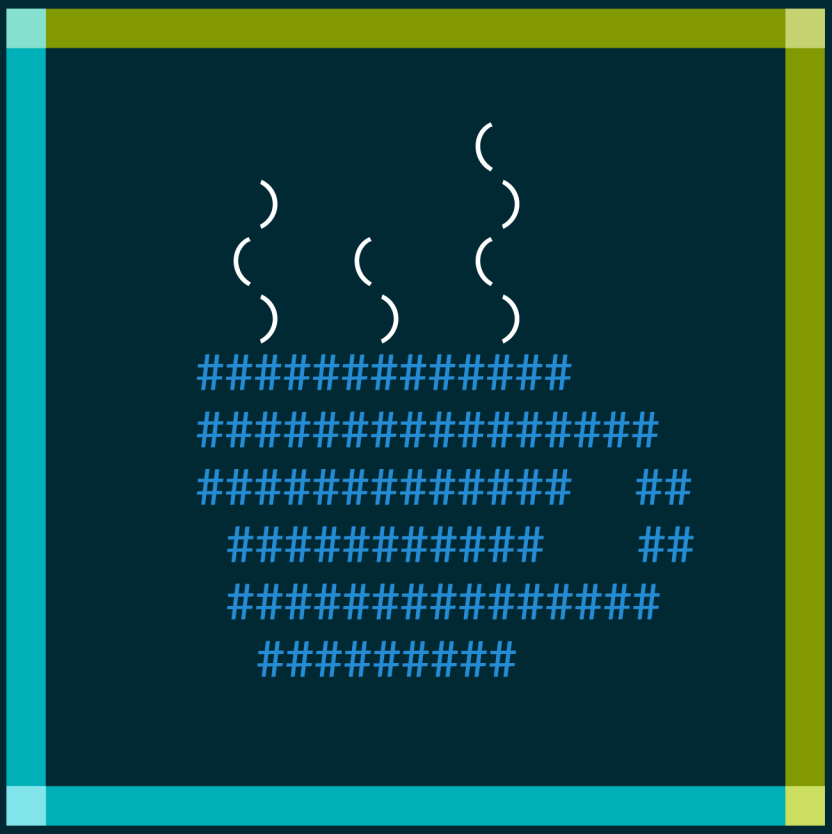
Start learning Backbone.js, Node.js and MongoDB

[AZAT MARDANOV]



RAPID PROTOTYPING WITH JS

AGILE JAVASCRIPT DEVELOPMENT

Rapid Prototyping with JS

Agile JavaScript Development

Azat Mardanov

This book is for sale at http://leanpub.com/rapid-prototyping-with-js

This version was published on 2013-11-27

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 Mardanov by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I've downloaded Rapid Prototyping with JS — book on JavaScript and Node.js by @azat_co #RPJS @RPJSbook

The suggested hashtag for this book is #RPJS.

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=#RPJS

Also By Azat Mardanov

Oh My JS

Express.js Guide

Contents

What Readers Say

> "Azat's tutorials are crucial to the development of Sidepon.com[1] interactive UX and the success of getting us featured on TheNextWeb.com[2] and reached profitability." — Kenson Goo (Sidepon.com[3])

> "I had a lot of fun reading this book and following its examples! It showcases and helps you discover a huge variety of technologies that everyone should consider using in their own projects." — Chema Balsas

Rapid Prototyping with JS is being successfully used at StartupMonthly[4] as a training[5] manual. Here are some of our trainees' testimonials:

> "Thanks a lot to all and special thanks to Azat and Yuri. I enjoyed it a lot and felt motivated to work hard to know these technologies." — Shelly Arora

> "Thanks for putting this workshop together this weekend… what we did with Bootstrap + Parse was really quick & awesome." — Mariya Yao

> "Thanks Yuri and all of you folks. It was a great session - very educative, and it certainly helped me brush up on my Javascript skills. Look forward to seeing/working with you in the future." — 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

Rapid Prototyping with JS on the Internet

Let's be Friends on the Internet

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

Other Ways to Reach Us

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

Share on Twitter

> *"I'm reading Rapid Prototyping with JS — book on agile development with JavaScript and Node.js by @azat_co #RPJS @RPJSbook"* — http://clicktotweet.com/biWsd

---

[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

Acknowledgment

I'm grateful to my copy editor David Moadel and technical editor Alexander Vlasyuk. I'm also grateful to the students of (Hack Reactor[15], Marakana[16], pariSOMA[17] and General Assembly[18]) where I taught and used Rapid Prototyping with JS (or its parts) as a training material.

I would like to thank the team of StartupMonthly[19]: Yuri and Vadim, for helping me with constructive feedback on the Rapid Prototyping with JS training[20] and its manual.

In addition, many gratitudes to my designer friends Ben, Ivan and Natalie who helped me with the cover design feedback.

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

Introduction

Summary: reasons behind rapid prototyping in general and writing of this book; answers to questions what to expect and what not, what are prerequisites; suggestions on how to use the book and examples; explanation of book's notation format.

"Get out of the building." — Steve Blank[21]

Rapid Prototyping with JS is a hands-on book which introduces you to rapid software prototyping using the latest cutting-edge web and mobile technologies including Node.js[22], MongoDB[23], Twitter Bootstrap[24], LESS[25], jQuery[26], Parse.com[27], Heroku[28] and others.

Why RPJS?

This book was borne out of frustration. I have been in software engineering for many years, and when I started learning Node.js and Backbone.js, I learned the hard way that their official documentation and the Internet lack in quick start guides and examples. Needless to say, it was virtually impossible to find all of the tutorials for JS-related modern technologies in one place.

The best way to *learn* is to *do*, right? Therefore, I've used the approach of small simple examples, i.e., quick start guides, to expose myself to the new cool tech. After I was done with the basic apps, I needed some references and organization. I started to write this manual mostly for myself, so I can understand the concepts better and refer to the samples later. Then StartupMonthly[29] and I taught a few 2-day intensive classes on the same subject — helping experienced developers to jump-start their careers with agile JavaScript development. The manual we used was updated and iterated many times based on the feedback received. The end result is this book.

What to Expect

A typical reader of RPJS should expect a collection of quick start guides, tutorials and suggestions (e.g., Git workflow). There is a lot of coding and not much theory. All the theory we cover is directly related to some of

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

the practical aspects, and essential for better understanding of technologies and specific approaches in dealing with them, e.g., JSONP and cross-domain calls.

In addition to coding examples, the book covers virtually all setup and deployment step-by-step.

You'll learn on the examples of Chat web/mobile applications starting with front-end components. There are a few versions of these applications, but by the end we'll put front-end and back-end together and deploy to the production environment. The Chat application contains all of the necessary components typical for a basic web app, and will give you enough confidence to continue developing on your own, apply for a job/promotion or build a startup!

Who This Book is For

The book is designed for advanced-beginner and intermediate-level web and mobile developers: somebody who has been (or still is) an expert in other languages like Ruby on Rails, PHP, Perl, Python or/and Java. The type of a developer who wants to learn more about JavaScript and Node.js related techniques for building web and mobile application prototypes *fast*. Our target user doesn't have time to dig through voluminous (or tiny, at the other extreme) official documentation. The goal of *Rapid Prototyping with JS* is not to make an expert out of a reader, but to help him/her to start building apps as soon as possible.

Rapid Prototyping with JS: Agile JavaScript Development, as you can tell from the name, is about taking your idea to a functional prototype in the form of a web or a mobile application as fast as possible. This thinking adheres to the Lean Startup[30] methodology; therefore, this book would be more valuable to startup founders, but big companies' employees might also find it useful, especially if they plan to add new skills to their resumes.

What This Book is Not

Rapid Prototyping with JS is **neither** a comprehensive book on several frameworks, libraries or technologies (or just a particular one), **nor** a reference for all the tips and tricks of web development. Examples similar to ones in this book might be *publicly* available online.

Even more so, if you're not familiar with fundamental programming concepts like loops, if/else statements, arrays, hashes, object and functions, you won't find them in *Rapid Prototyping with JS*. Additionally, it would be challenging to follow our examples.

Many volumes of great books have been written on fundamental topics — the list of such resources is at the end of the book in the chapter *Further Reading*. The purpose of *Rapid Prototyping with JS* is to give agile tools without replicating theory of programming and computer science.

Prerequisites

We recommend the following things to get the full advantage of the examples and materials covered:

[30] http://theleanstartup.com

- Knowledge of the fundamental programming concepts such as objects, functions, data structures (arrays, hashes), loops (for, while), conditions (if/else, switch)
- Basic web development skills including, but not limited to, HTML and CSS
- Mac OS X or UNIX/Linux systems are highly recommended for this book's examples and for web development in general, although it's still possible to hack your way on a Windows-based system
- Access to the Internet
- 5-20 hours of time
- Some cloud services require users' credit/debit card information even for free accounts

How to Use the Book

For soft-copy (digital version) the book comes in three formats:

1. PDF: suited for printing; opens in Adobe Reader, Mac OS X Preview, iOS apps, and other PDF viewers.
2. ePub: suited for iBook app on iPad and other iOS devices; to copy to devices use iTunes, Dropbox or email to yourself.
3. mobi: suited for Kindles of all generations as well as desktop and mobile Amazon Kindle apps and Amazon Cloud Reader; to copy to devices use Whispernet, USB cable or email to yourself.

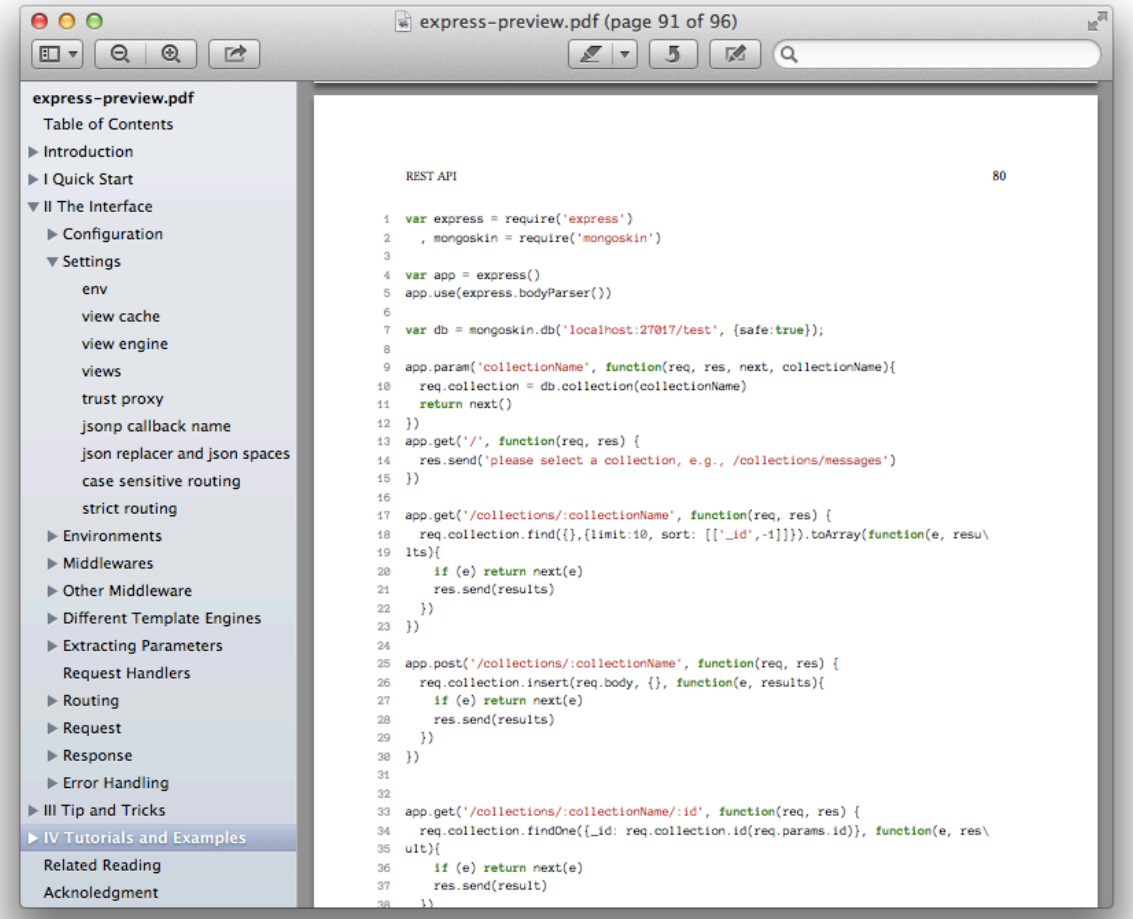
This is a digital version of the book, so most of the links are hidden just like on any other web page, e.g., jQuery[31] instead of http://jquery.com. In the PDF version, URLs are in the footnotes at the bottom of the page. The table of contents has local hyperlinks which allow you to jump to any part or chapter of the book.

There are summaries in the beginning of each chapter describing in a few short sentences what examples and topics the particular chapter covers.

In PDF, EPUB and Mobi versions you could use the **Table of Contents**, which is in the beginning of the book and has internal links, to jump to the most interesting parts or chapters.

For faster navigation between parts, chapters and sections of the book, please use book's navigation pane which is based on the **Table of Contents** (the screenshot is below).

[31] http://jquery.com



The Table of Contents pane in the Mac OS X Preview app.

Examples

All of the source code for examples used in this book is available in the book itself for the most part, as well as in a public GitHub repository github.com/azat-co/rpjs[32]. You can also download files as a ZIP archive[33] or use Git to pull them. More on how to install and use Git will be covered later in the book. The source code files, folder structure and deployment files are supposed to work locally and/or remotely on PaaS solutions, i.e., Windows Azure and Heroku, with minor or no modifications.

Source code which is in the book is technically limited by the platform to the width of about 70 characters. We tried our best to preserve the best JavaScript and HTML formatting styles, but from time to time you might see backslashes (\). There is nothing wrong with the code. Backslashes are line escape characters, and if you

[32] http://github.com/azat-co/rpjs
[33] https://github.com/azat-co/rpjs/archive/master.zip

copy-paste the code into the editor, the example should work just fine. Please note that code in GitHub and in the book might differ in formatting. Also, let us know via email (hi@rpjs.co[34]) if you spot any bugs!

Notation

This is what source code blocks look like:

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

Terminal commands have a similar look but start with dollar sign or $:

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

Inline file names, path/folder names, quotes and special words/names are *italicized*, while command names, e.g., **mongod**, and emphasized words, e.g., **Note**, are **bold**.

Terms

For the purpose of this book, we're using some terms interchangeably, while depending on the context, they might not mean exactly the same thing. For example, function = method = call, attribute = property = member = key, value = variable, object = hash = class, list = array, framework = library = module.

[34] mailto:hi@rpjs.co

About the Author

Azat Mardanov: a software engineer, an author and a yogi.

Azat Mardanov has over 12 years of experience in web, mobile and software development. With a Bachelor's Degree in Informatics and a Master of Science in Information Systems Technology degree, Azat possesses deep academic knowledge as well as extensive practical experience.

Currently, Azat works as a Senior Software Engineer at DocuSign[35], where his team rebuilds 50 million user product (DocuSign web app) using the tech stack of Node.js, Express.js, Backbone.js, CoffeeScript, Jade, Stylus and Redis.

Recently, he worked as an engineer at the curated social media news aggregator website, Storify.com[36] (acquired by LiveFyre[37]) which is used by BBC, NBC, CNN, The White House and others. Storify runs everything on Node.js unlike other companies. It's the maintainer of the open-source library jade-browser[38].

Before that, Azat worked as a CTO/co-founder at Gizmo[39] — an enterprise cloud platform for mobile marketing campaigns, and has undertaken the prestigious 500 Startups[40] business accelerator program.

Prior to this, Azat was developing he developed mission-critical applications for government agencies in Washington, DC, including the National Institutes of Health[41], the National Center for Biotechnology Information[42], and the Federal Deposit Insurance Corporation[43], as well as Lockheed Martin[44].

Azat is a frequent attendee at Bay Area tech meet-ups and hackathons (AngelHack[45] hackathon '12 finalist with team FashionMetric.com[46]).

---

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

In addition, Azat teaches technical classes at General Assembly[47], Hack Reactor[48], pariSOMA[49] and Marakana[50] (acquired by Twitter) to much acclaim.

In his spare time, he writes about technology on his blog: webAppLog.com[51] which is number one[52] in "express.js tutorial" Google search results. Azat is also the author of Express.js Guide[53], Rapid Prototyping with JS[54] and Oh My JS[55]; and the creator of open-source Node.js projects, including ExpressWorks[56], mongoui[57] and HackHall[58].

Let's be Friends on the Internet

- Twitter: @RPJSbook[59] and @azat_co[60]
- Facebook: facebook.com/RapidPrototypingWithJS[61]
- Website: rapidprototypingwithjs.com[62]
- Blog: webapplog.com[63]
- GitHub: github.com/azat-co/rpjs[64]
- Storify: Rapid Prototyping with JS[65]

Other Ways to Reach Us

- Email: hi@rpjs.co[66]
- Google Group: rpjs@googlegroups.com[67] and https://groups.google.com/forum/#!forum/rpjs

Share on Twitter

> "I've finished Rapid Prototyping with JS — book on agile development with JavaScript and Node.js by @azat_co #RPJS @RPJSbook" — http://clicktotweet.com/40dvj

---

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

I Quick Start

1 Basics

Summary: overview of HTML, CSS, and JavaScript syntaxes; brief introduction to Agile methodology; advantages of cloud computing, Node.js and MongoDB; descriptions of HTTP requests/responses, RESTful API concepts.

"I think everyone should learn how to program a computer, because it teaches you how to think. I view computer science as a liberal art, something everyone should learn to do." — Steve Jobs

1.1 Front-End Definitions

1.1.1 Bigger Picture

The bigger picture of web and mobile application development consists of the following steps:

1. User types a URL or follows a link in her browser (a.k.a. client)
2. Browser makes HTTP request to the server
3. Server processes the request, and if there are any parameters in a query string and/or body of the request, it takes them into account
4. Server updates/gets/transforms data in the database
5. Server responds with HTTP response containing data in HTML, JSON or other formats
6. Browser receives HTTP response
7. Browser renders HTTP response to the user in HTML or any other format, e.g., JPEG, XML, JSON

Mobile applications act in the same manner as regular websites, only instead of a browser there is a native app. Other minor differences include: data transfer limitation due to carrier bandwidth, smaller screens, and the more efficient use of the local storage.

There are a few approaches to mobile development, each with its own advantages and disadvantages:

- Native iOS, Android, Blackberry apps build with Objective-C and Java
- Native apps build with JavaScript in Appcelerator[1], or similar tools, and then complied into native Objective-C or Java
- Mobile websites tailored for smaller screens with responsive design, CSS frameworks like Twitter Bootstrap[2] or Foundation[3], regular CSS or different templates

[1] http://www.appcelerator.com/
[2] http://twitter.github.io/bootstrap/
[3] http://foundation.zurb.com/

- HTML5 apps which consist of HTML, CSS and JavaScript, and are usually built with frameworks like Sencha Touch[4], Trigger.io[5], JO[6], and then wrapped into native app with PhoneGap[7]

1.1.2 HyperText Markup Language

A HyperText Markup Language, or HTML, is not a programming language in itself. It is a set of markup tags which describe the content and present it in a structured and formatted way. HTML tags consist of a **tag name** inside of the angle brackets (<>). In most cases, tags surround the content with the end tag having **forward slash** before the tag name.

