res.send((result===1)?{msg:'success'}:{msg:'error'})
    }
```

```
58    )
59  })
60
61  app.listen(3000)
```

Выйдите из редактора и запустите в терминале:

```
1  $ node express.js
```

И в другом окне (не закрывая первый):

```
1  $ mocha express.test.js
```

Если вам действительно не нравится Mocha и/или BDD, CURL всегда рядом. :-)
Например, данные CURL, чтобы сделать POST-запрос:

```
1  $ curl -d "" http://localhost:3000
```

GET-запросы также работают в браузере, например http://localhost:3000/test.

В этом туториале наши тесты больше, чем сам код приложения, поэтому, отказ от test-driven development может быть заманчивым, но, поверьте мне, **хорошая привычка TDD позволит вам сэкономить много часов** в течении любой серьезной разработки, когда сложность приложений, над которыми вы работаете, велика.

8.7.4 Заключение

Библиотеки Express.js и Mongoskin хороши, когда вам нужно создать простой REST API сервер в несколько строк кода. Позже, если возникнет необходимость расширить библиотеки, они также предоставляют возможность настраивать и организовать ваш код.

NoSQL базы данных, такие как MongoDB, хороши в free-REST API, где мы не должны определять схемы и можем выдать любые данные и это будет безопасно.

Полный код файлов тестов и приложения: https://gist.github.com/azat-co/6075685.

Если вы хотите узнать больше о Express.js и других JavaScript-библиотеках, обратитесь к Intro to Express.js tutorials[81].

Примечание: *в этом примере я не использую точку с запятой. Точки с запятой в JavaScript совершенно необязательны[82] за исключением двух случаев: в цикле for и перед выражением/объявлением, которое начинается со скобок (напр., Immediately-Invoked Function Expression[83]).

[81] http://webapplog.com/tag/intro-to-express-js/

[82] http://blog.izs.me/post/2353458699/an-open-letter-to-javascript-leaders-regarding

[83] http://en.wikipedia.org/wiki/Immediately-invoked_function_expression

8.8 Node.js MVC: Express.js + Derby “Hello World”-туториал

8.8.1 Node MVC фреймворк

Express.js[84] — популярный node-фреймворк, который использует концепцию middleware, чтобы повысить функциональность приложений. Derby[85] — это новый современный фреймворк Model View Controller (MVC[86]), который предназначен для использования с Express[87] в качестве его middleware. Derby также поставляется с поддержкой Racer[88], движком синхронизации данных, Handlebars[89], в качестве шаблонизатора, и многими другими возможностями[90]

8.8.2 Установка Derby

Давайте напишем базовую архитектуру Derby-приложения без использования скаффолдинга. Обычно генераторы проектов сбивают с толку, когда люди просто начинают изучать новый фреймворк. Это туториал по созданию простого приложения “Hello World”, который иллюстрирует скелет Derby и демонстрирует живые шаблоны с вебсокетами.

Конечно нам нужен Node.js[91] и NPM[92], которые можно скачать тут nodejs.org[93]. Для установки Derby глобально, запустите:

```
1 $ npm install -g derby
```

Для проверки установки:

```
1 $ derby -V
```

Моя версия 0.3.15, по состоянию на апрель 2013 года. У нас все готово, чтобы перейти к созданию нашего первого приложения!

8.8.3 Структура файлов

Это структура папки проекта:

[84] http://expressjs.com
[85] http://derbyjs.com
[86] https://ru.wikipedia.org/wiki/Model-View-Controller
[87] http://expressjs.com
[88] https://github.com/codeparty/racer
[89] https://github.com/wycats/handlebars.js/
[90] http://derbyjs.com/#features
[91] http://nodejs.org
[92] http://npmjs.org
[93] http://nodejs.org

```
project/
  -package.json
  -index.js
  -derby-app.js
  views/
    derby-app.html
  styles/
    derby-app.less
```

8.8.4 Зависимости

Давайте включим зависимости и другие основные сведения в файл **package.json**:

```
 {
  "name": "DerbyTutorial",
  "description": "",
  "version": "0.0.0",
  "main": "./server.js",
  "dependencies": {
    "derby": "*",
    "express": "3.x"
  },
  "private": true
}
```

Теперь мы можем запустить команду `npm install`, которая скачает наши зависимости в папку **node_modules**.

8.8.5 Представления

Представления должны быть в папке **views** и они должны находится либо в файле **index.html** и в папке, которая имеет то же имя, что и JS-файл derby-приложения, т.е., `views/derby-app/index.html`, либо быть внутри файла, который имеет то же имя, что и JS-файл derby-приложения, т.е., **derby-app.html**.

В этом примере приложения “Hello World” мы будем использовать шаблон `<Body:>` и переменную `{message}`. Derby использует mustach[94]-handlebars-подобный синтаксис для реактивной привязки. **index.html** выглядит так:

```
<Body:>
  <input value="{message}"><h1>{message}</h1>
```

То же самое и с файлами Stylus/LESS. В нашем примере index.css имеет только одну строку:

[94] http://mustache.github.io/

```
h1 {
  color: blue;
}
```

Чтобы узнать больше об этих удивительных препроцессорах CSS, почитайте документацию на Stylus[95] и LESS[96].

8.8.6 Главный сервер

index.js — это наш основной файл сервера и мы начинаем его с включения зависимостей с помощью функции `require()`:

```
var http = require('http'),
  express = require('express'),
  derby = require('derby'),
  derbyApp = require('./derby-app');
```

Последняя строка — это наш файл derby-приложения **derby-app.js**.

Сейчас мы создаем Express.js-приложение (v3.x имеет существенные отличия от 2.x) и HTTP-сервер:

```
var expressApp = new express(),
  server = http.createServer(expressApp);
```

Derby[97] использует библиотеку синхронизации данных Racer[98], которую мы создаем так:

```
var store = derby.createStore({
  listen: server
});
```

Для выборки данных с back-end для front-end, мы создаем экземпляр объекта модели:

```
var model = store.createModel();
```

Самое главное, мы должны передать модели и маршруты в качестве middleware в Express.js-приложение. Нам нужно указать общую папку для socket.io, чтобы все работало должным образом.

[95] http://learnboost.github.io/stylus/
[96] http://lesscss.org/
[97] http://derbyjs.com
[98] https://github.com/codeparty/racer

```
expressApp.
  use(store.modelMiddleware()).
  use(express.static(__dirname + '/public')).
  use(derbyApp.router()).
  use(expressApp.router);
```

Теперь мы можем запустить сервер на порту 3001 (или любом другом):

```
server.listen(3001, function(){
  model.set('message', 'Hello World!');
});
```

Полный код файла **index.js**:

```
var http = require('http'),
  express = require('express'),
  derby = require('derby'),
  derbyApp = require('./derby-app');

var expressApp = new express(),
  server = http.createServer(expressApp);

var store = derby.createStore({
  listen: server
});

var model = store.createModel();

expressApp.
  use(store.modelMiddleware()).
  use(express.static(__dirname + '/public')).
  use(derbyApp.router()).
  use(expressApp.router);

server.listen(3001, function(){
  model.set('message', 'Hello World!');
});
```

8.8.7 Derby-приложение

Наконец файл Derby-приложения, который содержит код для обоих front-end и back-end. Чисто front-end код находится внутри коллбэка `app.ready()`. Для начала, давайте затребуем и создадим приложение. Derby использует необычную конструкцию (не те же знакомые старые добрые `module.exports = app`):

```
var derby = require('derby'),
  app = derby.createApp(module);
```

Чтобы заставить работать магию socket.io, мы должны подписать атрибуты модели на их визуальное представление, иными словами, выполнить привязку данных и представления данных. Мы можем сделать это в корневом маршруте (сокращение `/`, a.k.a. root):