In this example, each line is an HTML element:

```
<h2>Overview of HTML</h2>
<div>HTML is a ...</div>
<link rel="stylesheet" type="text/css" href="style.css" />
```

An HTML document itself is an element of the *html* tag, and all other elements are children of that *html* tag:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <link rel="stylesheet" type="text/css" href="style.css" />
  </head>
  <body>
    <h2>Overview of HTML</h2>
    <p>HTML is a ...</p>
  </body>
</html>
```

There are different flavors and versions of HTML, e.g., DHTML, XHTML 1.0, XHTML 1.1, XHTML 2, HTML 4, HTML 5. This article does a good job of explaining the differences — Misunderstanding Markup: XHTML 2/HTML 5 Comic Strip[8].

Any HTML element can have attributes. The most important of them are: class, id, style, data-name, onclick, and other event attributes.

class

Class attribute defines a class which is used for styling in CSS or DOM manipulation, e.g.:

[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/

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

id

Id attribute defines an ID which is similar in purpose to element class but has to be unique, e.g.:

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

style

Style attribute defines inline CSS to style an element, e.g.:

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

title

Title attribute specifies additional information which is usually presented in tooltips by most browsers, e.g.:

```
<a title="Up-vote the answer">...</a>
```

data-name

Data-name attribute allows for meta data to be stored in DOM, e.g.:

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

onclick

Onclick attribute calls inline JavaScript code when click event happens, e.g.:

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

onmouseover

Onmouseover attribute is similar to `onclick` but for mouse hover events, e.g.:

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

Other HTML element attributes for inline JavaScript code are:

- onfocus: when browser focuses on an element
- onblur: when browser focus leaves an element
- onkeydown: when a user presses a keyboard key
- ondblclick: when a user double-clicks the mouse
- onmousedown: when a user presses a mouse button

- onmouseup: when a user releases a mouse button
- onmouseout: when a user moves mouse out of the element area
- oncontextmenu: when a user brings up a context menu

The full list of such events and a browser compatibility table are presented in Event compatibility tables[9].

We'll use classes extensively with Twitter Bootstrap framework, while the use of inline CSS and JavaScript code is generally a **bad** idea, so we'll try to avoid it. However, it's good to know the names of the JavaScript events because they are used all over the place in jQuery, Backbone.js and of course plain JavaScript. To convert the list of attribute to a list of JS events, just remove the prefixes `on`, e.g., `onclick` attribute means `click` event.

More information is available at Example: Catching a mouse click[10], Wikipedia[11] and w3schools[12].

1.1.3 Cascading Style Sheets

Cascading Style Sheets, or CSS, is a way to format and present content. An HTML document can have an external stylesheet included in it by a **link** tag, as shown in the previous examples, or can have CSS code directly inside of a **style** tag:

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

Each HTML element can have **id** and/or **class** attributes:

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

In CSS we access elements by their *id*, *class*, tag name and in some edge cases by parent-child relationship or element attribute value.

This sets a color of all the paragraphs (**p** tag) to grey (`#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://en.wikipedia.org/wiki/HTML
[12] http://www.w3schools.com/html/html_intro.asp

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

This sets padding of a div element with id main:

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

This sets the font size to 14 pixels for all elements with a class `large`:

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

This hides div which are direct children of body element:

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

This sets the width to 150 pixels for input which name attribute is email:

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

More information is available at Wikipedia[13] and w3schools[14].

CSS3 is an upgrade to CSS which includes new ways of doing things such as rounded corners, borders and gradients, which were possible in regular CSS only with the help of PNG/GIF images and by using other tricks.

For more information refer to CSS3.info[15], w3school[16] and CSS3 vs. CSS comparison article on Smashing[17].

1.1.4 JavaScript

JavaScript was started in 1995 at Netscape as LiveScript. It has the same relationship with Java as a hamster and a ham. :-) These days, JavaScript is used for both client and server-side web, as well as in desktop application development.

Putting JS code into a *script* tag is the easiest way to use JavaScript in an HTML document:

[13] http://en.wikipedia.org/wiki/Cascading_Style_Sheets
[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>
```

Be advised that mixing HTML and JS code is not a good idea, so to separate them we can move the code to an external file, and include it by setting source attribute `src="filename.js"` on **script** tag, e.g., for *app.js* resource:

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

Note

The closing `</script>` tag is mandatory even with an empty element like we have where we include the external source file. **Type** and **language** attributes over the years became optional in modern browsers due to the overwhelming JavaScript dominance.

Other ways to run JavaScript include:

- Inline approach already covered above
- WebKit browser Developer Tools and FireBug consoles
- The interactive Node.js shell

One of the advantages of the JavaScript language is that it's loosely typed. This loose/weak typing, as opposed to strong typing[18] in languages like C and Java, makes JavaScript a better programming language for prototyping. Here are some of the main types of JavaScript objects/classes (there are not classes per se; objects inherit from objects):

Number primitives

Numerical values, e.g.:

```
var num = 1;
```

Number Object

Number[19] object and its methods, e.g.:

[18] http://en.wikipedia.org/wiki/Strong_typing
[19] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Number

```
var numObj = new Number("123"); //Number object
var num = numObj.valueOf(); //number primitive
var numStr = numObj.toString(); //string representation
```

String primitives

Sequences of characters inside of single or double quotes, e.g.:

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

For convenience, JS automatically wraps string primitives with String object methods, but they are not quite the same[20].

String object

String object has a lot of useful methods, like `length`, `match`, etc., for example:

```
var strObj = new String("abcde");//String object
var str = strObj.valueOf(); //string primitive
strObj.match(/ab/);
str.match(/ab/); //both call will work
```

RegExp object

Regular Expressions or RegExps are patterns of characters used in finding matches, replacing, testing of strings:

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

Special Types

When in doubt, you can always call `typeof obj`. Here are some of the special types used in JS:

- NaN
- null
- undefined
- function

Globals

You can call these methods from anywhere in your code, because they are global methods:

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

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

JSON

JSON library allows us to parse and serialize JavaScript objects, e.g.:

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

Array object

Arrays[21] are zero-index-based lists. For example, to create an array:

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

The Array object has a lot of nice methods, like `indexOf`, `slice`, `join`. Make sure that you're familiar with them, because if used correctly, they'll save a lot of time.

Data Object

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

or

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

More on inheritance patterns below.

Boolean primitives and objects

Same as with String and Number, Boolean[22] can be a primitive and an object.

[21] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array
[22] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Boolean

```
var bool1 = true;
var bool2 = false;
var bool1Obj = new Boolean(false);
```

Date object

Date[23] objects allow us to work with dates and time, e.g.:

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

Math object

Mathematical constants and functions[24], e.g.:

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

Browser objects

Gives us access to browser and its properties like URL, e.g.:

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

DOM objects

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

Warning

JavaScript supports numbers only up to 53-bit in size. Check out large numbers' libraries if you need to deal with numbers larger than that.

The full references of JavaScript and DOM objects are available at Mozilla Developer Network[25] and w3school[26].

For JS resources such as ECMA specs, check out this list JavaScript Language Resources[27]. As of this writing, the latest JavaScript specification is ECMA-262 Edition 5.1: PDF[28] and HTML[29].

Another important distinction of JS is that it's a functional and prototypal language. Typical syntax for function declaration looks like this:

[23] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Date
[24] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Math
[25] https://developer.mozilla.org/en-US/docs/JavaScript/Reference
[26] http://www.w3schools.com/jsref/default.asp
[27] https://developer.mozilla.org/en-US/docs/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));
```

Functions in JavaScript are first-class citizens[30] due to the functional programming[31] nature of the language. Therefore, functions can be used as other variables/objects; for example, functions can be passed to other functions as arguments:

```
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'));
```

It's good to know that there are several ways to instantiate an object in JS:

- Classical inheritance[32] pattern
- Pseudo classical inheritance[33] pattern
- Functional inheritance pattern

For further reading on inheritance patterns, check out Inheritance Patterns in JavaScript[34] and Inheritance revisited[35].

More information about browser-run JavaScript is available at Mozilla Developer Network[36], Wikipedia[37] and w3schools[38].

1.2 Agile Methodologies

The Agile software development methodology evolved due to the fact that traditional methods like Waterfall weren't good enough in situations of high unpredictability, i.e., when the solution is unknown[39]. Agile methodology includes Scrum/Sprint, Test-Driven Development, Continuous Deployment, Paired Programming and other practical techniques, many of which were borrowed from Extreme Programming.

[30] http://en.wikipedia.org/wiki/First-class_function
[31] http://en.wikipedia.org/wiki/Functional_programming
[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/en-US/docs/JavaScript/Reference
[37] http://en.wikipedia.org/wiki/JavaScript
[38] http://www.w3schools.com/js/default.asp
[39] http://www.startuplessonslearned.com/2009/03/combining-agile-development-with.html

1.2.1 Scrum

In regard to the management, Agile methodology uses Scrum approach. More about Scrum can be read at:

- Scrum Guide in PDF[40]
- Scrum.org[41]
- Scrum development Wikipedia article[42]

The Scrum methodology is a sequence of short cycles, and each cycle is called **sprint**. One sprint usually lasts from one to two weeks. A typical sprint starts and ends with a sprint planning meeting where new tasks are assigned to team members. New tasks cannot be added to the sprint in progress; they can be added only at the sprint meetings.

An essential part of the Scrum methodology is the daily **scrum** meeting — hence the name. Each scrum is a 5-to-15-minutes-long meeting which is often conducted in the hallways. In scrum meetings, each team member answers three questions:

1. What have you done since yesterday?
2. What are you going to do today?
3. Do you need anything from other team members?

Flexibility makes Agile an improvement over Waterfall methodology, especially in situations of high uncertainty, i.e., in startups.

The advantage of Scrum methodology: effective where it is hard to plan ahead of time, and also in situations where a feedback loop is used as a main decision-making authority.

1.2.2 Test-Driven Development

Test-Driven Development, or TDD, consists of the following steps:

1. Write failing automated test cases for new feature/tasks or enhancement by using assertions that are either true or false.
2. Write code to successfully pass the test cases.
3. Refactor code if needed, and add functionality while keeping the test cases passed.
4. Repeat until all tasks are complete.

Tests can be split into functional and unit testing. The latter is when a system tests individual units, methods and functions with dependencies mocked up, while the former (a.k.a., integration testing) is when a system tests a slice of a functionality, including dependencies.

The advantages of Test-Driven Development:

[40] http://www.scrum.org/storage/scrumguides/Scrum_Guide.pdf
[41] http://www.scrum.org/
[42] http://en.wikipedia.org/wiki/Scrum_(development)

- Fewer bugs/defects
- More efficient codebase
- Confidence that code works and doesn't break the old functionality

1.2.3 Continuous Deployment and Integration

Continuous Deployment, or CD, is the set of techniques to rapidly deliver new features, bug fixes, and enhancements to the customers. CD includes automated testing and automated deployment. By utilizing Continuous Deployment, the manual overheard is decreased and the feedback loop time is minimized. Basically, the faster a developer can get the feedback from the customers, the sooner the product can pivot, which leads to more advantages over the competition. Many startups deploy multiple times in a single day in comparison to the 6-to-12-month release cycle which is still typical for corporations and big companies.

The advantages of the Continuous Deployment approach: decreases feedback loop time and manual labor overhead.

The difference between Continuous Deployment and Continuous Integration is outlined in the post Continuous Delivery vs. Continuous Deployment vs. Continuous Integration - Wait huh?[43]

Some of the most popular solutions for Continuous Integration:

- Jenkins[44]: An extendable open source continuous integration server
- CircleCI[45]: Ship better code, faster
- Travis CI[46]: A hosted continuous integration service for the open source community

1.2.4 Pair Programming

Pair Programming is a technique when two developers work together in one environment. One of the developers is a driver, and the other is an observer. The driver writes code, and the observer assists by watching and making suggestions. Then they switch roles. The driver has a more tactical role of focusing on the current task. In contrast, the observer has a more strategic role, overseeing "the bigger picture" and finding bugs and ways to improve an algorithm.

The advantages of Paired Programming:

- Pairs attributes to shorter and more efficient codebase, and introduces fewer bugs and defects.
- As an added bonus, knowledge is passed along programmers as they work together. However, situations of conflicts between developers are possible, and not uncommon at all.

[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/

1.3 Back-End Definitions

1.3.1 Node.js

Node.js is an open-source event-driven asynchronous I/O technology for building scalable and efficient web servers. Node.js consists of Google's V8 JavaScript engine[47] and is maintained by cloud company Joyent[48].

The purpose and use of Node.js is similar to Twisted[49] for Python and EventMachine[50] for Ruby. The JavaScript implementation of Node was the third one after attempts at using Ruby and C++ programming languages.

Node.js is not in itself a framework like Ruby on Rails; it's more comparable to the pair PHP+Apache. More on Node.js frameworks later in the *Node.js and MongoDB* chapter.

The advantages of using Node.js:

- Developers have high likelihood of familiarity with JavaScript language due to its status as a de facto standard for web and mobile development
- One language for front-end and back-end development speeds up the coding process. A developer's brain doesn't have to switch between different syntaxes. So-called context switch. The learning of methods and classes goes faster.
- With Node.js, you could prototype quickly and go to market to do your customer development and customer acquisition early. This is an important competitive advantage over the other companies, which use less agile technologies, e.g., PHP and MySQL.
- Node.js is built to support real-time applications by utilizing web-sockets.

For more information go to Wikipedia[51], Nodejs.org[52], and articles on ReadWrite[53] and O'Reilly[54].

For the current state of Node.js as of this writing, refer to State of the Node slides by Isaac Z. Schlueter – 2013[55].

1.3.2 NoSQL and MongoDB

MongoDB, from hu**MONGO**us, is a high-performance no-relationship database for huge quantities of data. The NoSQL concept came out when traditional Relational Database Management Systems, or RDBMS, were unable to meet the challenges of huge amounts of data.

The advantages of using MongoDB:

- Scalability: due to a distributed nature, multiple servers and data centers can have redundant data.

[47] http://en.wikipedia.org/wiki/V8_(JavaScript_engine)
[48] http://joyent.com
[49] http://twistedmatrix.com/trac/
[50] http://rubyeventmachine.com/
[51] http://en.wikipedia.org/wiki/Nodejs
[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

- High-performance: MongoDB is very effective for storing and retrieving data, partially owing to the absence of relationships between elements and collections in the database.
- Flexibility: a key-value store is ideal for prototyping because it doesn't require developers to know the schema and there is no need for a fixed data models, or complex migrations.

1.3.3 Cloud Computing

Could computing consists of:

- Infrastructure as s Service (IaaS), e.g., Rackspace, Amazon Web Services
- Platform as a Service (PaaS), e.g., Heroku, Windows Azure
- Backend as a Service (BaaS — newest, coolest kid on the block), e.g., Parse.com, Firebase
- Software as a Service (SaaS), e.g., Google Apps, Salesforce.com

Cloud application platforms provide:

- Scalability, e.g., spawn new instances in a matter of minutes;
- Ease of deployment, i.e., to push to Heroku you can just use `$ git push`
- Pay-as-you-go plans where users add or remove memory and disk space based on demands
- Add-ons for easier installation and configuration of databases, app servers, packages, etc.
- Security and support

PaaS and BaaS are ideal for prototyping, building minimal viable products (MVP) and for early-stage startups in general.

Here is the list of most popular PaaS solutions:

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

1.3.4 HTTP Requests and Responses

Each HTTP Request and Response consists of the following components:

1. Header: information about encoding, length of the body, origin, content type, etc.
2. Body: content, usually parameters or data which is passed to the server or sent back to a client

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

In addition, the HTTP Request contains:

- Method: There are several methods with the most common being `GET`, `POST`, `PUT` and `DELETE`
- URL: host, port, path, e.g., `https://graph.facebook.com/498424660219540`
- Query string: everything after a question mark in the URL (e.g., `?q=rpjs&page=20`)

1.3.5 RESTful API

RESTful (REpresentational State Transfer) API became popular due to the demand in distributed systems whereby each transaction needs to include enough information about the state of the client. In a sense, this standard is stateless because no information about the clients' states is stored on the server, thus making it possible for each request to be served by a different system.

Distinct characteristics of RESTful API:

- Has better scalability support due to the fact that different components can be independently deployed to different servers
- Replaced Simple Object Access Protocol (SOAP) because of the simpler verb and noun structure
- Utilizes HTTP methods: GET, POST, DELETE, PUT, OPTIONS, etc.

Here is an example of simple Create, Read, Update and Delete (CRUD) REST API for Message Collection:

Method	URL	Meaning
GET	/messages.json	Return list of messages in JSON format
PUT	/messages.json	Update/replace all messages and return status/error in JSON
POST	/messages.json	Create new message and return its id in JSON format
GET	/messages/{id}.json	Return message with id {id} in JSON format
PUT	/messages/{id}.json	Update/replace message with id {id}, if {id} message doesn't exists create it
DELETE	/messages/{id}.json	Delete message with id {id}, return status/error in JSON format

REST is not a protocol; it is an architecture in the sense that it's more flexible than SOAP, which is a protocol. Therefore, REST API URLs could look like `/messages/list.html` or `/messages/list.xml` in case we want to support these formats.

PUT and DELETE are idempotent methods[60], which means that if the server receives two or more similar requests, the end result will be the same.

GET is nullipotent and POST is not idempotent and might affect state and cause side-effects.

Further reading on REST API can be found at Wikipedia[61] and A Brief Introduction to REST article[62].

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

2 Setup

Summary: suggestions for the toolset; step-by-step installation of local components; preparation for the use of cloud services.

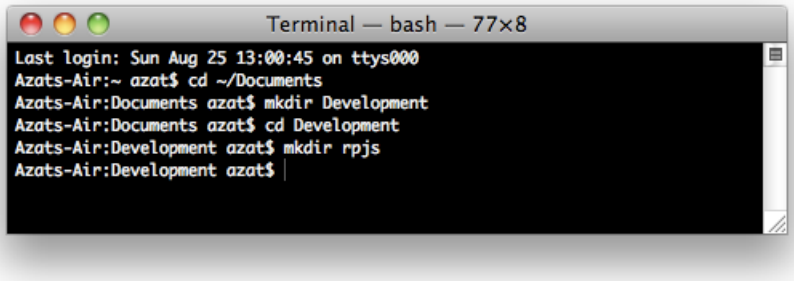
"One of my most productive days was throwing away 1,000 lines of code." - Ken Thompson[1]

2.1 Local Setup

2.1.1 Development Folder

If you don't have a specific development folder for your web development projects, you could create a *Development* folder in the *Documents* folder (path will be *Documents/Development*). To work on the code example, create a *rpjs* folder inside your web development projects folder, e.g., if you create a *rpjs* folder inside of the *Development* folder, the path will be *Documents/Development/rpjs*. You could use the Finder on Mac OS X or the following terminal commands on OS X/Linux systems:

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



Initial development environment setup.

[1] http://en.wikipedia.org/wiki/Ken_Thompson

Tip

To open Mac OS Finder app in the current directory from Terminal, just type and run the `$ open .` command.

To get the list of files and folders, use this UNIX/Linix command:

```
$ ls
```

or to display hidden files and folders, like *.git*:

```
$ ls -lah
```

Another alternative to `$ ls` is `$ ls -alt`. The difference between the `-lah` and the `-alt` options is that the latter sorts items chronologically and the former alphabetically.

Note

You can use the Tab key to autocomplete names of the files and folders.

Later, you could copy examples into the *rpjs* folder as well as create apps in that folder.

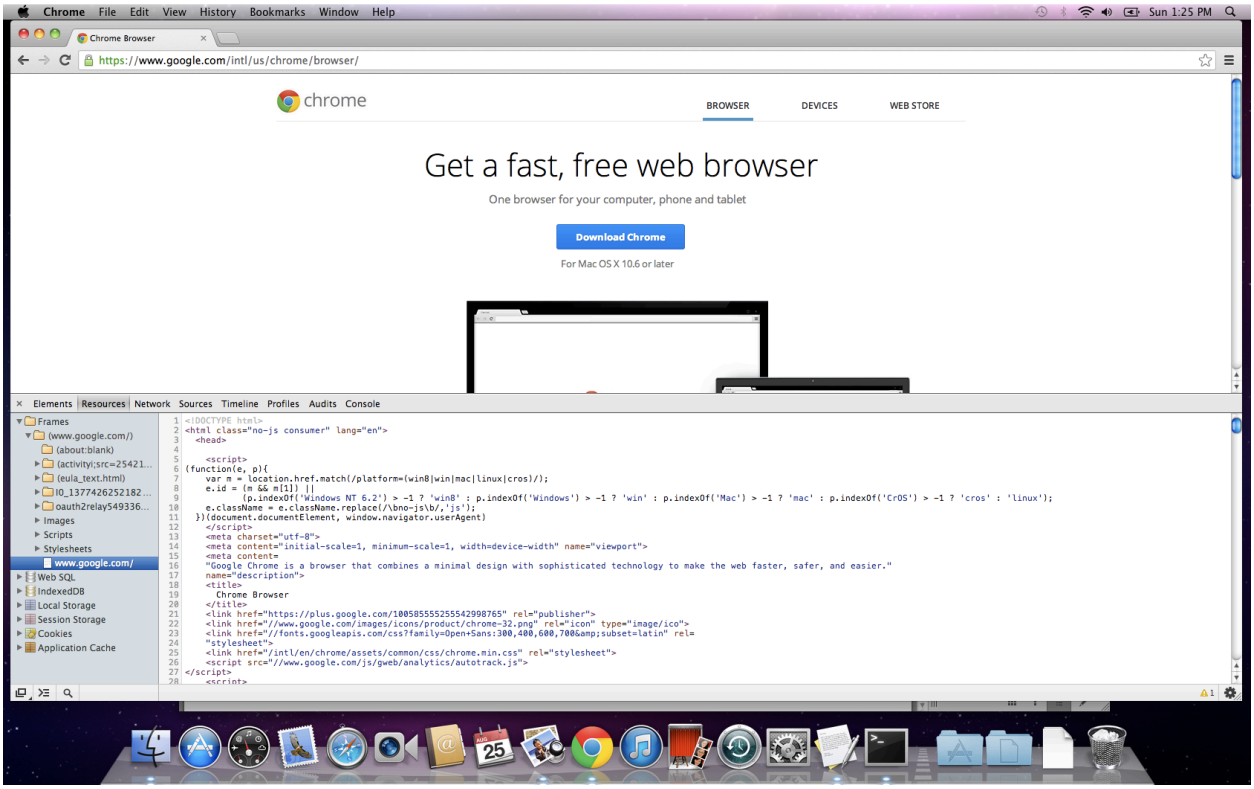
Note

Another useful thing is to have the "New Terminal at Folder" option in Finder on Mac OS X. To enable it, open your "System Preferences" (you could use Command + Space, a.k.a. Spotlight, for it). Find "Keyboard" and click on it. Open "Keyboard Shortcuts" and click on "Services." Check the "New Terminal at Folder" and "New Terminal Tab at Folder" boxes. Close the window (optional).

2.1.2 Browsers

We recommend downloading the latest version of the WebKit[2] or Gecko[3] browser of your choice: Chrome[4], Safari[5] or Firefox[6]. While Chrome and Safari already come with built-in Developer Tools, you'll need the Firebug[7] plug-in for Firefox.

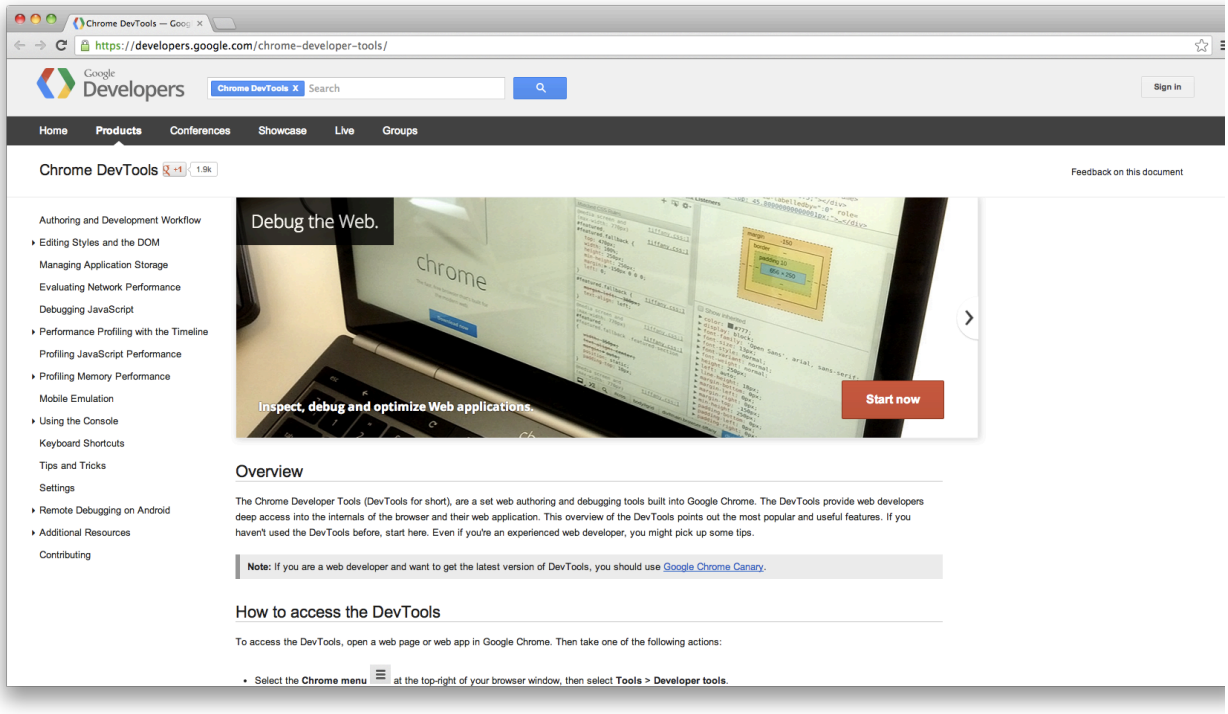
[2] http://en.wikipedia.org/wiki/WebKit
[3] http://en.wikipedia.org/wiki/Gecko_(layout_engine)
[4] http://www.google.com/chrome
[5] http://www.apple.com/safari/
[6] http://www.mozilla.org/en-US/firefox/new/
[7] http://getfirebug.com/



Chrome dev tools.

Firebug and Developer Tools allow developers to do many things like:

- Debug JavaScript
- Manipulate HTML and DOM elements
- Modify CSS on the fly
- Monitor HTTP requests and responses
- Run profiles and inspect heap dumps
- See loaded assets such as images, CSS and JS files



Google tutorials for mastering web deb tools.

Great Chrome DevTools tutorials:

- 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/

Mastering chrome DevTools.

2.1.3 IDEs and Text Editors

One of the best things about JavaScript is that you don't need to compile the code. Because JS lives in and is run in a browser, you can do debugging right there, in a browser! Therefore, we highly recommend a lightweight text editor vs. a full-blown integrated development environment[11], or IDE, but if you are already familiar and comfortable with the IDE of your choice like Eclipse[12], NetBeans[13] or Aptana[14], feel free to stick with it.

[11] http://en.wikipedia.org/wiki/Integrated_development_environment
[12] http://www.eclipse.org/
[13] http://netbeans.org/
[14] http://aptana.com/

Sublime Text code editor home page.

Here is the list of the most popular text editors and IDEs used in web development:

- TextMate[15]: Mac OS X version only, free 30-day trial for v1.5, dubbed *The Missing Editor for Mac OS X*.
- Sublime Text[16]: Mac OS X and Windows versions are available, even better alternative to TextMate, unlimited evaluation period.
- Coda[17]: all-in-one editor with FTP browser and preview, has support for development with/on an iPad.
- Aptana Studio[18]: full-sized IDE with a built-in terminal and many other tools.
- Notepad ++[19]: free Windows-only lightweight text editor with the support of many languages.
- WebStorm IDE[20]: feature-rich IDE which allows for Node.js debugging; it's developed by JetBrains and marketed as *the smartest 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/

WebStorm IDE home page.

2.1.4 Version Control Systems

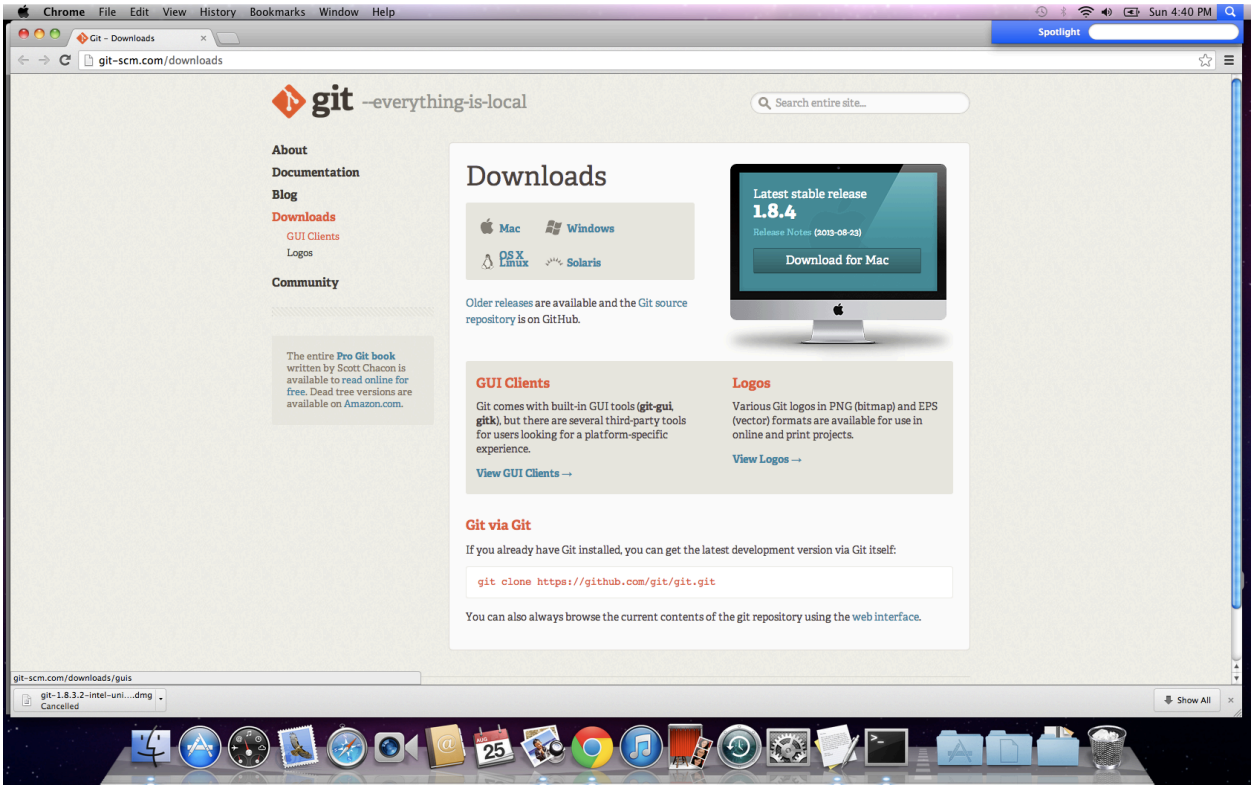
Version control system[21] is a must-have even in an only-one-developer situation. Also many cloud services, e.g., Heroku, require Git for deployment. We also highly recommend getting used to Git and Git terminal commands instead of using Git visual clients/apps with a graphical user interface: GitX[22], Gitbox[23] or GitHub for Mac[24].

Subversion is a non-distributed version control system. This article compares Git vs. Subversion[25].

Here are the steps to install and set up Git on your machine:

1. Download the latest version for your OS at http://git-scm.com/downloads.

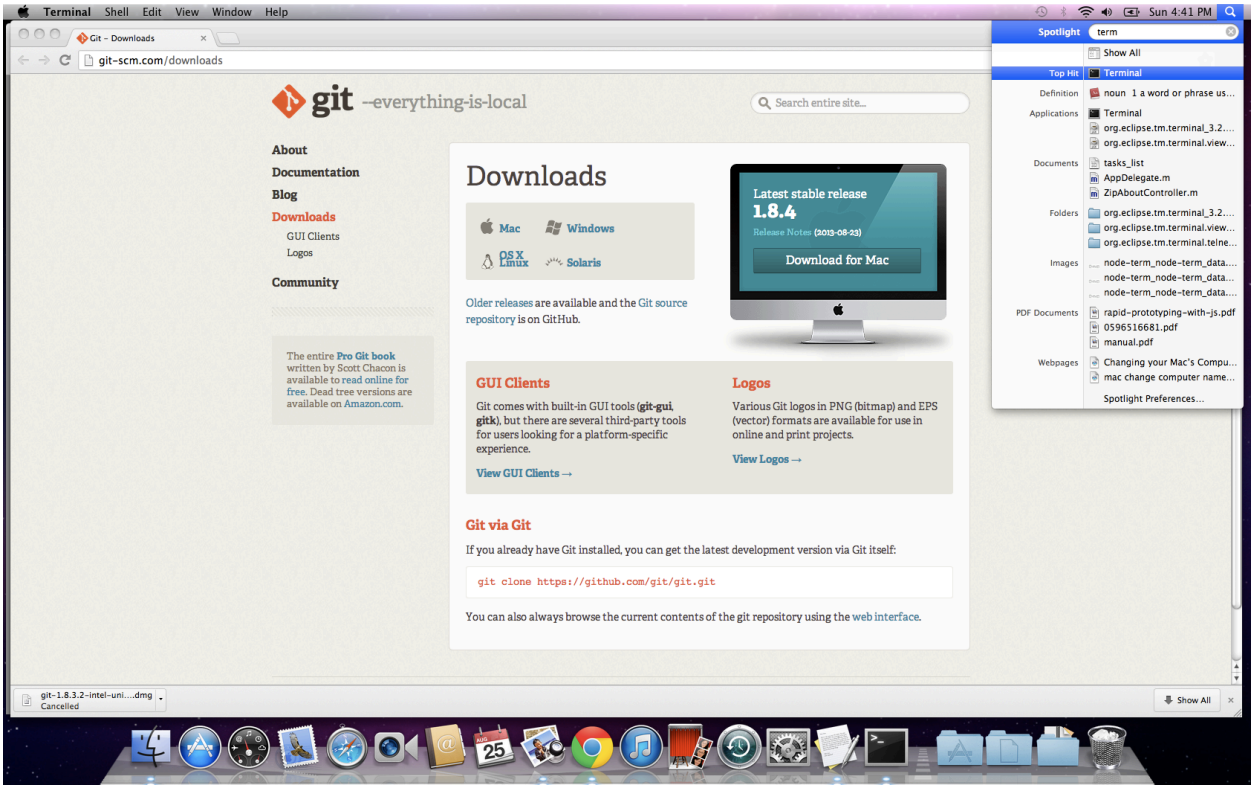
[21] http://en.wikipedia.org/wiki/Revision_control
[22] http://gitx.frim.nl/
[23] http://www.gitboxapp.com/
[24] http://mac.github.com/
[25] https://git.wiki.kernel.org/index.php/GitSvnComparison



Downloading latest release of Git.

2. Install Git from the downloaded `*.dmg` package, i.e., run `*.pkg` file and follow the wizard.
3. Find the terminal app by using Command + Space, a.k.a. Spotlight (please see the screenshot below), on OS X. For Windows you could use PuTTY[26] or Cygwin[27].

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



Using Spotlight to find and run an application.

4. In your terminal, type these commands, **substituting** "John Doe" and johndoe@example.com with your name and email:

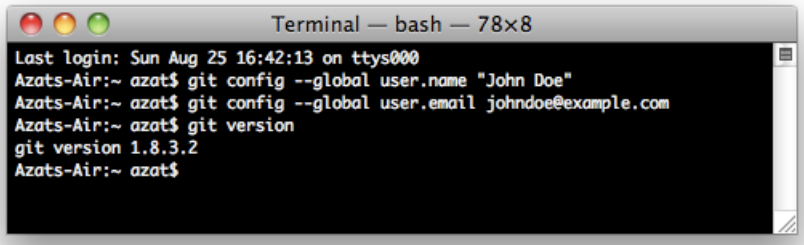
```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

5. To check the installation, run command:

```
$ git version
```

6. You should see something like this in your terminal window (your version might vary; in our case it's 1.8.3.2):

```
git version 1.8.3.2
```



Configuring & Testing Git installation.

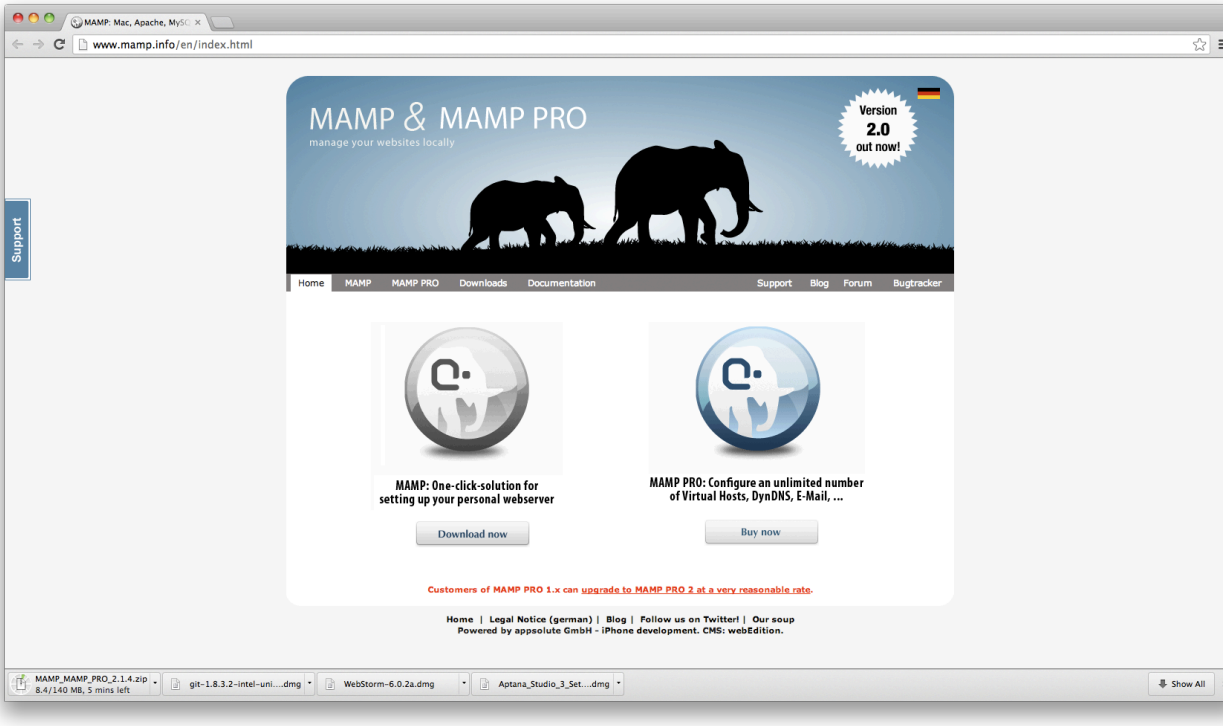
Generation of SSH keys and uploading them to SaaS/PaaS websites will be covered later.

2.1.5 Local HTTP Servers

While you can do most of the front-end development without a local HTTP server, it is needed for loading files with HTTP Requests/AJAX calls. Also, it's just a good practice in general to use a local HTTP server. This way, your development environment is as close to the production environment as possible. You might want to consider the following modifications of the Apache web server:

- MAMP[28]: Mac, Apache, MySQL, PHP personal web server for Mac OS X
- MAMP Stack[29]: Mac app with PHP, Apache, MySQL and phpMyAdmin stack build by BitNami (Apple app store[30])
- XAMPP[31]: Apache distribution containing MySQL, PHP and Perl for Windows, Mac, Linux and Solaris.

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



MAMP for Mac home page.

MAMP, MAMP Stack and XAMPP have intuitive Graphical User Interfaces (GUIs) which allow you to change configurations and host file settings.

Note

Node.js as many other back-end technologies have their own servers for development.

2.1.6 Database: MongoDB

The following steps are better suited for Max OS X/Linux based systems but with some modification can be used for Windows systems as well, i.e., $PATH variable - step #3. Below, we describe the MongoDB installation from the official package, because we found that this approach is more robust and leads to less conflicts. However, there are many other ways to install it on Mac[32], for example using Brew, as well as on other systems[33].

1. MongoDB can be downloaded at http://www.mongodb.org/downloads. For the latest Apple laptops, like MacBook Air, select OS X 64-bit version. The owners of older Macs should browse the link http://dl.mongodb.org/dl/osx/i386.

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

Tip

To figure out the architecture type of your processor, type the `$ uname -p` in the command line.