```
app.get('/', function(page, model, params) {
  model.subscribe('message', function() {
    page.render();
  });
});
```

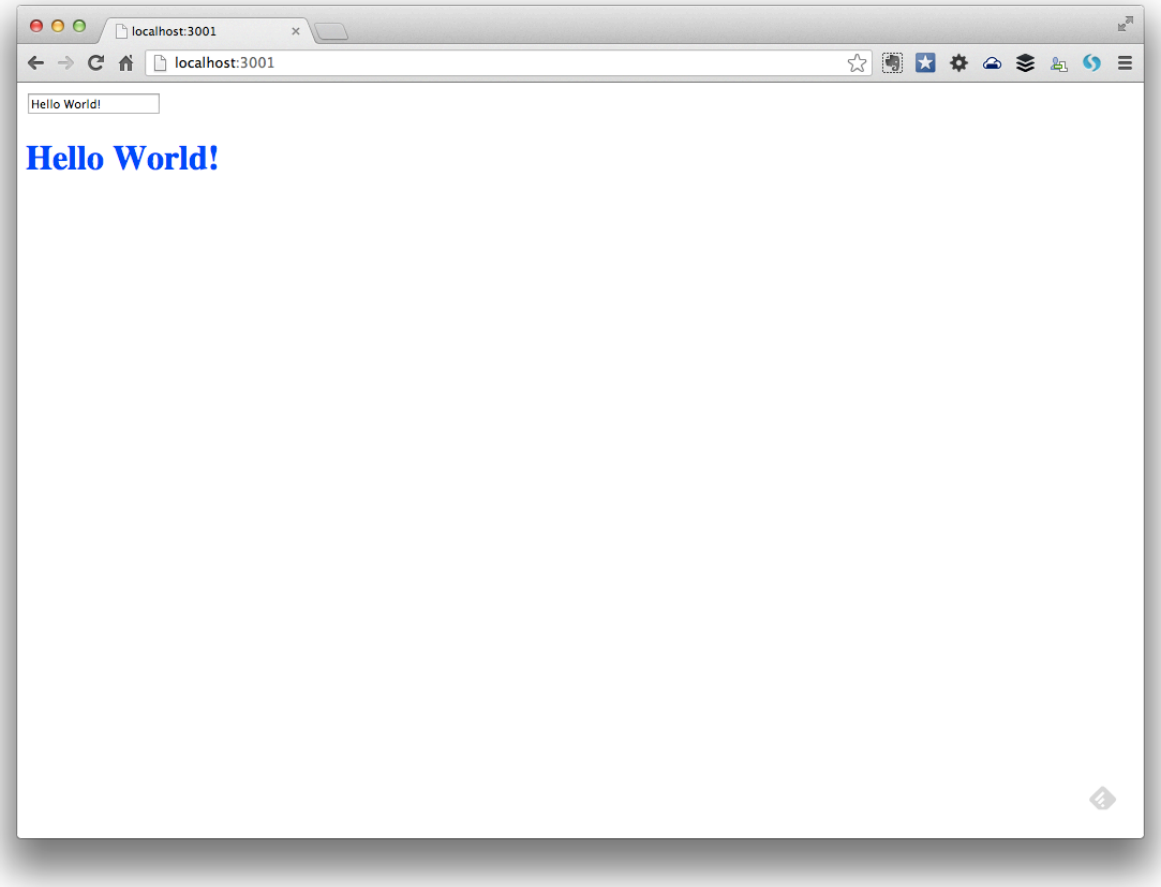
Полный код файла **derby-app.js**:

```
var derby = require('derby'),
  app = derby.createApp(module);

app.get('/', function(page, model, params) {
  model.subscribe('message', function() {
    page.render();
  });
});
```

8.8.8 Запуск приложения “Hello World”

Теперь все должно быть готово для запуска нашего сервера. Выполните `node .` или `node index.js` и откройте в браузере localhost:3001[99]. Вы должны увидеть что-то вроде этого:

[99] http://localhost:3001



Derby + Express.js приложение “Hello World”

8.8.9 Передача значений на Back-End

Конечно, статических данные не так много, поэтому мы можем немного изменить наше приложение, чтобы заставить back-end и front-end общаться друг с другом.

В серверном файле **index.js** добавьте `store.afterDb`, чтобы слушать события `set` на атрибуте **message**:

```
server.listen(3001, function(){
  model.set('message', 'Hello World!');
  store.afterDb('set','message', function(txn, doc, prevDoc, done){
    console.log(txn)
    done();
  });
});
```

Полный код **index.js** после модификаций:

```
var http = require('http'),
  express = require('express'),
  derby = require('derby'),
  derbyApp = require('./derby-app');

var expressApp = new express(),
  server = http.createServer(expressApp);

var store = derby.createStore({
  listen: server
});

var model = store.createModel();

expressApp.
  use(store.modelMiddleware()).
  use(express.static(__dirname + '/public')).
  use(derbyApp.router()).
  use(expressApp.router);

server.listen(3001, function(){
  model.set('message', 'Hello World!');
  store.afterDb('set','message', function(txn, doc, prevDoc, done){
    console.log(txn)
    done();
  });
});
```

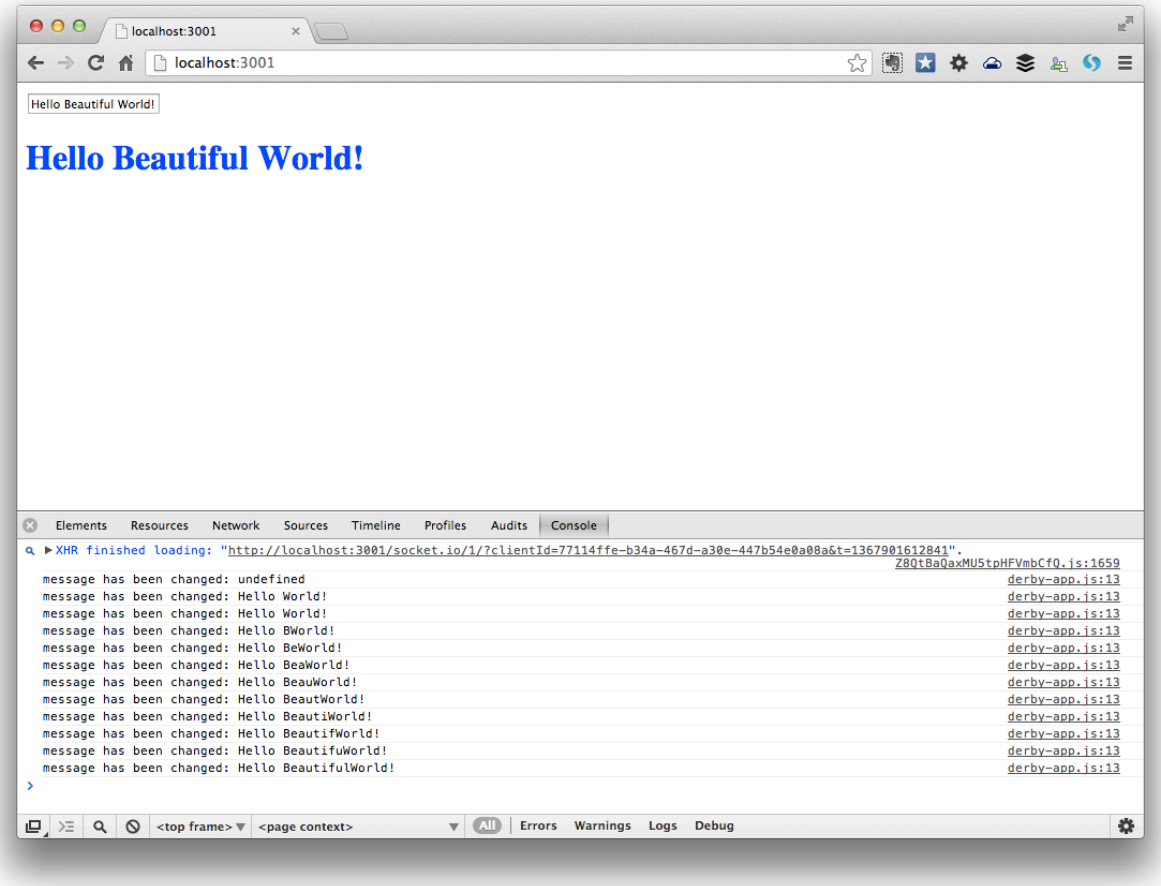
В файле Derby-приложения **derby-app.js** добавьте `model.on()` к `app.ready()`:

```
  app.ready(function(model){
            model.on('set', 'message',function(path, object){
            console.log('message has been changed: '+ object);
          })
  });
```

Полностью файл **derby-app.js** после модификаций:

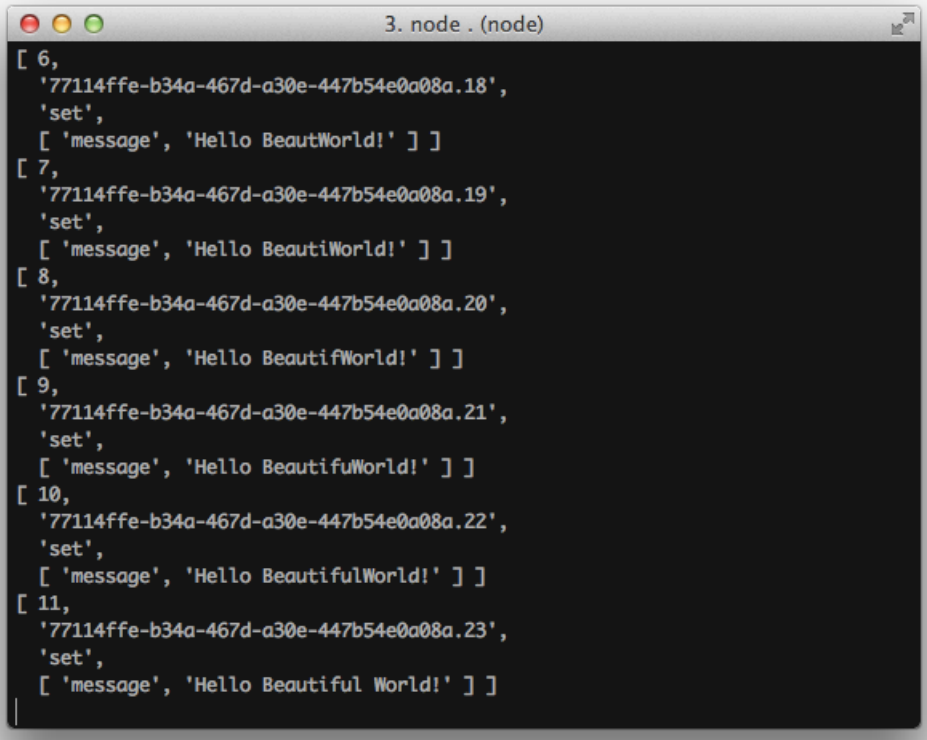
```
var derby = require('derby'),
  app = derby.createApp(module);

app.get('/', function(page, model, params) {
  model.subscribe('message', function() {
    page.render();
  })
});