1. Unpack the package into your web development folder (~/*Documents/Development* or any other). If you want, you could install MongoDB into */usr/local/mongodb* folder.
2. **Optional**: If you would like to access MongoDB commands from anywhere on your system, you need to add your **mongodb** path to the $PATH variable. For Mac OS X the open system *paths* file with:

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

   or, if you prefer TextMate:

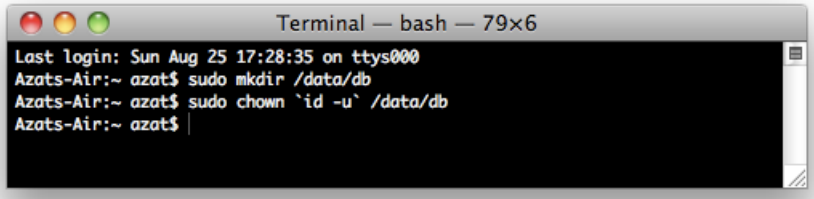
   ```
   mate /etc/paths
   ```

   And add this line to the */etc/paths* file:

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

3. Create a data folder; by default, MongoDB uses */data/db*. Please note that this might be different in a new versions of MongoDB. To create it, type and execute the following commands in the terminal:

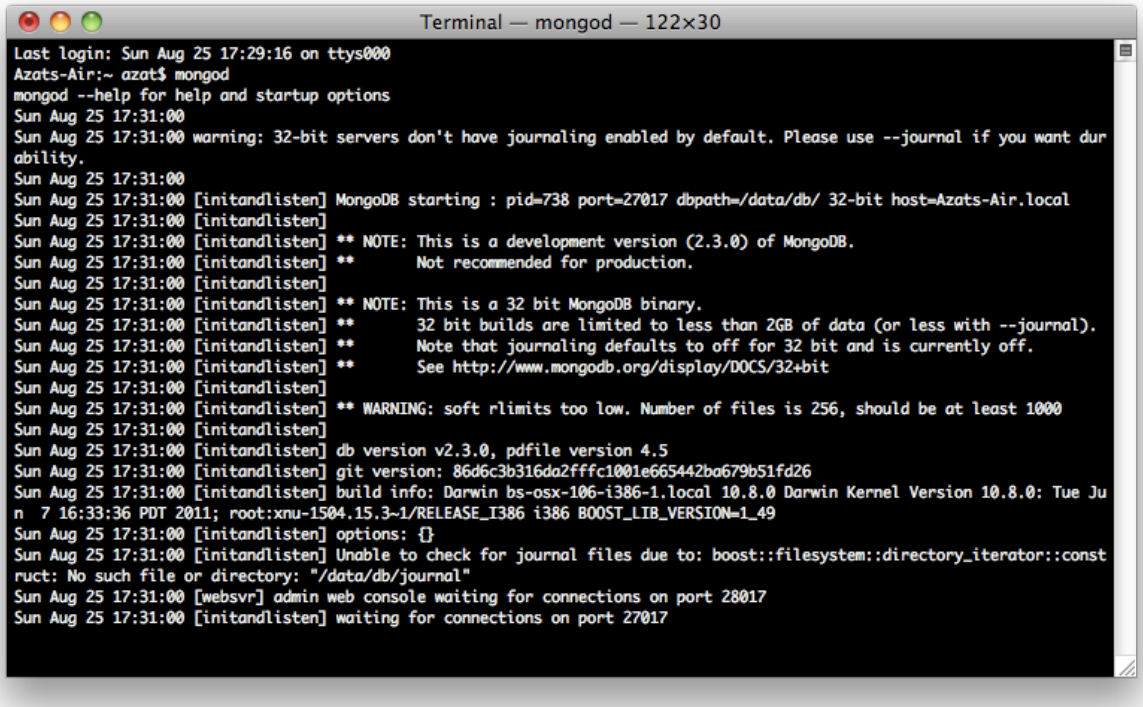
   ```
   $ sudo mkdir -p /data/db
   $ sudo chown `id -u` /data/db
   ```



Initial setup for MongoDB: create the data directory.

   If you prefer to use path other than */data/db* you could specify it using the –dbpath option to **mongod** (main MongoDB service).
4. Go to the folder where you unpacked MongoDB. That location should have a *bin* folder in it. From there, type the following command in your terminal:

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



Starting-up the MongoDB server.

5. If you see something like

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

   it means that the MongoDB database server is running. By default, it's listening at http://localhost:27017. If you go to your browser and type http://localhost:28017 you should be able to see the version number, logs and other useful information. In this case MondoDB server is using **two** different ports (27017 and 28017): one is primary (native) for the communications with apps and the other is web based GUI for monitoring/statistics. In our Node.js code we'll be using only 27017.

Note

Don't forget to restart the Terminal window after adding a new path to the $PATH variable.

Now, to take it even further, we can test to determine if we have access to the MongoDB console/shell, which will act as a client to this server. This means that we'll have to keep the terminal window with the server open and running.

1. Open another terminal window at the same folder and execute:

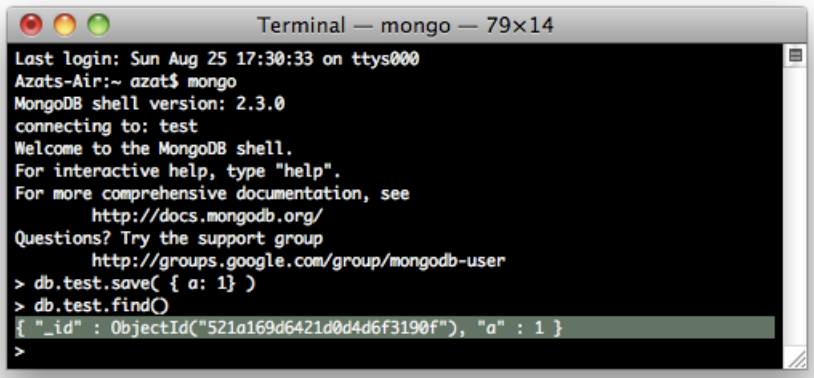
```
$ ./bin/mongo
```

You should be able to see something like "MongoDB shell version 2.0.6 ..."

2. Then type and execute:

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

If you see that your record is being saved, then everything went well:



Running MongoDB client and storing sample data.

Commands *find* and *save* do exactly what you might think they do. ;-)

Detailed instructions are also available at MongoDB.org: Install MongoDB on OS X[34]. For Windows, users there is a good walk-through article: Installing MongoDB[35].

Note

MAMP and XAMPP applications come with MySQL — open-source traditional SQL database, and phpMyAdmin — web interface for MySQL database.

Note

On Max OS X (and most Unix systems), to close the process use `control + c`. If you use `control + z` it will put the process to sleep (or detach the terminal window); in this case, you might end up with the lock on data files and will have to use the kill command or Activity Monitor, and manually delete the locked file in the data folder. In vanilla Mac Terminal `command + .` is an alternative to `control + c`.

[34] http://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x/
[35] http://www.tuanleaded.com/blog/2011/10/installing-mongodb/

2.1.7 Other Components

2.1.7.1 Node.js Installation

Node.js is available at http://nodejs.org/#download (please see the screenshot below). The installation is trivial, i.e., download the archive, run the `*.pkg` package installer. To check the installation of Node.js you could type and execute:

```
$ node -v
```

It should show something similar to this (we use v.0.8.1, but your version might vary):

```
v0.8.1
```

Node.js package already includes Node Package Manager[36] (NPM). We'll use NPM extensively to install Node.js modules.

Node.js home page.

2.1.7.2 JS Libraries

Front-end JavaScript libraries are downloaded and unpacked from their respective websites. Those files are usually put in Development folder (e.g., `~/Documents/Development`) for future use. Oftentimes, there is a

[36] https://npmjs.org

choice between minified production version (more on that in AMD and Require.js section of the *Intro to Backbone.js* chapter) and extensively rich in comments development one.

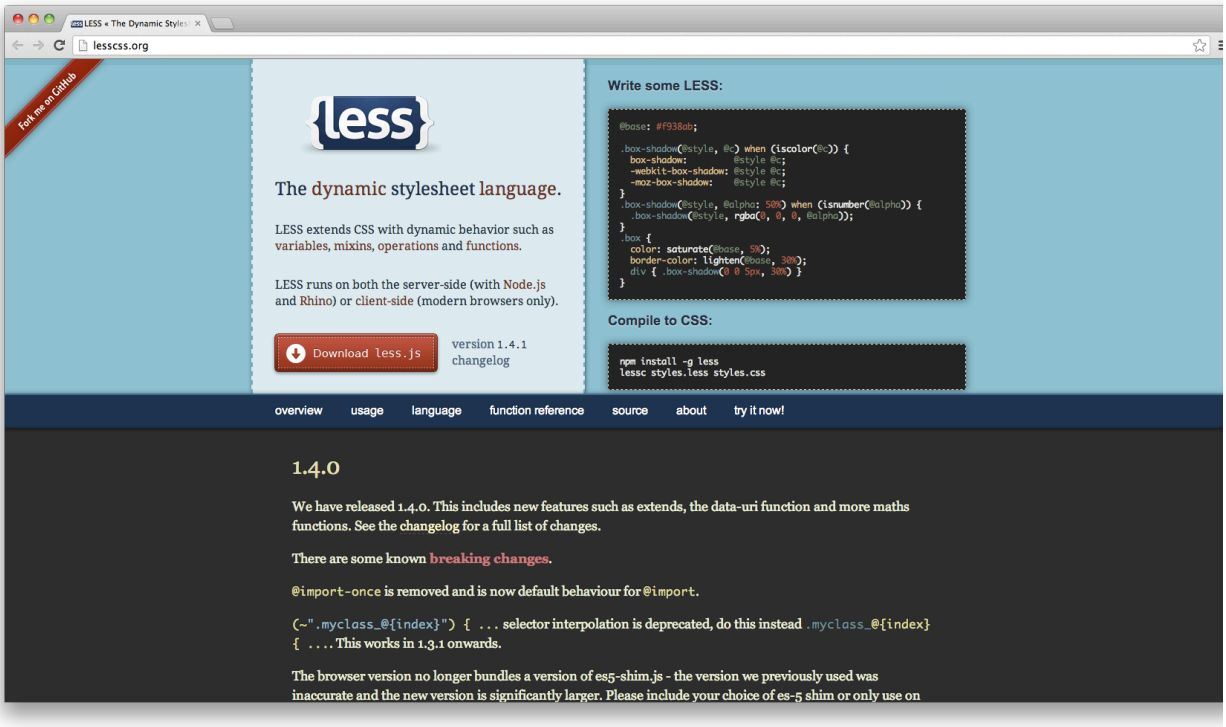
Another approach is to hot-link these scripts from CDNs such as Google Hosted Libraries[37], CDNJS[38], Microsoft Ajax Content Delivery Network[39] and others. By doing so the apps will be faster for some users, but won't work locally at all without the Internet.

- LESS as a front-end interpreter is available at lesscss.org[40] — you could unpack it into your development folder (*~/Documents/Development*) or any other.
- Twitter Bootstrap is a CSS/LESS framework. It's available at twitter.github.com/bootstrap[41].
- jQuery is available at jquery.com[42].
- Backbone.js is available at backbonejs.org[43].
- Underscore.js is available at underscorejs.org[44].
- Require.js is available at requirejs.org[45].

2.1.7.3 LESS App

The LESS App is a Mac OS X application for "on-the-fly" compilation of LESS to CSS. It's available at incident57.com/less[46].

[37] https://developers.google.com/speed/libraries/devguide
[38] http://cdnjs.com/
[39] http://www.asp.net/ajaxlibrary/cdn.ashx
[40] http://lesscss.org/
[41] http://twitter.github.com/bootstrap/
[42] http://jquery.com
[43] http://backbonejs.org
[44] http://underscorejs.org
[45] http://requirejs.org
[46] http://incident57.com/less/



LESS App for Mac home page.

2.2 Cloud Setup

2.2.1 SSH Keys

SSH keys provide a secure connection without the need to enter username and password every time. For GitHub repositories, the latter approach is used with HTTPS URLs, e.g., `https://github.com/azat-co/rpjs.git`, and the former with SSH URLs, e.g., `git@github.com:azat-co/rpjs.git`.

To generate SSH keys for GitHub on Mac OS X/Unix machines do the following:

1. Check for existing SSH keys

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

2. If you see some files like **id_rsa** (please refer to the screenshot below for an example), you could delete them or backup into a separate folder by using following commands:

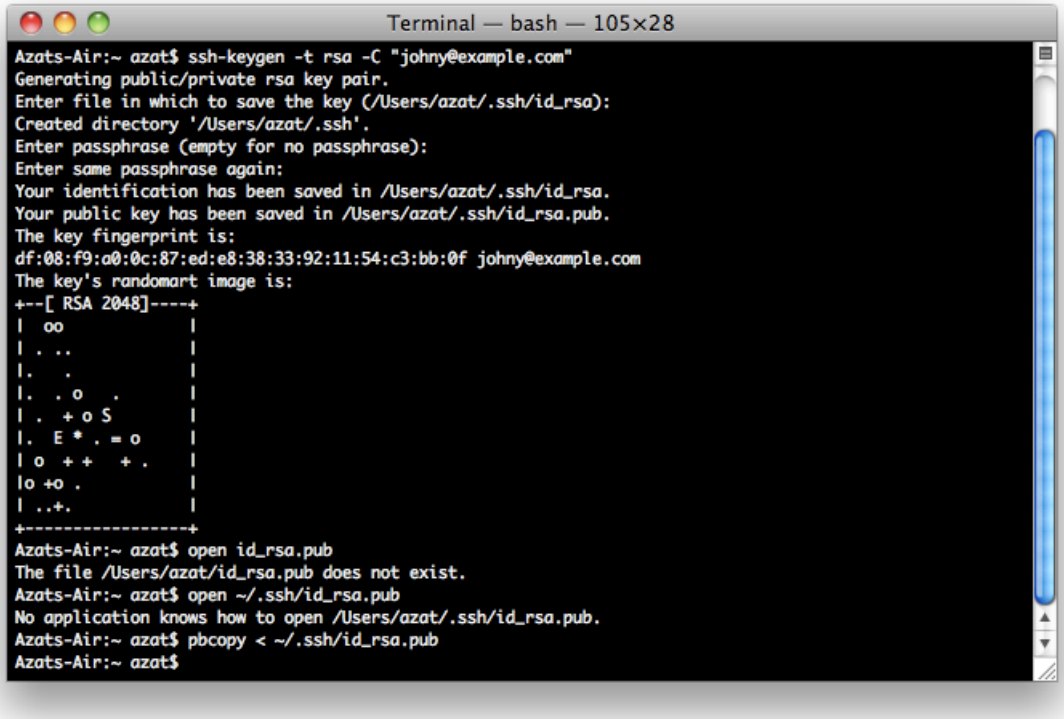
```
$ mkdir key_backup
$ cp id_rsa* key_backup
$ rm id_rsa*
```

3. Now we can generate a new SSH key pair using the **ssh-keygen** command, assuming we are in `~/.ssh` folder:

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

4. Answer the questions; it is better to keep the default name: **id_rsa**. Then copy the content of the **id_rsa.pub** file to your clipboard:

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



Generating RSA key for SSH and copying public key to clipboard.

5. Or alternatively, open **id_rsa.pub** file in the default editor:

```
$ open id_rsa.pub
```

6. Or in TextMate:

```
$ mate id_rsa.pub
```

2.2.2 GitHub

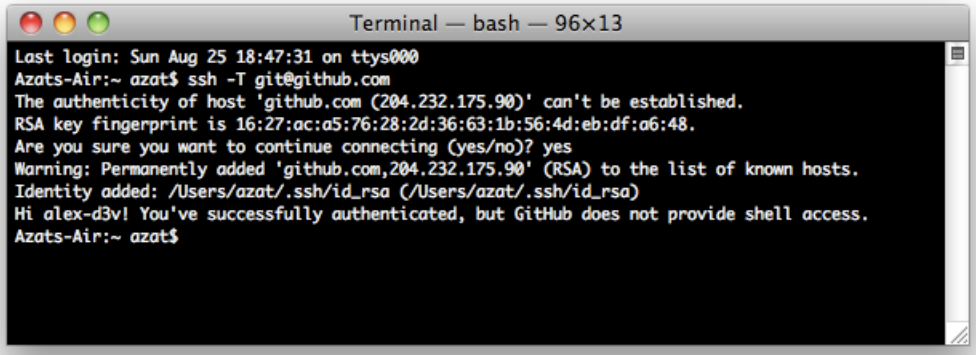
1. After you have copied the public key, go to github.com[47], log in, go to your account settings, select "SSH key" and add the new SSH key. Assign a name, e.g., the name of your computer, and paste the value of your **public** key.
2. To check if you have an SSH connection to GitHub, type and execute the following command in your terminal:

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

   If you see something like:

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

   then everything is set up.
3. While the first time connecting to GitHub, you can receive "authenticity of host ... can't be established" warning. Please don't be confused with such a message — just proceed by answering 'yes' as shown on the screenshot below.



Testing SSH connection to GitHub for the very first time.

If for some reason you have a different message, please repeat steps 3-4 from the previous section on *SSH Keys* and/or re-upload the content of your `*.pub` file to GitHub.

Warning

Keep your **id_rsa** file private and don't share it with anybody!

More instructions are available at GitHub: Generating SSH Keys[48].

Windows users might find useful the SSH key generator feature in [PuTTY].

[47] http://github.com
[48] https://help.github.com/articles/generating-ssh-keys

2.2.3 Windows Azure

Here are the steps to set up a Windows Azure account:

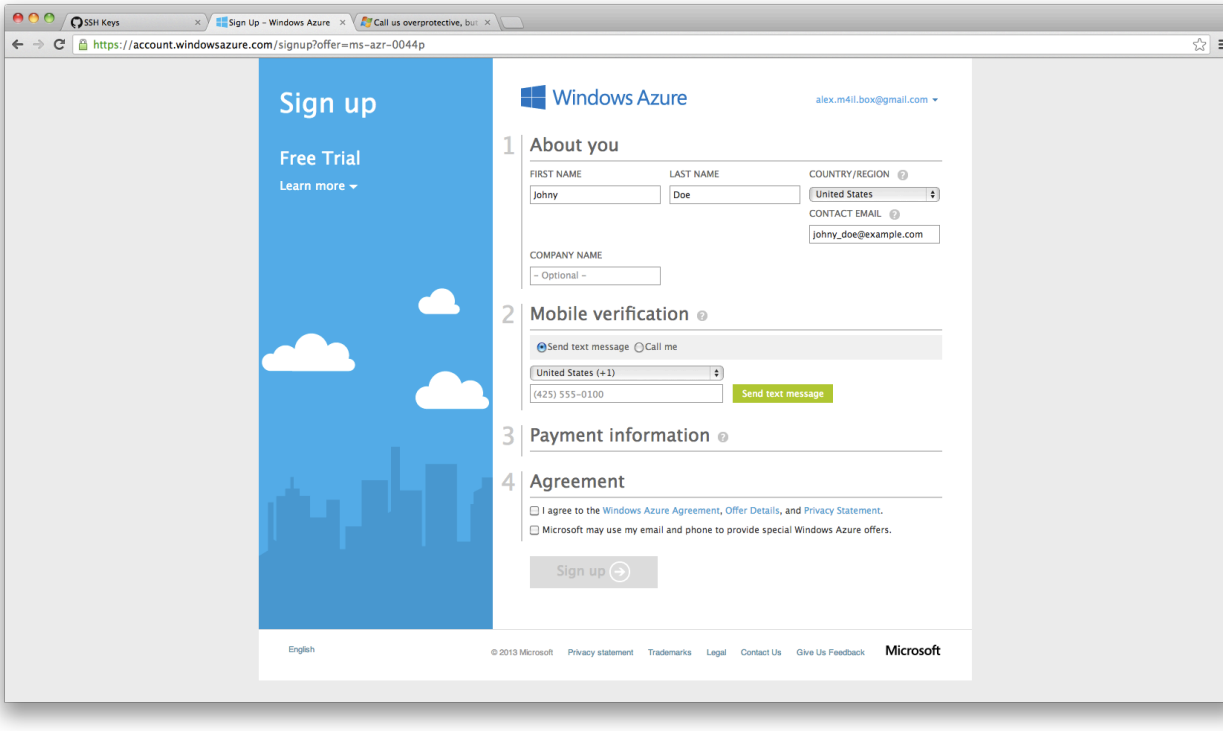
1. You'll need to sign up for Windows Azure Web Site and Virtual Machine previews. Currently they have a 90-day free trial https://www.windowsazure.com/en-us/.
2. Enable Git Deployment and create a username and password. Then upload SSH public key to Windows Azure.
3. Install Node.js SDK, which is available at https://www.windowsazure.com/en-us/develop/nodejs/.
4. To check your installation type:

   ```
   $ azure -v
   ```

   You should be able to see something like:

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

5. Log in to Windows Azure Portal at https://windows.azure.com/.



Registering on Windows Azure.

6. Select "New," then select "Web Site," "Quick Create." Type the name which will serve as the URL for your website, and click "OK."
7. Go to this newly created Web Site's Dashboard and select "Set up Git publishing." Come up with a username and password. This combination can be used to deploy to any web site in your subscription, meaning that you do not need to set credentials for every web site you create. Click "OK."
8. On the follow-up screen, it should show you the Git URL to push to, something like

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

and instructions on how to proceed with deployment. We'll cover them later.

9. **Advanced user option**: follow this tutorial to create a virtual machine and install MongoDB on it: Install MongoDB on a virtual machine running CentOS Linux in Windows Azure[49].

2.2.4 Heroku

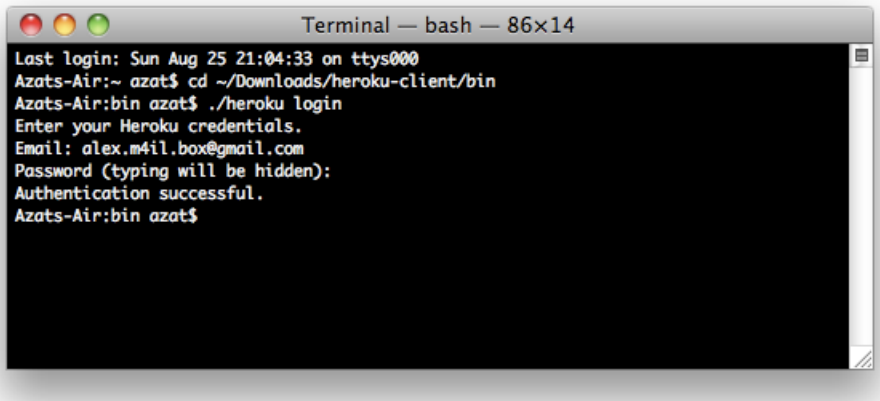
Heroku is a polyglot agile application deployment http://www.heroku.com/ platform. Heroku works similarly to Windows Azure in the sense that you can use Git to deploy applications. There is no need to install Virtual Machine for MongoDB because Heroku has MongoHQ add-on[50].

To set up Heroku, follow these steps:

1. Sign up at http://heroku.com. Currently they have a free account; to use it, select all options as minimum (0) and database as shared.
2. Download Heroku Toolbelt at https://toolbelt.heroku.com. Toolbelt is a package of tools, i.e., libraries which consists of Heroku, Git, and Foreman[51]. For users of older Macs get this client[52] direclty. If you utilize another OS, browse Heroku Client GitHub[53].
3. After the installation is done, you should have access to the **heroku** command. To check it and log in to Heroku, type:

   ```
   $ heroku login
   ```

   It will ask you for Heroku credentials (username and password), and if you've already created the SSH key, it will automatically upload it to the Heroku website:



The response to the successful `$ heroku login` command.

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

4. If everything went well, to create a Heroku application inside of your specific project folder, you should be able to run:

```
$ heroku create
```

More detailed step-by-step instructions are available at Heroku: Quickstart[54] and Heroku: Node.js[55].

2.2.5 Cloud9

Cloud9 is an in-browser IDE with which, by using your GitHub or BitBucket account, you can browse your repositories, edit them and deploy to Windows Azure or other services. No installations are needed; everything works in the browser, pretty much like Google Docs.

[54] https://devcenter.heroku.com/articles/quickstart
[55] https://devcenter.heroku.com/articles/nodejs

II Front-End Prototyping

3 jQuery and Parse.com

Summary: overview of main jQuery functions, Twitter Bootstrap scaffolding, main LESS components; definitions of JSON, AJAX and CORS; illustrations of JSONP calls on Twitter REST API example; explanations on how to build Chat front-end only application with jQuery and Parse.com; step-by-step instructions on deployment to Heroku and Windows Azure.

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult." — Tony Hoare[1]

3.1 Definitions

3.1.1 JavaScript Object Notation

Here is the definition of JavaScript Object Notation, or JSON, from json.org[2]:

> JavaScript Object Notation, or JSON, is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999[3]. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON has become a standard for transferring data between different components of web/mobile applications and third-party services. JSON is also widely used inside the applications as a format for configuration, locales, translation files or any other data.

Typical JSON object looks like this:

[1] http://en.wikipedia.org/wiki/Charles_Antony_Richard_Hoare
[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"
}
```

We have an object with key-value pairs. Keys are on the left and values are on the right side of colons (:). In Computer Science terminology, JSON is equivalent to a hash table, a keyed list or an associative array (depending on a particular language). The only big difference between JSON and JS object literal notation (native JS objects) is that the former is more stringent and requires double quotes (") for key identifiers and string values. Both types can be serialized into a string representation with `JSON.stringify()` and deserialized with `JSON.parse()` assuming we have a valid JSON object in a string format.

However, every member of an object can be an array, primitive, or another object; for example:

```
{
  "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();
  }
}
```

In the above example, we have an object with the **posts** property. The value of the **posts** property is an array of objects with each one of them having **title**, **votes** and **comments** keys. The votes property holds a number primitive, while comments is an array of strings. We also can have function as values; in this case, key is called a method, i.e., `getData`.

JSON is much more flexible and compact than XML or other data formats as outlined in this article — JSON: The Fat-Free Alternative to XML[4]. Conveniently, MongoDB uses a JSON-like format called BSON[5], or Binary JSON. More on BSON later in the *Node.js and MongoDB* chapter.

3.1.2 AJAX

AJAX stands for Asynchronous JavaScript and XML, and is used on a client-side (browser) to send and receive data from the server by utilizing an **XMLHttpRequest** object in JavaScript language. Despite the name, the

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

use of XML is not required, and JSON is often used instead. That's why developers almost never say AJAX anymore. Keep in mind that HTTP requests could be made synchronously, but it's not a good practice to do so. The most typical example of a sync request would be the **script** tag inclusion.

3.1.3 Cross-Domain Calls

For security reasons, the initial implementation of a XMLHTTPRequest object did not allow for cross-domain calls, i.e., when a client-side code and a server-side one are on different domains. There are methods to work around this issue.

One of them is to use JSONP[6] — JSON with padding/prefix. It's basically a dynamically, via DOM manipulation, generated **script** tag. Script tags don't fall into the same domain limitation. The JSONP request includes a name of a callback function in a request query string. For example, the `jQuery.ajax()` function automatically generates a unique function name and appends it to the request (which is a one string broken into multiple lines for readability):

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

The second approach is to use Cross-Origin Resource Sharing, or CORS[7], which is a better solution but it requires control over the server-side to modify response headers. We'll use this technique in the final version of the Chat example application.

Example of CORS server response header:

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

More about CORS: Resources by Enable CORS [8] and [Using CORS by HTML5 Rocks Tutorials] (http://www.html5rocks.com/en/
Test CORS requests at test-cors.org[9].

3.2 jQuery

During the training we'll be using jQuery (http://jquery.com/) for DOM manipulations, HTTP Requests and JSONP calls. jQuery became a de facto standard because of its $ object/function, which provides a simple yet efficient way to access any HTML DOM element on a page by its ID, class, tag name, attribute value, structure or by any combination of thereof. The syntax is very similar to CSS, where we use `#` for id and `.` for class selection, e.g.:

[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

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

Here is the list of most commonly used jQuery API functions:

- **find()**[10]: Selects elements based on the provided selector string
- **hide()**[11]: Hides an element if it was visible
- **show()**[12]: Shows an element if it was hidden
- **html()**[13]: Gets or sets an inner HTML of an element
- **append()**[14] Injects an element into the DOM after the selected element
- **prepend()**[15] Injects an element into the DOM before the selected element
- **on()**[16]: Attaches an event listener to an element
- **off()**[17] Detaches an event listener from an element
- **css()**[18]: Gets or sets the style attribute value of an element
- **attr()**[19] Gets or sets any attribute of an element
- **val()**[20]: Gets or sets the value attribute of an element
- **text()**[21]: Gets the combined text of an element and its children
- **each()**[22]: Iterates over a set of matched elements

Most jQuery functions act not only on a single element, on which they are called, but on a set of matched elements if the result of selection has multiples items. This is a common pitfall that leads to bugs, and it usually happens when a jQuery selector is too broad.

Also, jQuery has many plug-ins and libraries, e.g., jQuery UI[23], jQuery Mobile[24], which provide a rich user interface or other functionality.

3.3 Twitter Bootstrap

Twitter Bootstrap[25] is a collection of CSS/LESS rules and JavaScript plug-ins for creating good User Interface and User Experience without spending *a lot of time* on such details as round-edge buttons, cross-compatibility,

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

responsiveness, etc. This collection or framework is perfect for rapid prototyping of your ideas. Nevertheless, due to its ability to be customized, Twitter Bootstrap is also a good foundation for serious projects. The source code is written in LESS[26], but plain CSS could be downloaded and used as well.

Here is a simple example of using Twitter Bootstrap scaffolding. The structure of the project should look like this:

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

First let's create the **index.html** file with proper tags:

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

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

Include Twitter Bootstrap library as a minified CSS file:

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

Apply scaffolding with container-fluid and row-fluid classes:

[26] http://lesscss.org/

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

Twitter Bootstrap uses a 12-column grid. The size of an individual cell could be specified by classes **spanN**, e.g.,"span1", "span2", "span12". There are also classes **offsetN**, e.g., "offset1", "offset2", ... "offset12" classes to move cells to the right. A complete reference is available at http://twitter.github.com/bootstrap/scaffolding.html.

We'll use **span12** and **hero-unit** classes for the main content block:

```
<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 -->
```

The full source code of the **index.html** from 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"
                      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 -->
    </div>  <!-- container-fluid -->
```

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

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

This example is available for downloading/pulling at GitHub public repository github.com/azat-co/rpjs[28] under rpjs/bootstrap folder[29].

Other useful tools - CSS frameworks and CSS preprocessors worth checking out:

- Compass[30]: CSS framework
- SASS[31]: extension of CSS3 and analog to LESS
- Blueprint[32]: CSS framework
- Foundation[33]: responsive front-end framework
- Bootswatch[34]: collection of customized Twitter Bootstrap themes
- WrapBootstrap[35]: market place for customized Bootstrap themes

3.4 LESS

LESS is the dynamic stylesheet language. Sometimes, and in this case, it's true that less is more and more is less[36]. :-)

The browser cannot interpret LESS syntax, so LESS source code must be compiled to CSS in one of the three ways:

1. In the browser by LESS JavaScript library[37]
2. On the server-side by language/framework, i.e., for Node.js there is the LESS module[38]
3. Locally on your Mac OS X machine by LESS App[39], SimpLESS[40] or a similar app

Warning

Option 1. is okay for a development environment but suboptimal for a production environment.

LESS has variables, mixins and operators, which make it faster for developers to reuse CSS rules. Here is an example of a variable:

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

3.4.1 Variables

Variables reduce redundancy and allow developers to change values fast by having them in one canonical place. And we know that in design (and styling) we often have to change values *very frequently*!

LESS code:

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

Equivalent in CSS:

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

3.4.2 Mixins

This is the syntax for mixin, which acts like a function:

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

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

Converts to this in 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;
}
```

Mixins can be used without parameters, or with multiple parameters.

3.4.3 Operations

With operations, we can perform math functions on numbers, colors or variables.

Example of an operator in 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%);
}
```

The above code compiles in this CSS:

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

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

As you can see, LESS dramatically improves the reusability of plain CSS. Here are some online tools for compilation:

- LESS2CSS[41]: slick browser-based LESS to CSS converter built on Express.js
- lessphp[42]: online demo compiler
- Dopefly[43]: online LESS converter

Other LESS features[44] include:

- Pattern-matching
- Nested Rules
- Functions
- Namespaces
- Scope
- Comments
- Importing

3.5 Example of using third-party API (Twitter) and jQuery

The example is for **purely** demonstrative purposes. It is not a part of the main Chat application covered in later chapters. The goal is to just illustrate the combination of jQuery, JSONP, and REST API technologies. Please take a look through the code and don't attempt to run it, because recently Twitter retired its API v1.0. This application most likely **won't runt as-is**. If you still want to run this example do so at your own risk by following the instructions below, downloading it from GitHub or copy-pasting it from the PDF version.

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

Note

This example was built with Twitter API v1.0 and might not work with Twitter API v1.1, which requires user authentication for REST API calls. You can get the necessary keys at dev.twitter.com[45].

In this example, we'll use jQuery's `$.ajax()` function. It has the following syntax:

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

In the code fragment of an `ajax()` function above, we used following parameters:

- **url** is an endpoint of the API
- **dataType** is the type of data we expect from the server, e.g., "json", "xml"
- **data** is the data to be sent to the server
- **jsonpCallback** is a name of the function, in a string format, to be called after the request comes back
- **type** is HTTP method of the request, e.g., "GET", "POST"

For more parameters and examples of `ajax()` function, go to api.jquery.com/jQuery.ajax[46].

To assign our function to a user-triggered event, we need to use the `click` function from the jQuery library. The syntax is very simple:

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

`$("#btn")` is a jQuery object which points to HTML element in the Document Object Model (DOM) with the **id** of "btn". An HTML code for the button itself, with Twitter Bootstrap classes applied:

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

To make sure that all of the elements we want to access/use are in the DOM, we need to enclose *all* of the DOM manipulation code inside of the following jQuery function:

[45] https://dev.twitter.com
[46] http://api.jquery.com/jQuery.ajax/

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

Note

This is a common mistake with dynamically generated HTML elements. They are not available before they have been created and injected into the DOM.

The following one-page application prints the words in a given Twitter user's last 200 tweets sorted by frequency of use.

For example, if **@jack** had tweeted:

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

The result could be:

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

The source code is available under the rpjs/jquery[47] folder. It's just one file app — **index.html**, and the main JavaScript algorithm implemented this way:

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

We use `document.ready` to postpone execution until the DOM is fully loaded.

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

This lets us attach the click event listener to an element with the "btn" class.

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

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

We instantiate the username variable and assign it a value from the input field with the username id attribute. On the next line we assign Twitter REST API endpoint URL to a `url` variable. The endpoint responds with tweets from a user's timeline.

```
    if (username!=''){
      list = [ ];  //unique global list of words
      counter = { };
      var pages = 0;
      getData(url);
    }
    else {
      alert('Please enter Twitter username')
    }
```

Checking for an empty username to avoid sending a bad request. If username is provided, `getData()` will make a request. We use a named function which we will have to define later, in order to prevent the callback from bloating (infamous pyramid of doom[48]).

```
  })
});
```

Closing click and ready callback constructions/blocks.

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

The JSONP fetching function that magically (thanks to jQuery) makes cross-domain calls by injecting script tag, and appending the callback function name to the request query string.

We'll use **list** array and **counter** object variables for the algorithm:

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

```
var list = [ ]; //unique global list of words
var counter = { }; //number of time each word is repeated
```

The actual function that performs matching and counting:

```
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,'');
      //some other code ...
      if (words[j].substring(0,4)!="http"&&words[j]!='') {
        if (list.indexOf(words[j])<0) {
          list.push(words[j]);
          counter[words[j]]=1;
        }
        else {
          //add plus one to word coutner
          counter[words[j]]++;
        }
      }
    }
  }
```

This code loops through words and uses hash as a lookup table and counter storage.

```
  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;
      }
    }
  }
```

The following fragment sorts words (by numbers of repetitions), and prints nicely by injecting output into the 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).');
  }
```

The full code of the **index.html** file:

```
$(document).ready(function(){
  $('#btn').click(function() {
    //replace loading image
    var username=$('#username').val();
    //make ajax call, callback
    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 = [ ];  //unique global list of words
      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 = [ ]; //unique global list of words
var counter = { };
var pages = 0;
```

```
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,'');
   ...
      if (words[j].substring(0,4)!="http"&&words[j]!='') {
        if (list.indexOf(words[j])<0) {
          list.push(words[j]);
          counter[words[j]]=1;
        }
        else {
          //add plus one to word couter
          counter[words[j]]++;
        }
      }
    }
  }
  //sort by number of repetitions
  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;
      }
    }
  }
  //print sorted
  //print nicely with number of repetition
  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).');
}
```

Try launching it and see if it works with or without the local HTTP server. **Hint**: It should not work without

an HTTP server because of its reliance on JSONP technology.

Note

This example was built with Twitter API v1.0 and might not work with Twitter API v1.1, which requires user authentication for REST API calls. You can get the necessary keys at dev.twitter.com[49]. If you feel that there must be a working example please submit your feedback to hi@rpjs.co.

3.6 Parse.com

Parse.com[50] is a service which can be a substitute for a database and a server. It started as means to support mobile application development. Nevertheless, with REST API and JavaScript SDK, Parse.com can be used in any web, and desktop, applications for data storage (and much more), thus making it ideal for rapid prototyping.

Go to Parse.com and sign up for a free account. Create an application, and copy the Application ID, REST API Key, and JavaScript Key. We'll need these keys to access our collection at Parse.com. Please note the "Data Browser" tab – that's where you can see you collections and items.

We'll create a simple application which will save values to the collections using Parse.com JavaScript SDK. Our application will consist of an **index.html** file and a **app.js** file. Here is the structure of our project folder:

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

The sample is available at the rpjs/parse[51] folder on GitHub, but you are encouraged to type your own code from scratch. To start, create the **index.html** file:

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

Include the minified jQuery library from Google CDN:

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

Include the Parse.com library from Parse CDN location:

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

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

Include our **app.js** file:

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

Create the **app.js** file and use the `$(document).ready` function to make sure that the DOM is ready for manipulation:

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

Change **parseApplicationId** and **parseJavaScriptKey** to values from the Parse.com application dashboard:

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

Since we've included the Parse JavaScript SDK library, we now have access to the global object `Parse`. We initialize a connection with the keys, and create a reference to a **Test** collection:

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

This simple code will save an object with the keys **name** and **text** to the Parse.com **Test** collection:

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

Conveniently, the `save()` method accepts the callback parameters `success` and `error` just like the `jQuery.ajax()` function. To get a confirmation, we'll just have to look at the browser console:

```
    success: function(object) {
      console.log("Parse.com object is saved: "+object);
     //alternatively you could use
     //alert("Parse.com object is saved");
    },
```

It's important to know why we failed to save an object:

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

The full source code of **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>
```

The full source code of the **app.js** file:

```
$(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);
     //alternatively you could use
     //alert("Parse.com object is saved");
    },
    error: function(object) {
      console.log("Error! Parse.com object is not saved: "+object);
    }
  });
})
```

Warning

We need to use the JavaScript SDK Key from the Parse.com dashboard with this approach. For the jQuery example, we'll be using the REST API Key from the same webpage. If you get a 401 Unauthorized error from Parse.com, that's probably because you have a wrong API key.

If everything was done properly, you should be able to see the **Test** in Parse.com's Data Browser populated with values "John" and "hi". Also, you should see the proper message in your browser's console in Developer Tools. Parse.com automatically creates objectIDs and timestamps, which will be very useful in our Chat application.

Parse.com also has thorough instructions for the 'Hello World' application which are available at the Quick Start Guide sections for new projects[52] and existing ones[53].

3.7 Chat with Parse.com Overview

The Chat will consist of an input field, a list of messages and a send button. We need to display a list of existing messages and be able to submit new messages. We'll use Parse.com as a back-end for now, and later switch to Node.js with MongoDB.

You can get a free account at Parse.com. The JavaScript Guide is available at https://parse.com/docs/js_guide and JavaScript API is available at https://parse.com/docs/js/.

After signing up for Parse.com, go to the dashboard and create a new app if you haven't done so already. Copy your newly created app's Application ID and JavaScript key and REST API Key. We'll need it later. There are a few ways to use Parse.com[54]:

- REST API: We're going to use this approach with the jQuery example
- JavaScript SDK: We just used this approach in our test example above, and we'll use it in the Backbone.js example later

[52] https://parse.com/apps/quickstart#js/blank
[53] https://parse.com/apps/quickstart#js/existing
[54] http://parse.com

REST API is a more generic approach. Parse.com provides endpoints which we can request with the `$.ajax()` method from jQuery library. Here is the description of available URLs and methods: parse.com/docs/rest[55].

3.8 Chat with Parse.com: REST API and jQuery version

The full code is available under the rpjs/rest[56] folder, but we encourage you to try to write your own application first.

We'll use Parse.com's REST API and jQuery. Parse.com supports different origin domain AJAX calls, so we won't need JSONP.

Note

When you decide to deploy your back-end application, which will act as a substitute for Parse.com, on a different domain you'll need to use either JSONP on the front-end or custom CORS headers on a back-end. This topic will be covered later in the book.

Right now the structure of the application should look like this:

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

Let's create a visual representation for the Chat app. What we want is just to display a list of messages with names of users in chronological order. Therefore, a table will do just fine, and we can dynamically create `<tr>` elements and keep inserting them as we get new messages.