app.ready(function(model) {
  model.on('set', 'message', function(path, object) {
    console.log('message has been changed: ' + object);
  })
});
```

Сейчас мы посмотрим логи в окне терминала и в консоли браузерного Developer Tools. Конечный результат в браузере должен выглядеть так:



Приложение “Hello World”: логи в консоли браузера

И вот так в терминале:



Приложение “Hello World”: логи в консоли терминала

За дополнительной магией обратитесь к Racer’s db property[100]. С его помощью вы сможете настроить автоматическую синхронизацию между представлениями и базой данных!

Полный код всех файлов этого Express.js + Derby приложения “Hello World” доступен как gist gist.github.com/azat-co/5530311[101].

[100] http://derbyjs.com/#persistence

[101] https://gist.github.com/azat-co/5530311

Заключение и что почитать далее

Резюме: заключение книги, списки блогов, статей, электронных книг, книг и других ресурсов о JavaScript.

Заключение

Мы надеемся, что вам понравилась эта книга. Предполагалось, что она будет содержать меньше теории и больше практики. Предполагалось дать вам обзор нескольких технологий, фреймворков и методов, используемых в современном agile web development. “Быстрое прототипирование с JS” затронула такие темы, как:

- jQuery
- AJAX
- CSS and LESS
- JSON and BSON
- Twitter Bootstrap
- Node.js
- MongoDB
- Parse.com
- Agile methodologies
- Git
- Heroku, MongoHQ and Windows Azure
- REST API
- Backbone.js
- AMD and Require.js
- Express.js
- Monk
- Derby

Если вам нужны углубленные знания или ссылки, они, как правило, доступны в один клик или в поиске Google.

Практический аспект включал в себя создание нескольких версий приложения Chat:

- jQuery + Parse.com JS REST API

- Backbone и Parse.com JS SDK
- Backbone и Node.js
- Backbone и Node.js + MongoDB

Приложение Chat имеет все признаки типичного web/mobile приложения: извлечение данных, их отображение, отправка новых данных. Другие примеры включают в себя:

- jQuery + Twitter RESP API “Tweet Analyzer”
- Parse.com “Save John”
- Node.js “Hello World”
- MongoDB “Print Collections”
- Derby + Express “Hello World”
- Backbone.js “Hello World”
- Backbone.js “Apple Database”
- Monk + Expres.js “REST API Server”

Пожалуйста, отправьте GitHub issue, если у вас есть отзывы, замечания, предложения или вы нашли опечатки, баги, ошибки или другие исправления: https://github.com/azat-co/rpjs/issues.

По вопросам русского перевода пишите на gartod@gmail.com

Другие способы связи: @azat_co[102], http://webapplog.com, http://azat.co.

В случае, если вам понравился Node.js и вы хотите узнать больше о создании боевых веб-сервисов с Express.js (де-факто стандарт для Node.js веб-приложений), тогда ознакомьтесь с моей новой книгой Express.js Guide: The Most Popular Node.js Framework Manual[103].

Что почитать далее

Вот список ресурсов, курсов, книг и блогов для дальнейшего чтения.

JavaScript-ресурсы и бесплатные электронные книги

- Oh My JS[104]: подборка лучших статей о JavaScript
- JavaScript For Cats[105]: введение для начинающих программистов
- Eloquent JavaScript[106]: современное введение в программирование
- Superhero.js[107]: полная коллекция JS-ресурсов
- Руководство по JavaScript[108] от Mozilla Developer Network
- Справочник по JavaScript 1.5[109] от Mozilla Developer Network

[102] http://twitter.com/azat_co

[103] http://expressjsguide.com

[104] https://leanpub.com/ohmyjs/read

[105] http://jsforcats.com/

[106] http://eloquentjavascript.net/

[107] http://superherojs.com/

[108] https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide_ru

[109] https://developer.mozilla.org/ru/docs/Web/JavaScript/%D0%A1%D0%BF%D1%80%D0%B0%D0%B2%D0%BE%D1%87%D0%BD%D0%B8%D0%BA_%D0%BF%D0%BE_JavaScript_1.5

- Why Use Closure[110]: практическое использование замыканий в событийном программировании
- Prototypal Inheritance[111]: объекты с наследуемыми и локальными свойствами
- Control Flow in Node[112]: параллельные vs последовательные потоки
- Truthy and Falsey Values[113]
- How to Write Asynchronous Code[114]
- Smooth CoffeeScript[115]: бесплатная интерактивная HTML5-книга и коллекция справок и других вкусностей
- Developing Backbone.js Applications[116]: бесплатный ранний релиз книги Эдди Османи и O'Reilly
- Step by step from jQuery to Backbone[117]
- Open Web Platform Daily Digest[118]: JS daily digest
- DISTILLED HYPE[119]: JS блог/новости