Create a simple HTML file **index.html** with the following content:

- Inclusion of JS and CSS files
- Responsive structure with Twitter Boostrap
- A table of messages
- A form for new messages

Let's start with the **head** and dependencies. We'll include CDN jQuery, local **app.js**, local minified Twitter Boostrap and custom stylesheet **style.css**:

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

```
<!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>
```

The body element will have typical Twitter Boostrap scaffolding elements defined by classes **container-fluid** and **row-fluid**:

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

The table of messages is empty, because we'll populate it programmatically from within the JS code:

```
        <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>
```

Another row and here is our new message form in which the *SEND* button uses Twitter Bootstrap classes **btn** and **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>
```

The full source code for **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" />
        <a id="send" class="btn btn-primary">SEND</a>
      </form>
    </div>
  </div>
 </body>
</html>
```

The table will contain our messages. The form will provide input for new messages.

Now we are going to write three main functions:

1. `getMessages()`: the function to get the messages
2. `updateView()`: the function to render the list of messages
3. `$('#send').click(...)`: the function that triggers sending a new message

To keep things simple, we'll put all of the logic in one file **app.js**. Of course, it a good idea to separate code base on the functionality when your project grows larger.

Replace these values with your own, and be careful to use the REST API Key (not the JavaScript SDK Key from the previous example):

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

Wrap everything in `document.ready`, fetch messages, and define *SEND* on click event:

```
$(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('!'')
```

When we submit a new message, we make the HTTP call with the `jQuery.ajax` function:

```
    $.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');
      }
    });

  });
})
```

The method to fetch messages from our remote REST API server also uses the `jQuery.ajax` function:

```
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',
```

If the request is completed successfully (status 200 or similar), we call the `updateView` function:

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

This function is rendering the list of messages which we get from the server:

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

We use the jQuery selector '.table tbody' to create an object referencing that element. Then we clean all the innerHTML of that element:

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

We use the `jQuery.each` function to iterate through every message:

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

The following code creates HTML elements (and the jQuery object of those elements) programmatically:

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

And append (injects after) the table's `tbody` element:

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

Then all of **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',
    success: function(data) {
      console.log('get');
      updateView(data);
    },
    error: function() {
      console.log('error');
    }
  });
}

function updateView(messages) {
  var table=$('.table tbody');
  table.html('');
  $.each(messages.results, function (index, value) {
    var trEl =
      $('<tr><td>'
        + value.username
        + '</td><td>'
        + value.message +
        '</td></tr>');
    table.append(trEl);
  });
  console.log(messages);
}
```

What it will do is call the `getMessages()` function and make a GET request with the `$.ajax` function from the jQuery library. A full list of parameters for the `ajax` function is available at api.jquery.com/jQuery.ajax[57]. The most important ones are URL, headers and type parameters.

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

Then, upon successful response, it will call the `updateView()` function, which clears the table **tbody** and iterates through results of the response using the `$.each` jQuery function (api.jquery.com/jQuery.each[58]). Clicking on the send button will send a POST request to Parse.com REST API and then, upon successful response, get messages calling the `getMessages()` function.

Here is one of the ways to dynamically create the *div* HTML element using jQuery:

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

3.9 Pushing to GitHub

To create a GitHub repository, go to github.com[59], log in and create a new repository. There will be an SSH address; copy it. In your terminal window, navigate to the project folder which you would like to push to GitHub.

1. Create a local Git and *.git* folder in the root of the project folder:

   ```
   $ git init
   ```

2. Add all of the files to the repository and start tracking them:

   ```
   $ git add .
   ```

3. Make the first commit:

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

4. Add the GitHub remote destination:

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

   It might look something like this:

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

5. Now everything should be set to push your local Git repository to the remote destination on GitHub with the following command:

   ```
   $ git push origin master
   ```

6. You should be able to see your files at github.com[60] under your account and repository.

Later, when you make changes to the file, there is no need to repeat all of the steps above. Just execute:

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

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

If there are no new untracked files which you want to start tracking:

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

To include changes from individual files, run:

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

To remove a file from the Git repository:

```
$ git rm filename
```

For more Git commands:

```
$ git --help
```

Deploying applications with Windows Azure or Heroku is as simple as pushing code/files to GitHub. The last three steps (#4-6) would be substituted with a different remote destination (URL) and a different alias.

3.10 Deployment to Windows Azure

You should be able to deploy to Windows Azure with Git.

1. Go to the Windows Azure Portal at https://windows.azure.com/, log in with your Live ID and create a Web Site if you haven't done so already. Enable "Set up Git publishing" by providing a username and password (they should be different from your Live ID credentials). Copy your URL somewhere.
2. Create a local Git repository in the project folder which you would like to publish or deploy:

   ```
   $ git init
   ```

3. Add all of the files to the repository and start tracking them:

   ```
   $ git add .
   ```

4. Make the first commit:

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

5. Add Windows Azure as a remote Git repository destination:

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

   In my case, this command looked like this:

   ```
   $ git remote add


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

6. Push your local Git repository to the remote Windows Azure repository, which will deploy the files and application:

   ```
   $ git push azure master
   ```

As with GitHub, there is no need to repeat the first few steps when you have updated the files later, since we already should have a local Git repository in the form of *.git* folder in the root of the project folder.

3.11 Deployment to Heroku

The only major difference is that Heroku uses Cedar Stack, which doesn't support static projects, a.k.a. plain HTML applications like our Parse.com test application or Parse.com version of the Chat application. We can use a "fake" PHP project to get past this limitation. Create a file **index.php** on the same level as **index.html** in the project folder, which you would like to publish/deploy to Heroku with the following content:

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

For your convenience, the **index.php** file is already included in *rpjs/rest*.

There is an even simpler way to publish static files on Heroku with Cedar stack which is described in the post Static Sites on Heroku Cedar[61]. In order to make Cedar Stack work with your static files, all you need to do is to type and execute following commands in your project folder:

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

Alternatively, you could use the Ruby Bamboo stack. In this case, we would need the following structure:

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

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

The path in *index.html* to CSS and other assets should be relative, i.e., 'css/style.css'. The *config.ru* file should contain the following code:

```
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)
  ]
}
```

For more details, you could refer to devcenter.heroku.com/articles/static-sites-on-heroku[62].

Once you have all of the support files for Cedar stack, or Bamboo, follow these steps:

1. Create a local Git repository and *.git* folder if you haven't done so already:

   ```
   $ git init
   ```

2. Add files:

   ```
   $ git add .
   ```

3. Commit files and changes:

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

4. Create the Heroku Cedar stack application and add the remote destination:

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

```
$ heroku create
```

If everything went well, it should tell you that the remote has been added and the app has been created, and give you the app name.

5. To look up the remote type and execute (*optional*):

```
$ git remote show
```

6. Deploy the code to Heroku with:

```
$ git push heroku master
```

Terminal logs should tell you whether or not the deployment went smoothly.

7. To open the app in your default browser, type:

```
$ heroku open
```

or just go to the URL of your app, something like "http://yourappname-NNNN.herokuapp.com".

8. To look at the Heroku logs for this app, type:

```
$ heroku logs
```

To update the app with the new code, repeat the following steps **only**:

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

Note

You'll be assigned a new application URL each time you create a new Heroku app with the command: `$ heroku create`.

3.12 Updating and Deleting of Messages

In accordance with REST API, an update on an object is performed via the PUT method and delete — DELETE. Both of them can easily be performed with the same `jQuery.ajax` function that we've used for GET and POST, as long as we provide an ID of an object on which we want to execute an operation.

4 Intro to Backbone.js

Summary: demonstration of how to build Backbone.js application from scratch and use views, collections, subviews, models, event binding, AMD, Require.js on the example of the apple database application.

> *"Code is not an asset. It's a liability. The more you write, the more you'll have to maintain later."*
> — Unknown

4.1 Setting up Backbone.js App from Scratch

We're going to build a typical starter "Hello World" application using Backbone.js and Mode-View-Controller (MVC) architecture. I know it might sound like overkill in the beginning, but as we go along we'll add more and more complexity, including Models, Subviews and Collections.

A full source code for the "Hello World" app is available at GitHub under github.com/azat-co/rpjs/backbone/hello-world[1].

4.1.1 Dependencies

Download the following libraries:

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

And include these frameworks in the **index.html** file like this:

[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 write some awesome JS code!
  </script>

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

Note

We can also put `<script>` tags right after the `</body>` tag in the end of the file. This will change the order in which scripts and the rest of HTML are loaded, and impact performance in large files.

Let's define a simple Backbone.js Router inside of a `<script>` tag:

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

Note

For now, to Keep It Simple Stupid (KISS), we'll be putting all of our JavaScript code right into the **index.html** file. This is not a good idea for a real development or production code. We'll refactor it later.

Then set up a special `routes` property inside of an **extend** call:

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

The Backbone.js **routes** property needs to be in the following format: `'path/:param':'action'` which will result in the `filename#path/param` URL triggering a function named **action** (defined in the Router object). For now, we'll add a single **home** route:

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

This is good, but now we need to add a **home** function:

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

We'll come back to the **home** function later to add more logic for creating and rendering of a View. Right now we should define our **homeView**:

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

It looks familiar, right? Backbone.js uses similar syntax for all of its components: the **extend** function and a JSON object as a parameter to it.

There are a multiple ways to proceed from now on, but the best practice is to use the **el** and ***template*** properties, which are magical, i.e., special in Backbone.js:

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

The property **el** is just a string that holds the jQuery selector (you can use class name with '.' and id name with '#'). The template property has been assigned an Underscore.js function **template** with just a plain text 'Hello World'.

To render our **homeView** we use `this.$el` which is a compiled jQuery object referencing element in an **el** property, and the jQuery `.html()` function to replace HTML with `this.template()` value. Here is what the full code for our Backbone.js View looks like:

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

Now, if we go back to the **router** we can add these two lines to the **home** function:

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

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

The first line will create the *homeView* object and assign it to the *homeView* property of the router. The second line will call the `render()` method in the *homeView* object, triggering the 'Hello World' output.

Finally, to start a Backbone app, we call `new Router` inside of a document-ready wrapper to make sure that the file's DOM is fully loaded:

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

Here is the full code of the **index.html** file:

```
<!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(){
       //some code to execute
       //when the object is instantiated
      },
      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>
```

Open **index.html** in the browser to see if it works, i.e., the 'Hello World' message should be on the page.

4.2 Working with Collections

The full source code of this example is under rpjs/backbone/collections[5]. It's built on top of "Hello World" example from the **Setting up Backbone.js App from Scratch** exercise which is available for download at rpjs/backbone/hello-world[6].

We should add some data to play around with, and to hydrate our views. To do this, add this right after the **script** tag and before the other code:

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

This is our apple *database*. :-) Or to be more correct, our REST API endpoint-substitute, which provides us with names and image URLs of the apples (data models).

Note

This mock dataset can be easily substituted by assigning REST API endpoints of your back-end to `url` properties in Backbone.js Collections and/or Models, and calling the `fetch()` method on them.

Now to make the User Experience (UX) a little bit better, we can add a new route to the **routes** object in the Backbone Route:

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

This will allow users to go to `index.html#apples/SOMENAME` and expect to see some information about an apple. This information will be fetched and rendered by the **loadApple** function in the Backbone Router definition:

[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);
}
```

Have you noticed an **appleName** variable? It's exactly the same name as the one that we've used in **route**. This is how we can access query string parameters (e.g, `?param=value&q=search`) in Backbone.js.

Now we'll need to refactor some more code to create a Backbone Collection, populate it with data in our **appleData** variable, and to pass the collection to **homeView** and **appleView**. Conveniently enough, we do it all in the Router constructor method **initialize**:

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

At this point, we're pretty much done with the Router class and it should look like this:

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

Let's modify our **homeView** a bit to see the whole *database*:

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

For now, we just output the string representation of the JSON object in the browser. This is not user-friendly at all, but later we'll improve it by using a list and subviews.

Our apple Backbone Collection is very clean and simple:

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

Note

Backbone automatically creates models inside of a collection when we use the `fetch()` or `reset()` functions.

Apple view is not any more complex; it has only two properties: **template** and **render**. In a template, we want to display **figure**, **img** and **figcaption** tags with specific values. The Underscore.js template engine is handy at this task:

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

To make a JavaScript string, which has HTML tags in it, more readable we can use the backslash line breaker escape (\) symbol, or close strings and concatenate them with a plus sign (+). This is an example of **appleView** above, which is refactored using the latter approach:

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

Please note the '<%=' and '%>' symbols; they are the instructions for Undescore.js to print values in properties **url** and **name** of the **attributes** object.

Finally, we're adding the render function to the **appleView** class.

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

We find a model within the collection via `where()` method and use `[]` to pick the first element. Right now, the **render** function is responsible for both loading the data and rendering it. Later we'll refactor the function to separate these two functionalities into different methods.

The whole app, which is in the rpjs/backbone/collections/index.html[7] folder, looks like this:

```
<!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

```
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);
  }
});

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)
  }));
  }
  //TODO subviews
});

var Apples = Backbone.Collection.extend({

});
var appleView = Backbone.View.extend({
  template: _.template('<figure>\
              <img src="<%= attributes.url %>"/>\
              <figcaption><%= attributes.name %></figcaption>\
            </figure>'),
  //TODO re-write with load apple and event binding
  render: function(appleName){
    var appleModel = this.collection.where({
      name:appleName
    })[0];
    var appleHtml = this.template(appleModel);
    $('body').html(appleHtml);
```

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

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

Open `collections/index.html` file in your browser. You should see the data from our "database", i.e., `Apple data: [{"name":"fuji","url":"img/fuji.jpg"},{"name":"gala","url":"img/gala.jpg"}]`.

Now, let' go to `collections/index.html#apples/fuji` or `collections/index.html#apples/gala` in your browser. We expect to see an image with a caption. It's a detailed view of an item, which in this case is an apple. Nice work!

4.3 Event Binding

In real life, getting data does not happen instantaneously, so let's refactor our code to simulate it. For a better UI/UX, we'll also have to show a loading icon (a.k.a. spinner or ajax-loader) to users to notify them that the information is being loaded.

It's a good thing that we have event binding in Backbone. Without it, we'll have to pass a function that renders HTML as a callback to the data loading function, to make sure that the rendering function is not executed before we have the actual data to display.

Therefore, when a user goes to detailed view (`apples/:id`) we only call the function that loads the data. Then, with the proper event listeners, our view will automagically (this is not a typo) update itself, when there is a new data (or on a data change, Backbone.js supports multiple and even custom events).

Let's change the code in the router:

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

Everything else remains the same utill we get to the **appleView** class. We'll need to add a constructor or an **initialize** method, which is a special word/property in the Backbone.js framework. It's called each time we create an instance of an object, i.e., `var someObj = new SomeObject()`. We can also pass extra parameters to

the **initialize** function, as we did with our views (we passed an object with the key **collection** and the value of **apples** Backbone Collection). Read more on Backbone.js constructors at backbonejs.org/#View-constructor[8].

```
...
var appleView = Backbone.View.extend({
  initialize: function(){
    //TODO: create and setup model (aka an apple)
  },
...
```

Great, we have our **initialize** function. Now we need to create a model which will represent a single apple and set up proper event listeners on the model. We'll use two types of events, `change` and a custom event called `spinner`. To do that, we are going to use the `on()` function, which takes these properties: `on(event, actions, context)` — read more about it at backbonejs.org/#Events-on[9]:

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

The code above basically boils down to two simple things:

1. Call `render()` function of **appleView** object when the model has changed
2. Call `showSpinner()` method of **appleView** object when event **spinner** has been fired.

So far, so good, right? But what about the spinner, a GIF icon? Let's create a new property in **appleView**:

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

Remember the `loadApple` call in the router? This is how we can implement the function in **appleView**:

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

```
...
loadApple:function(appleName){
  this.trigger('spinner');
  //show spinner GIF image
  var view = this;
  //we'll need to access that inside of a closure
  setTimeout(function(){
  //simulates real time lag when
  //fetching data from the remote server
    view.model.set(view.collection.where({
      name:appleName
    })[0].attributes);
  },1000);
},
...
```

The first line will trigger the `spinner` event (the function for which we still have to write).

The second line is just for scoping issues (so we can use **appleView** inside of the closure).

The `setTimeout` function is simulating a time lag of a real remote server response. Inside of it, we assign attributes of a selected model to our view's model by using a `model.set()` function and a **model.attributes** property (which returns the properties of a model).

Now we can remove an extra code from the `render` method and implement the `showSpinner` function:

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

That's all! Open `index.html#apples/gala` or `index.html#apples/fuji` in your browser and enjoy the loading animation while waiting for an apple image to load.

The full code of the **index.html** file:

```
<!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)
```

```
      }));
    }
    //TODO subviews
  });

  var Apples = Backbone.Collection.extend({

  });
  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; //we'll need to access
      //that inside of a closure
      setTimeout(function(){ //simulates real time
      //lag when fetching data from the remote server
        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);
    }

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

```
    })

  </script>
</head>
<body>
  <a href="#apples/fuji">fuji</a>
  <div></div>
</body>
</html>
```

4.4 Views and Subviews with Underscore.js

This example is available at rpjs/backbone/subview[10].

Subviews are Backbone Views that are created and used inside of another Backbone View. A subviews concept is a great way to abstract (separate) UI events (e.g., clicks), and templates for similarly structured elements (e.g., apples).

A use case of a Subview might include a row in a table, a list item in a list, a paragraph, a new line, etc.

We'll refactor our home page to show a nice list of apples. Each list item will have an apple name and a "buy" link with an **onClick** event. Let's start by creating a subview for a single apple with our standard Backbone `extend()` function:

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

Now we can populate the object with **tagName**, **template**, **events**, **render** and **addToCart** properties/methods.

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

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

tagName automatically allows Backbone.js to create an HTML element with the specified tag name, in this case `<li>` — list item. This will be a representation of a single apple, a row in our list.

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

The template is just a string with Undescore.js instructions. They are wrapped in `<%` and `%>` symbols. `<%=` simply means print a value. The same code can be written with backslash escapes:

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

Each `<li>` will have two anchor elements (`<a>`), links to a detailed apple view (`#apples/:appleName`) and a **buy** button. Now we're going to attach an event listener to the **buy** button:

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

The syntax follows this rule:

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

Both the key and the value (right and left parts separated by the colon) are strings. For example:

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

or

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

To render each item in the list, we'll use the jQuery `html()` function on the `this.$el` jQuery object, which is the `<li>` HTML element based on our **tagName** attribute:

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

addToCart will use the `trigger()` function to notify the collection that this particular model (apple) is up for the purchase by the user:

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

Here is the full code of the **appleItemView** Backbone View class:

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

Easy peasy! But what about the master view, which is supposed to render all of our items (apples) and provide a wrapper `<ul>` container for `<li>` HTML elements? We need to modify and enhance our **homeView**.

To begin with, we can add extra properties of string type understandable by jQuery as selectors to **homeView**:

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

We can use properties from above in the template, or just hard-code them (we'll refactor our code later) in **homeView**:

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

The **initialize** function will be called when **homeView** is created (`new homeView()`) — in it we render our template (with our favorite by now `html()` function), and attach an event listener to the collection (which is a set of apple models):

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

The syntax for the binding event is covered in the previous section. In essence, it is calling the `showCart()` function of **homeView**. In this function, we append **appleName** to the cart (along with a line break, a `<br/>` element):

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

Finally, here is our long-awaited `render()` method, in which we iterate through each model in the collection (each apple), create an **appleItemView** for each apple, create an `<li>` element for each apple, and append that element to **view.listEl** — `<ul>` element with a class apples-list in the DOM:

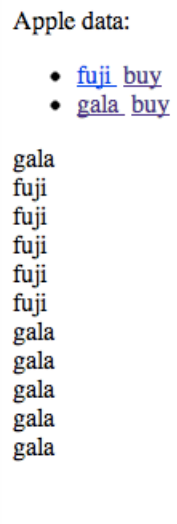
```
...
render: function(){
  view = this;
  //so we can use view inside of closure
  this.collection.each(function(apple){
    var appleSubView = new appleItemView({model:apple});
    // creates subview with model apple
    appleSubView.render();
    // compiles template and single apple data
    $(view.listEl).append(appleSubView.$el);
    //append jQuery object from single
    //apple to apples-list DOM element
  });
}
...
```

Let's make sure we didn't miss anything in the **homeView** Backbone View:

```
...
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; //so we can use view inside of closure
    this.collection.each(function(apple){
      var appleSubView = new appleItemView({model:apple});
      // create subview with model apple
      appleSubView.render();
      // compiles tempalte and single apple data
      $(view.listEl).append(appleSubView.$el);
      //append jQuery object from single apple
      //to apples-list DOM element
    });
```

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

You should be able to click on the buy, and the cart will populate with the apples of your choice. Looking at an individual apple does not require typing its name in the URL address bar of the browser anymore. We can click on the name and it opens a new window with a detailed view.



The list of apples rendered by subviews.

By using subviews, we reused the template for all of the items (apples) and attached a specific event to each of them. Those events are smart enough to pass the information about the model to other objects: views and collections.

Just in case, here is the full code for the subviews example, which is also available at rpjs/backbone/subview/index.html[11]:

```
<!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"
```

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

```
    },
    {
      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 appleItemView = Backbone.View.extend({
    tagName: 'li',
    // template: _.template('' 
    //    +'<a href="#apples/<%=name%>" target="_blank">'
    //    +'<%=name%>'
    //    +'</a> <a class="add-to-cart" href="#">buy</a>'),
    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);
    }
  });

  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; //so we can use view inside of closure
      this.collection.each(function(apple){
        var appleSubView = new appleItemView({model:apple});
        // create subview with model apple
        appleSubView.render();
        // compiles tempalte and single apple data
        $(view.listEl).append(appleSubView.$el);
        //append jQuery object from
        //single apple to apples-list DOM element
      });
    }
  });

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

  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;
        //we'll need to access that inside of a closure
        setTimeout(function(){
        //simulates real time lag when fetching data
        // from the remote server
          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);
      }
    });

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

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

The link to an individual item, e.g., `collections/index.html#apples/fuji`, also should work independently, by typing it in the browser address bar.

4.5 Refactoring

At this point you are probably wondering what is the benefit of using the framework and still having multiple classes, objects and elements with different functionalities in one *single* file. This was done for the purpose of adhering to the *Keep it Simple Stupid* (KISS) principle.

The bigger your application is, the more pain there is in unorganized code base. Let's break down our application into multiple files where each file will be one of these types:

- view
- template
- router
- collection
- model

Let's write these scripts to include tags into our **index.html** head — or body, as noted previously:

```
  <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>
```

The names don't have to follow the convention of dashes and dots, as long as it's easy to tell what each file is supposed to do.

Now, let's copy our objects/classes into the corresponding files.

Our main **index.html** file should look very minimalistic:

```
<!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>
```

The other files just have the code that corresponds to their filenames.

The content of **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;
    //we'll need to access that inside of a closure
    setTimeout(function(){
    //simulates real time lag when fetching
    //data from the remote server
      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);
  }

});
```

The **apple-home.view.js** file has the **homeView** object:

```
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; //so we can use view inside of closure
    this.collection.each(function(apple){
      var appleSubView = new appleItemView({model:apple});
      // create subview with model apple
      appleSubView.render();
      // compiles tempalte and single apple data
      $(view.listEl).append(appleSubView.$el);
      //append jQuery object from
      //single apple to apples-list DOM element
    });
  }
});
```

The **apple.view.js** file contains the master apples' list:

```
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;
```

```
    //we'll need to access that inside of a closure
    setTimeout(function(){
    //simulates real time lag when
    //fetching data from the remote server
      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);
  }
});
```

apples.js is an empty collection:

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

apple-app.js is the main application file with the data, the router, and the starting command:

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

Now let's try to open the application. It should work exactly the same as in the previous Subviews example.

It's a way better code organization, but it's still far from perfect, because we still have HTML templates directly in the JavaScript code. The problem is that designers and developers can't work on the same files, and any change to the presentation requires a change in the main code base.

We can add a few more JS files to our **index.html** file:

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

Usually, one Backbone view has one template, but in the case of our **appleView** — detailed view of an apple in a separate window — we also have a spinner, a "loading" GIF animation.

The contents of the files are just global variables which are assigned some string values. Later we can use these variables in our views, when we call the Underscore.js helper method **_.template()**.

The **apple-item.tpl.js** file:

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

The **apple-home.tpl.js** file:

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

The **apple-spinner.tpl.js** file:

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

The **apple.tpl.js** file:

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

Try to start the application now. The full code is under the rpjs/backbone/refactor[12] folder.

As you can see in the previous example, we used global scoped variables (without the keyword `window`).

Warning

Be careful when you introduce a lot of variables into the global namespace (window keyword). There might be conflicts and other unpredictable consequences. For example, if you wrote an open source library and other developers started using the methods and properties directly, instead of using the interface, what happens later when you decide to finally remove/deprecate those global leaks? To prevent this, properly written libraries and applications use JavaScript closures[13].

Example of using closure and a global variable module definition:

```
(function() {
  var apple= function() {
  ...//do something useful like return apple object
  };
  window.Apple = apple;
}())
```

Or in case when we need to access the app object (which creates a dependency on that object):

[12] https://github.com/azat-co/rpjs/tree/master/backbone/refactor
[13] https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Closures

```
(function() {
  var app = this.app;
  //equivalent of window.appliation
  //in case we need a dependency (app)
  this.apple = function() {
  ...//return apple object/class
  //use app variable
  }
  // eqivalent of window.apple = function(){...};
}())
```

As you can see, we've created the function and called it immediately while also wrapping everything in parentheses `()`.

4.6 AMD and Require.js for Development

AMD allows us to organize development code into modules, manage dependencies, and load them asynchronously. This article does a great job at explaining why AMD is a good thing: WHY AMD?[14]

Start your local HTTP server, e.g., MAMP[15].

Let's enhance our code by using the Require.js library.

Our **index.html** will shrink even more:

```
<!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>
```

We only included libraries and the single JavaScript file with our application. This file has the following structure:

[14] http://requirejs.org/docs/whyamd.html
[15] http://www.mamp.info/en/index.html

```
require([...],function(...){...});
```

Or in a more explanatory way:

```
require([
  'name-of-the-module',
  ...
  'name-of-the-other-module'
  ],function(referenceToModule, ..., referenceToOtherModule){
  ...//some useful code
  referenceToModule.someMethod();
});
```

Basically, we tell a browser to load the files from the array of filenames — first parameter of the `require()` function — and then pass our modules from those files to the anonymous callback function (second argument) as variables. Inside of the main function (anonymous callback) we can use our modules by referencing those variables. Therefore, our **apple-app.js** metamorphoses into:

```
  require([
    'apple-item.tpl', //can use shim plugin
    '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",
```

```
        url: "img/gala.jpg"
      }
    ];
    var app;
    var router = Backbone.Router.extend({
    //check if need to be required
      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);

      }
    });

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

We put all of the code inside the function which is a second argument of `require()`, mentioned modules by their filenames, and used dependencies via corresponding parameters. Now we should define the module itself. This is how we can do it with the `define()` method:

```
define([...],function(...){...})
```

The meaning is similar to the `require()` function: dependencies are strings of filenames (and paths) in the array which is passed as the first argument. The second argument is the main function that accepts other libraries as parameters (the order of parameters and modules in the array is important):

```
define(['name-of-the-module'],function(nameOfModule){
  var b = nameOfModule.render();
  return b;
})
```

Note

There is no need to append **.js** to filenames. Require.js does it automatically. Shim plugin is used for importing text files such as HTML templates.

Let's start with the templates and convert them into the Require.js modules.

The new **apple-item.tpl.js** file:

```
define(function() {
  return '\
            <a href="#apples/<%=name%>" target="_blank">\
           <%=name%>\
           </a> <a class="add-to-cart" href="#">buy</a>\
           '
});
```

The **apple-home.tpl** file:

```
define(function(){
  return 'Apple data: \
        <ul class="apples-list">\
        </ul>\
        <div class="cart-box"></div>';
});
```

The **apple-spinner.tpl.js** file:

```
define(function(){
  return '<img src="img/spinner.gif" width="30"/>';
});
```

The **apple.tpl.js** file:

```
define(function(){
  return '<figure>\
          <img src="<%= attributes.url %>"/>\
          <figcaption><%= attributes.name %></figcaption>\
        </figure>';
 });
```

The **apple-item.view.js** file:

```
define(function() {
  return '\
             <a href="#apples/<%=name%>" target="_blank">\
            <%=name%>\
            </a> <a class="add-to-cart" href="#">buy</a>\
            '
});
```

In the **apple-home.view.js** file, we need to declare dependencies on apple-home.tpl and apple-item.view.js files:

```
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; //so we can use view inside of closure
        this.collection.each(function(apple){
          var appleSubView = new appleItemView({model:apple});
          // create subview with model apple
          appleSubView.render();
          // compiles tempalte and single apple data
          $(view.listEl).append(appleSubView.$el);
          //append jQuery object from
```

```
          //single apple to apples-list DOM element
        });
      }
    });
})
```

The **apple.view.js** file depends on two templates:

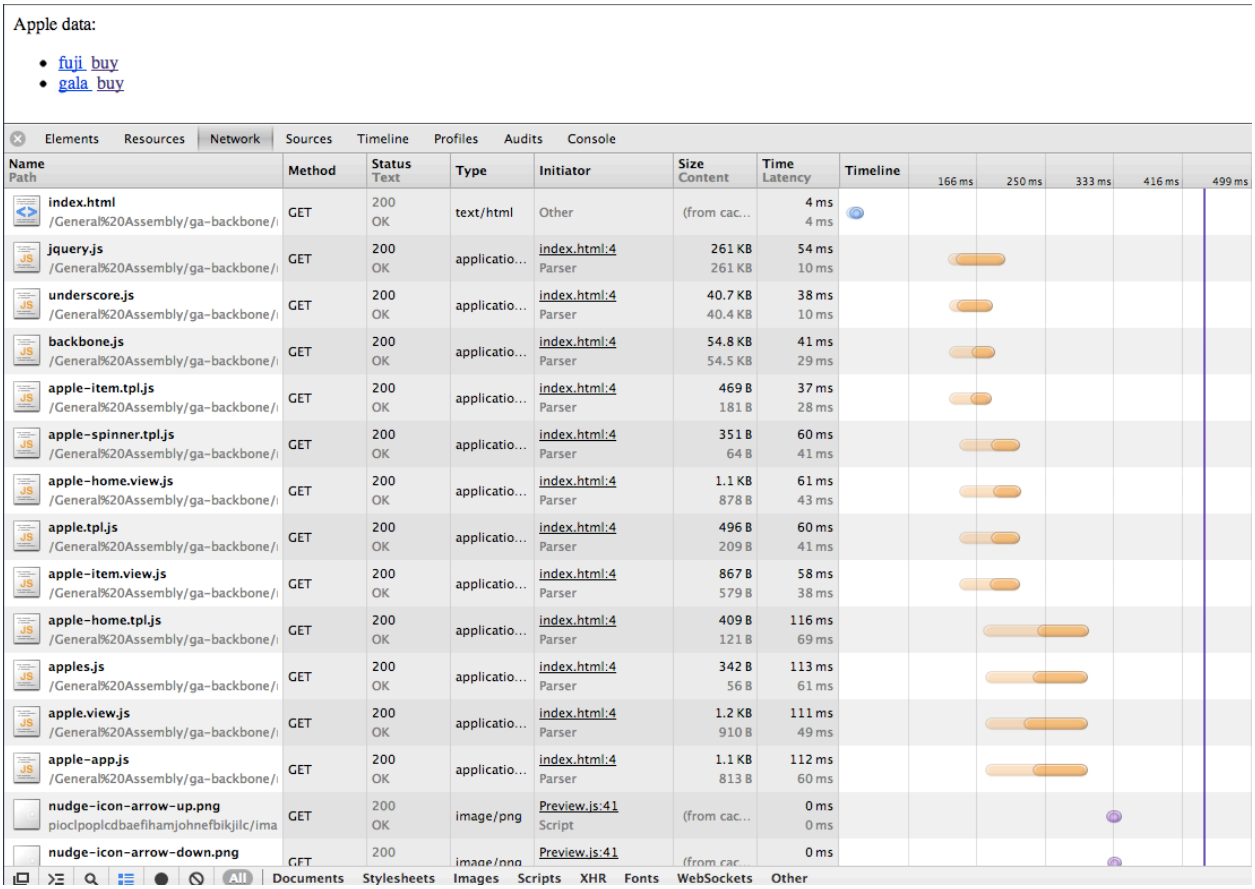
```
define([
  'apple.tpl',
  'apple-spinner.tpl'
],function(appleTpl,appleSpinnerTpl){
  return  Backbone.View.extend({
    initialize: function(){
      this.model = new (Backbone.Model.extend({}));
      this.model.on('change', this.render, this);
      this.on('spinner',this.showSpinner, this);
    },
    template: _.template(appleTpl),
    templateSpinner: appleSpinnerTpl,
    loadApple:function(appleName){
      this.trigger('spinner');
      var view = this;
      //we'll need to access that inside of a closure
      setTimeout(function(){
      //simulates real time lag when
      //fetching data from the remote server
        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);
    }
  });
});
```

The **apples.js** file:

```
define(function(){
    return Backbone.Collection.extend({})
});
```

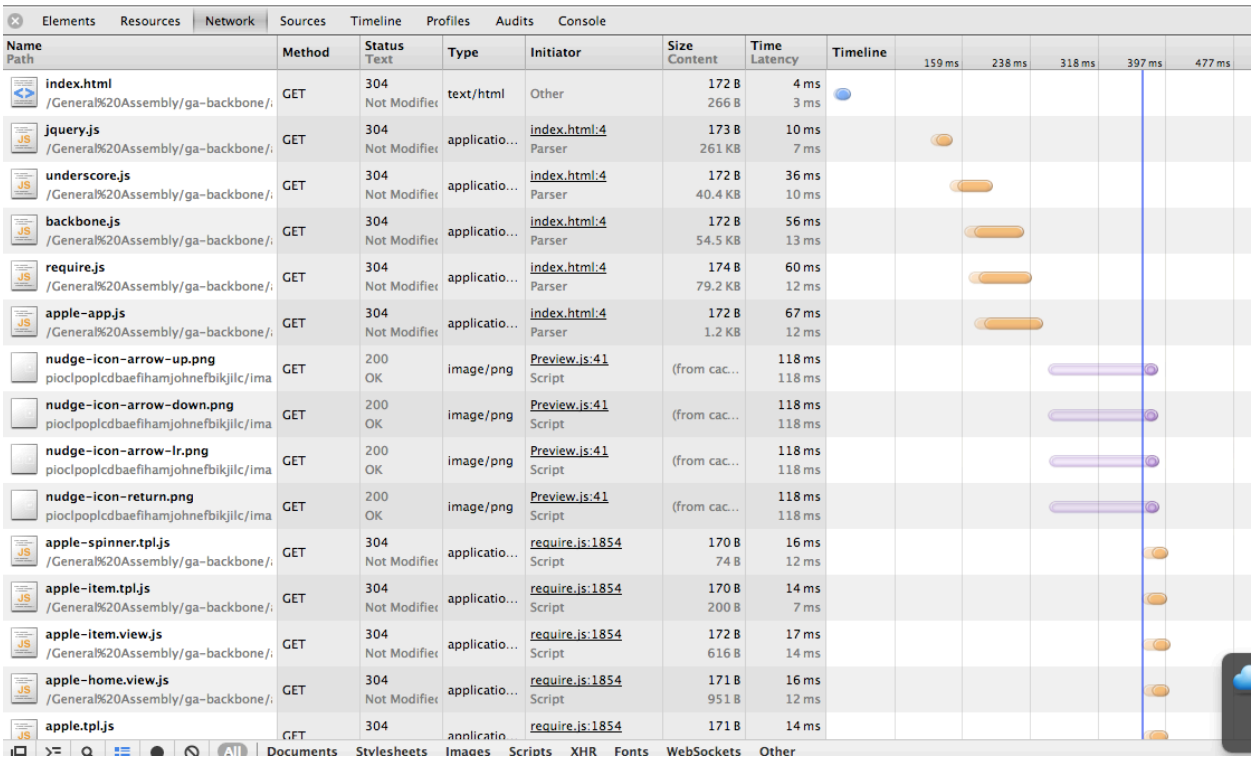
I hope you can see the pattern by now. All of our code is split into the separate files based on the logic (e.g., view class, collection class, template). The main file loads all of the dependencies with the `require()` function. If we need some module in a non-main file, then we can ask for it in the `define()` method. Usually, in modules we want to return an object, e.g., in templates we return strings and in views we return Backbone View classes/objects.

Try launching the example under the rpjs/backbone/amd[16] folder. Visually, there shouldn't be any changes. If you open the Network tab in the Developers Tool, you can see a difference in how the files are loaded. The old rpjs/backbone/refactor/index.html[17] file loads our JS scripts in a serial manner while the new the new rpjs/backbone/amd/index.html[18] file loads them in parallel.



The old rpjs/backbone/refactor/index.html file

[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



The new rpjs/backbone/amd/index.html file

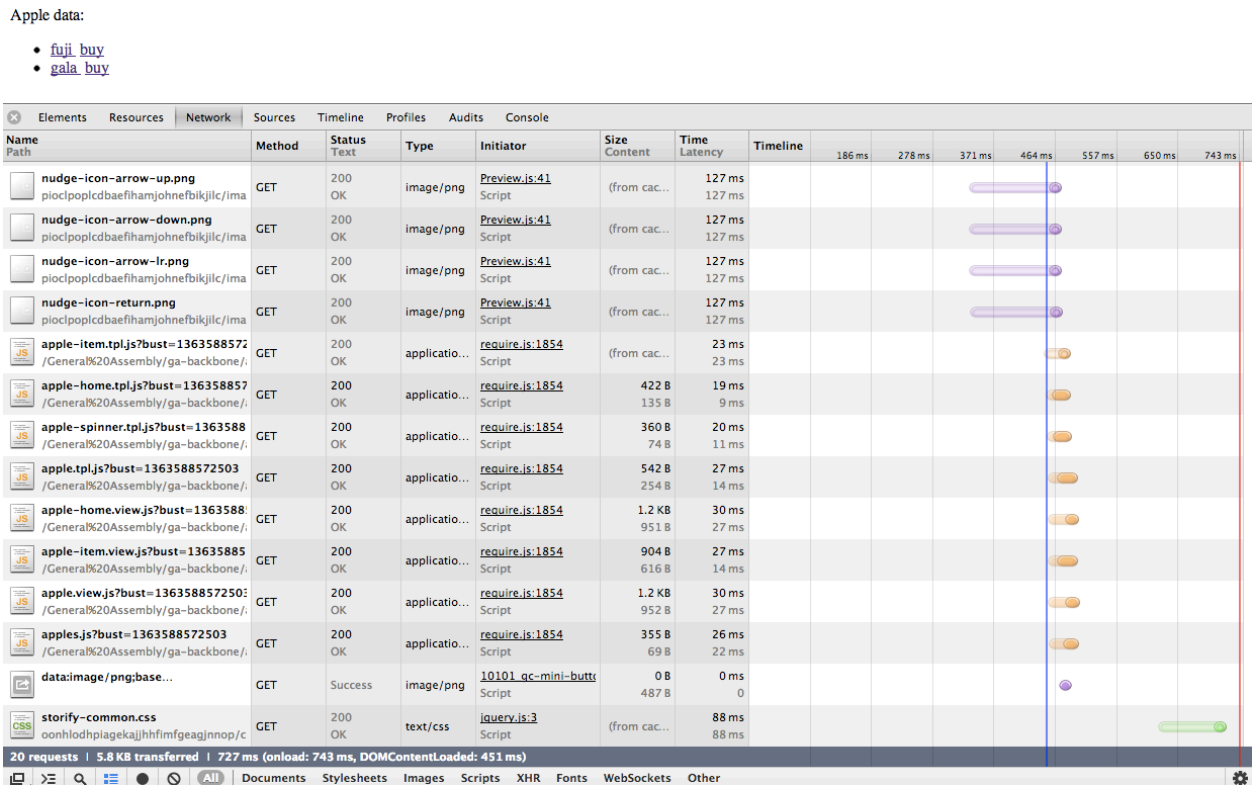
Require.js has a lot of configuration options which are defined through `requirejs.config()` call in a top level of an HTML page. More information can be found at requirejs.org/docs/api.html#config[19].

Let's add a bust parameter to our example. The bust argument will be appended to the URL of each file preventing a browser from caching the files. Perfect for development and terrible for production. :-)

Add this to the **apple-app.js** file in front of everything else:

```
requirejs.config({
  urlArgs: "bust=" +  (new Date()).getTime()
});
require([
...
```

[19] http://requirejs.org/docs/api.html#config



Network Tab with bust parameter added

Please note that each file request now has status 200 instead of 304 (not modified).

4.7 Require.js for Production

We'll use the Node Package Manager (NPM) to install the **requirejs** library (it's not a typo; there's no period in the name). In your project folder, run this command in a terminal:

```
$ npm install requirejs
```

Or add `-g` for global installation:

```
$ npm install -g requirejs
```

Create a file **app.build.js**:

```
({
    appDir: "./js",
    baseUrl: "./",
    dir: "build",
    modules: [
        {
            name: "apple-app"
        }
    ]
})
```

Move the script files under the `js` folder (appDir property). The builded files will be placed under the `build` folder (dir parameter). For more information on the build file, check out this *extensive* example with comments: https://github.com/jrburke/r.js/blob/master/build/example.build.js.