Книги по JavaScript

- JavaScript: The Good Parts[120]
- JavaScript: The Definitive Guide[121]
- Secrets of the JavaScript Ninja[122]
- Pro JavaScript Techniques[123]

Node.js-ресурсы и бесплатные электронные книги

- Node.js для начинающих[124]
- Felix's Node.js Beginners Guide[125]
- Felix's Node.js Style Guide[126]
- Felix's Node.js Convincing the boss guide[127]

[110] http://howtonode.org/why-use-closure
[111] http://howtonode.org/prototypical-inheritance
[112] http://howtonode.org/control-flow
[113] http://docs.nodejitsu.com/articles/javascript-conventions/what-are-truthy-and-falsy-values
[114] http://docs.nodejitsu.com/articles/getting-started/control-flow/how-to-write-asynchronous-code
[115] http://autotelicum.github.com/Smooth-CoffeeScript/
[116] http://addyosmani.github.com/backbone-fundamentals/
[117] https://github.com/kjbekkelund/writings/blob/master/published/understanding-backbone.md
[118] http://daily.w3viewer.com/
[119] http://distilledhype.com/
[120] http://shop.oreilly.com/product/9780596517748.do
[121] http://www.amazon.com/dp/0596101996/?tag=stackoverfl08-20
[122] http://www.manning.com/resig/
[123] http://www.amazon.com/dp/1590597273/?tag=stackoverfl08-20
[124] http://nodebeginner.ru/
[125] http://nodeguide.com/beginner.html
[126] http://nodeguide.com/style.html
[127] http://nodeguide.com/convincing_the_boss.html

- Introduction to NPM[128]
- NPM Cheatsheet[129]
- Interactive Package.json Cheatsheet[130]
- Official Node.js Documentation[131]
- Node Guide[132]
- Node Tuts[133]
- What Is Node?[134]: бесплатная Kindle версия
- Mastering Node.js[135]: электронная книга по node с открытым кодом
- Mixu's Node book[136]: книга об использовании Node.js
- Learn Node.js Completely and with Confidence[137]: гайд по изучению JavaScript за 2 недели
- How to Node[138]: дзен кодинга на node.js

Книги по Node.js

- The Node Beginner Book[139]
- Hands-on Node.js[140]
- Backbone Tutorials[141]
- Smashing Node.js[142]
- The Node Beginner Book[143]
- Hands-on Node.js[144]
- Node: Up and Running[145]
- Node.js in Action[146]
- Node: Up and Running[147]: масштабируемый серверный код с JavaScript
- Node Web Development[148]: практическое введение в Node
- Node Cookbook[149]

[128] http://howtonode.org/introduction-to-npm
[129] http://blog.nodejitsu.com/npm-cheatsheet
[130] http://package.json.nodejitsu.com/
[131] http://nodejs.org/api/
[132] http://nodeguide.com/
[133] http://nodetuts.com/
[134] http://www.amazon.com/What-Is-Node-ebook/dp/B005ISQ7JC
[135] http://visionmedia.github.com/masteringnode/
[136] http://book.mixu.net/
[137] http://javascriptissexy.com/learn-node-js-completely-and-with-confidence/
[138] http://howtonode.org/
[139] https://leanpub.com/nodebeginner
[140] https://leanpub.com/hands-on-nodejs
[141] https://leanpub.com/backbonetutorials
[142] http://www.amazon.com/Smashing-Node-js-JavaScript-Everywhere-Magazine/dp/1119962595/
[143] http://www.nodebeginner.org/
[144] http://nodetuts.com/handson-nodejs-book.html
[145] http://shop.oreilly.com/product/0636920015956.do
[146] http://www.manning.com/cantelon/
[147] http://www.amazon.com/Node-Running-Scalable-Server-Side-JavaScript/dp/1449398588
[148] http://www.amazon.com/Node-Web-Development-David-Herron/dp/184951514X
[149] http://www.amazon.com/Node-Cookbook-David-Mark-Clements/dp/1849517185/

Интерактивные онлайн-уроки и курсы

- Cody Academy[150]: интерактивные курсы программирования
- Programr[151]
- LearnStreet[152]
- Treehouse[153]
- lynda.com[154]: курсы по программному обеспечению, творчеству и бизнесу
- Udacity[155]: массивные открытые онлайн-курсы
- Coursera[156]

Книги и блоги по стартапам

- Hackers & Painters[157]
- The Lean Startup[158]
- The Startup Owner's Manual[159]
- The Entrepreneur's Guide to Customer Development[160]
- Venture Hacks[161]
- WebAppLog[162]

[150] http://www.codecademy.com/
[151] http://www.programr.com/
[152] http://www.learnstreet.com/
[153] http://teamtreehouse.com/
[154] http://www.lynda.com/
[155] https://www.udacity.com/
[156] https://www.coursera.org/
[157] http://www.amazon.com/Hackers-Painters-Big-Ideas-Computer/dp/1449389554
[158] http://theleanstartup.com/book
[159] http://www.amazon.com/Startup-Owners-Manual-Step-Step/dp/0984999302
[160] http://www.amazon.com/The-Entrepreneurs-Guide-Customer-Development/dp/0982743602/
[161] http://venturehacks.com/
[162] http://webapplog.com