Now everything should be ready for building one gigantic JavaScript file, which will have all of our dependencies/modules:

```
$ r.js -o app.build.js
```

or

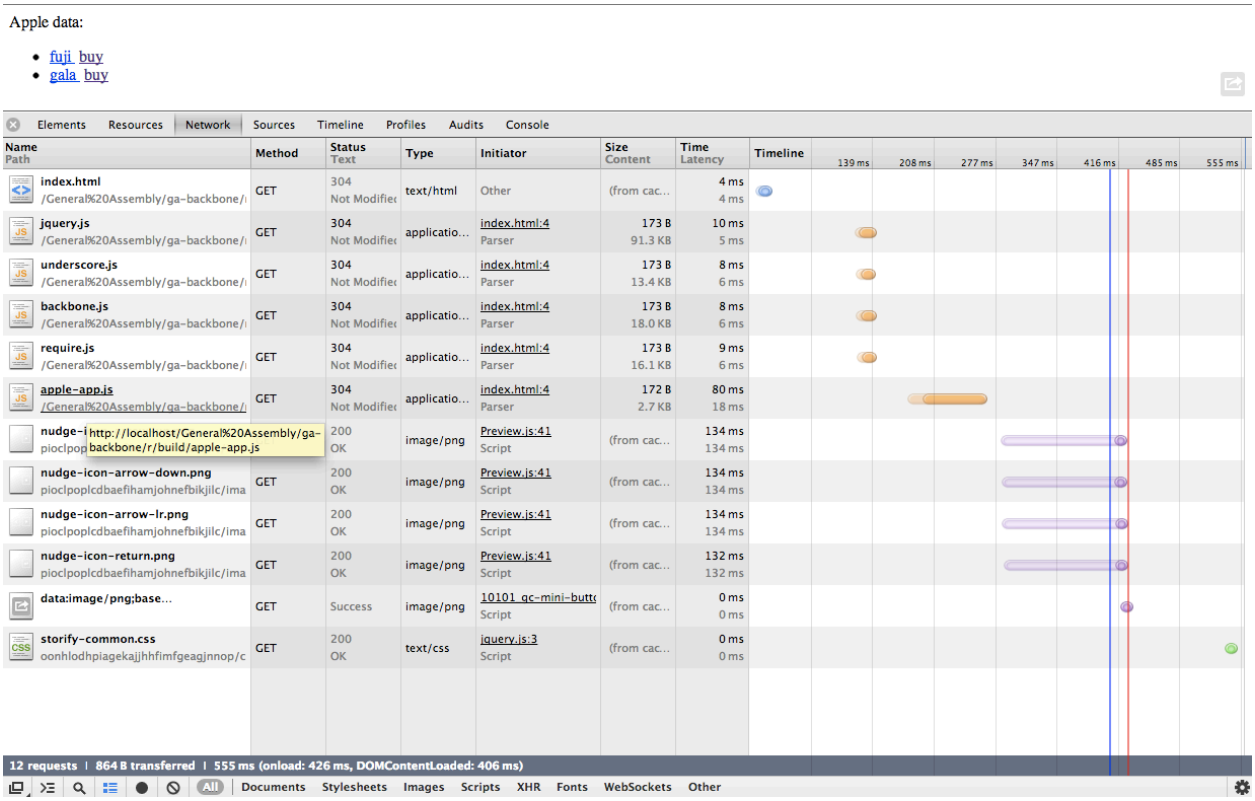
```
$ node_modules/requirejs/bin/r.js -o app.build.js
```

You should get a list of the r.js processed files.

```
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
```

A list of the r.js processed files.

Open **index.html** from the build folder in a browser window, and check if the Network Tab shows any improvement now with just one request/file to load.



Performance improvement with one request/file to load.

For more information, check out the official r.js documentation at requirejs.org/docs/optimization.html[20].

The example code is available under the rpjs/backbone/r[21] and rpjs/backbone/r/build[22] folders.

For uglification of JS files (decreases the files' sizes), we can use the Uglify2[23] module. To install it with NPM, use:

```
$ npm install uglify-js
```

Then update the **app.build.js** file with the `optimize: "uglify2"` property:

[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"
        }
    ]
})
```

Run r.js with:

```
$ node_modules/requirejs/bin/r.js -o app.build.js
```

You should get something like this:

```
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(){});
```

Note

The file is not formatted on purpose to show how Uglify2 works. Without the line break escape symbols, the code is on one line. Also notice that variables and objects' names are shortened.

4.8 Super Simple Backbone Starter Kit

To jump-start your Backbone.js development, consider using Super Simple Backbone Starter Kit[24] or similar projects:

- Backbone Boilerplate[25]
- Sample App with Backbone.js and Twitter Bootstrap[26]
- More Backbone.js tutorials 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 and Parse.com

Summary: illustration the Backbone.js uses with Parse.com and its JavaScript SDK on the modified Chat app; suggested list of additional features for the app.

"Java is to JavaScript what Car is to Carpet." — Chris Heilmann[1]

If you've written some complex client-side applications, you might have found that it's challenging to maintain the spaghetti code of JavaScript callbacks and UI events. Backbone.js provides a lightweight yet powerful way to organize your logic into a Model-View-Controller (MVC) type of structure. It also has nice features like URL routing, REST API support, event listeners and triggers. For more information and step-by-step examples of building Backbone.js applications from scratch, please refer to the chapter *Intro to Backbone.js*.

You can download the Backbone.js library at backbonejs.org[2]. Then, after you included it just like any other JavaScript file in the head (or body) of your main HTML file, you'll be able to access the *Backbone* class. For example, to create the router:

```
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();
    this.homeView.render();
  },
  signup: function() {
    ...
```

[1] http://christianheilmann.com/
[2] http://backbonejs.org

```
  }
});
```

View, Models and Collections are created the same way:

```
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({
  ...
});
```

For a more details on Backbone.js, please refer to the *Intro to Backbone.js* chapter.

5.1 Chat with Parse.com: JavaScript SDK and Backbone.js version

It's easy to see that if we keep adding more and more buttons like "DELETE", "UPDATE" and other functionalities, our system of asynchronous callback will grow more complicated. And we'll have to know when to update the view, i.e., the list of messages, based on whether or not there were changes to the data. The Backbone.js Model-View-Controller (MVC) framework can be used to make complex applications more manageable and easier to maintain.

If you felt comfortable with the previous example, let's build upon it with the use of the Backbone.js framework. Here we'll go step by step, creating a Chat application using Backbone.js and Parse.com

JavaScript SDK. If you feel familiar enough with it, you could download the Super Simple Backbone Starter Kit at github.com/azat-co/super-simple-backbone-starter-kit[3]. Integration with Backbone.js will allow for a straightforward implementation of user actions by binding them to asynchronous updates of the collection.

The application is available under rpjs/sdk[4], but again you are encouraged to start from scratch and try to write your own code using the example only as a reference.

Here is the structure of Chat with Parse.com, JavaScript SDK and Backbone.js version:

```
/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
```

Create a folder; in the folder create an **index.html** file with the following content skeleton:

```
<!DOCTYPE html>
<html lang="en">
  <head>
  ...
  </head>
  <body>
  ...
  </body>
</html>
```

Download the necessary libraries or hot-link them from Google API. Now include JavaScript libraries and Twitter Bootstrap stylesheets into the head element along with other important but not required *meta* elements.

[3] http://github.com/azat-co/super-simple-backbone-starter-kit
[4] https://github.com/azat-co/rpjs/tree/master/sdk

```
<head>
  <meta charset="utf-8" />
  <title></title>
  <meta name="description" content="" />
  <meta name="author" content="" />
```

We need this for responsive behavior:

```
  <meta name="viewport"
    content="width=device-width, initial-scale=1.0" />
```

Hot-linked jQuery from Google API:

```
  <script type="text/javascript"
src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js">
  </script>
```

Nice to have Twitter Bootstrap plug-ins:

```
  <script type="text/javascript" src="js/bootstrap.min.js"></script>
```

Parse JavaScript SDK is hot-linked from Parse.com CDN:

```
  <script type="text/javascript"
    src="http://www.parsecdn.com/js/parse-1.0.5.min.js">
  </script>
```

Twitter Bootstrap CSS inclusion:

```
  <link type="text/css"
    rel="stylesheet"
    href="css/bootstrap.min.css" />
  <link type="text/css"
    rel="stylesheet"
    href="css/bootstrap-responsive.min.css" />
```

Our JS application inclusion:

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

Populate the **body** element with Twitter Bootstrap scaffolding (more about it in the *Basics* chapter):

```
<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>
```

Create an **app.js** file and put Backbone.js views inside:

- headerView: menu and app-common information
- footerView: copyrights and contact links
- homeView: home page content

We use Require.js syntax and shim plugin for HTML templates:

```
require([
'libs/text!header.html',
'libs/text!home.html',
'libs/text!footer.html'], function (
 headerTpl,
 homeTpl,
 footerTpl) {
```

The application router with a single index route:

```
var ApplicationRouter = Backbone.Router.extend({
  routes: {
    "": "home",
    "*actions": "home"
  },
```

Before we do anything else, we can initialize views which are going to be used across the app:

```
  initialize: function() {
    this.headerView = new HeaderView();
    this.headerView.render();
    this.footerView = new FooterView();
    this.footerView.render();
  },
```

This code takes care of the home route:

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

The header Backbone View is attached to the `#header` element and uses the **headerTpl** template:

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

To render the HTML, we use the `jQuery.html()` function:

```
    FooterView = Backbone.View.extend({
      el: "#footer",
      template: footerTpl,
      render: function() {
        this.$el.html(_.template(this.template));
      }
    });
```

The home Backbone View definition uses the `#content` DOM element:

```
    HomeView = Backbone.View.extend({
      el: "#content",
      // template: "home.html",
      template: homeTpl,
      initialize: function() {
      },
      render: function() {
        $(this.el).html(_.template(this.template));
      }
    });
```

To start an app, we create a new instance and call `Backbone.history.start()`:

```
    app = new ApplicationRouter();
    Backbone.history.start();
  });
```

The full code of the **app.js** file:

```
require([
'libs/text!header.html',
//example of a shim plugin use
'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",
    // template: "home.html",
    template: homeTpl,
    initialize: function() {
    },
    render: function() {
      $(this.el).html(_.template(this.template));
    }
  });
  app = new ApplicationRouter();
  Backbone.history.start();
});
```

The code above displays templates. All views and routers are inside, requiring the module to make sure that the templates are loaded before we begin to process them.

Here is what **home.html** looks like:

- A table of messages
- Underscore.js logic to output rows of the table
- A new message form

Let's use the Twitter Bootstrap library structure (with its responsive components) by assigning `row-fluid` and `span12` classes:

```
<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>
```

This part has Underscore.js template instructions, which are just some JS code wrapped in `<%` and `%>` marks. `_.each()` is an iteration function from the UnderscoreJS library (underscorejs.org/#each[5]), which does exactly what it sounds like — iterates through elements of an object/array:

```
      <% 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>
```

For the new message form, we also use the `row-fluid` class and then add `input` elements:

[5] http://underscorejs.org/#each

```
<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>
```

The full code of the **home.html** template file:

```
<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>
```

Now we can add the following components to:

- Parse.com collection,
- Parse.com model,
- Send/add message event,
- Getting/displaying messages functions.

Backbone compatible model object/class from Parse.com JS SDK with a mandatory **className** attribute (this is the name of the collection that will appear in the Data Browser of the Parse.com web interface):

```
Message = Parse.Object.extend({
        className: "MessageBoard"
});
```

Backbone compatible collection object from Parse.com JavaScript SDK that points to the model:

```
MessageBoard = Parse.Collection.extend ({
        model: Message
});
```

The home view needs to have click event listener on the “SEND” button:

```
HomeView = Backbone.View.extend({
        el: "#content",
        template: homeTpl,
        events: {
                "click #send": "saveMessage"
        },
```

When we create homeView, let's also create a collection and attach event listeners to it:

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

The definition of `saveMessage()` calls for the "SEND" button click event:

```
        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
    ));
        }
```

The end result of our manipulations in **app.js** might look something like this:

```
/*
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"
    },
    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() {
      $(this.el).html(_.template(this.template));
    }
  });

  FooterView = Backbone.View.extend({
    el: "#footer",
    template: footerTpl,
    render: function() {
      this.$el.html(_.template(this.template));
    }
  });
  Message = Parse.Object.extend({
    className: "MessageBoard"
  });
  MessageBoard = Parse.Collection.extend ({
    model: Message
  });

  HomeView = Backbone.View.extend({
    el: "#content",
    template: homeTpl,
    events: {
      "click #send": "saveMessage"
    },

    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);
      })
    },
```

```
    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
      ));
    }
  });

  app = new ApplicationRouter();
  Backbone.history.start();
});
```

The full source code of the Backbone.js and Parse.com Chat application is available under rpjs/sdk[6].

5.2 Deploying Chat to PaaS

Once you are comfortable that your front-end application works well locally, with or without a local HTTP server like MAMP or XAMPP, deploy it to Windows Azure or Heroku. In-depth deployment instructions are described in the *jQuery and Parse.com* chapter.

5.3 Enhancing Chat

In the last two examples, Chat had very basic functionality. You could enhance the application by adding more features.

Additional features for **intermediate** level developers:

[6] https://github.com/azat-co/rpjs/tree/master/sdk

- Sort the list of messages through the *updateAt* attribute before displaying it
- Add a "Refresh" button to update the list of messages
- Save the username after the first message entry in a run-time memory or in a session
- Add an up-vote button next to each message, and store the votes
- Add a down-vote button next to each message, and store the votes

Additional features for **advanced** level developers:

- Add a User collection
- Prevent the same user from voting multiple times
- Add user sign-up and log-in actions by using Parse.com functions
- Add a *Delete Message* button next to each message created by a user
- Add an *Edit Message* button next to each message created by a user

III Back-End Prototyping

6 Node.js and MongoDB

Summary: exhibition of the "Hello World" application in Node.js, list of some of its the most important core modules, NPM workflow, detailed commands for deployment of Node.js apps to Heroku and Windows Azure; MongoDB and its shell, run-time and database Chat applications; example of a test-driven development practice.

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." — Martin Fowler[1]

6.1 Node.js

6.1.1 Building "Hello World" in Node.js

To check if you have Node.js installed on your computer, type and execute this command in your terminal:

```
$ node -v
```

As of this writing, the latest version is 0.8.1. If you don't have Node.js installed, or if your version is behind, you can download the latest version at nodejs.org/#download[2].

As usual, you could copy example code from rpjs/hello[3] or write your own program from scratch. If you wish to do the latter, create a folder **hello** for your "Hello World" Node.js application. Then create file a **server.js** and line by line type the code below.

This will load the core **http** module for the server (more on the modules later):

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

We'll need a port number for our Node.js server. To get it from the environment, or assign 1337 if the environment is not set, use:

```
var port = process.env.PORT || 1337;
```

This will create a server and a call-back function will contain the response handler code:

[1] http://en.wikipedia.org/wiki/Martin_Fowler
[2] http://nodejs.org/#download
[3] https://github.com/azat-co/rpjs/tree/master/hello

```
var server = http.createServer(function (req, res) {
```

To set the right header and status code, use:

```
  res.writeHead(200, {'Content-Type': 'text/plain'});
```

To output "Hello World" with the line end symbol, use:

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

To set a port and display the address of the server and the port number, use:

```
server.listen(port, function() {
  console.log('Server is running at %s:%s ',
    server.address().address, server.address().port);
});
```

From the folder in which you have *server.js*, launch in your terminal the following command:

```
$ node server.js
```

Open localhost:1337[4] or 127.0.0.1:1337[5] or any other address you see in the terminal as a result of `console.log()` function, and you should see "Hello World" in a browser. To shut down the server, press Control + C.

Note

The name of the main file could be different from server.js, e.g., index.js or app.js. In case you need to launch the **app.js** file, just use `$ node app.js`.

6.1.2 Node.js Core Modules

Unlike other programming technologies, Node.js doesn't come with a heavy standard library. The core modules of node.js are a bare minimum and the rest can be cherry-picked via the Node Package Manager (NPM) registry. The main core modules, classes, methods and events include:

- http[6]
- util[7]

[4] http://localhost:1337/
[5] http://127.0.0.1:1337/
[6] http://nodejs.org/api/http.html#http_http
[7] http://nodejs.org/api/util.html

- querystring[8]
- url[9]
- fs[10]

http[11]

This is the main module responsible for Node.js HTTP server. Here are the main methods:

- `http.createServer()`: returns a new web server object
- `http.listen()`: begins accepting connections on the specified port and hostname
- `http.createClient()`: node app can be a client and make requests to other servers
- `http.ServerRequest()`: incoming requests are passed to request handlers
  - **data**: emitted when a piece of the message body is received
  - **end**: emitted exactly once for each request
  - `request.method()`: the request method as a string
  - `request.url()`: request URL string
- `http.ServerResponse()`: this object is created internally by an HTTP server — not by the user, and used as an output of request handlers
  - `response.writeHead()`: sends a response header to the request
  - `response.write()`: sends a response body * `response.end()`: sends and ends a response body

util[12]

This module provides utilities for debugging. Some of the methods include:

- `util.inspect()`: Return a string representation of an object, which is useful for debugging

querystring[13]

This module provides utilities for dealing with query strings. Some of the methods include:

- `querystring.stringify()`: Serialize an object to a query string
- `querystring.parse()`: Deserialize a query string to an object

url[14]

This module has utilities for URL resolution and parsing. Some of the methods include:

- `parse()`: Take a URL string, and return an object

[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
[13] http://nodejs.org/api/querystring.html
[14] http://nodejs.org/api/url.html

fs[15]

fs handles file system operations such as reading and writing to/from files. There are synchronous and asynchronous methods in the library. Some of the methods include:

- `fs.readFile()`: reads file asynchronously
- `fs.writeFile()`: writes data to file asynchronously

There is no need to install or download core modules. To include them in your application, all you need is to follow the syntax:

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

The lists of non-core modules can be found at:

- npmjs.org[16]: Node Package Manager registry
- GitHub hosted list[17]: list of Node.js modules maintainded by Joyent
- nodetoolbox.com[18]: registry based on stats
- Nipster[19]: NPM search tool for Node.js
- Node Tracking[20]: registry based on GitHub stats

If you would like to know how to code your own modules, take a look at the article Your first Node.js module[21].

6.1.3 Node Package Manager

Node Package Manager, or NPM, manages dependencies and installs modules for you. Node.js installation comes with NPM, whose website is npmjs.org[22].

package.json contains meta information about our Node.js application such as a version number, author name and, most importantly, what dependencies we use in the application. All of that information is in the JSON formatted object, which is read by NPM.

If you would like to install packages and dependencies specified in **package.json**, type:

```
$ npm install
```

A typical **package.json** file might look like this:

[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/
[22] http://npmjs.org/

```
{
   "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"
   }
}
```

To update a package to its current latest version, or the latest version that is allowable by the version specification defined in **package.json**, use:

```
$ npm update name-of-the-package
```

Or for single module installation:

```
$ npm install name-of-the-package
```

The only module used in the examples — and which does not belong to the core Node.js package — is **mongodb**. We'll install it later in the book.

Heroku will need *package.json* to run NPM on the server.

For more information on NPM, take a look at the article Tour of NPM[23].

6.1.4 Deploying "Hello World" to PaaS

For Heroku and Windows Azure deployment, we'll need a Git repository. To create it from the root of your project, type the following command in your terminal:

[23] http://tobyho.com/2012/02/09/tour-of-npm/

```
$ git init
```

Git will create a hidden .**git** folder. Now we can add files and make the first commit:

```
$ git add .
$ git commit -am "first commit"
```

Tip

To view hidden files on the Mac OS X Finder app, execute this command in a terminal window: `defaults write com.apple.finder AppleShowAllFiles -bool true`. To change the flag back to hidden: `defaults write com.apple.finder AppleShowAllFiles -bool false`.

6.1.5 Deploying to Windows Azure

In order to deploy our "Hello World" application to Windows Azure, we must add Git **remote**. You could copy the URL from Windows Azure Portal, under Web Site, and use it with this command:

```
$ git remote add azure yourURL
```

Now we should be able to make a push with this command:

```
$ git push azure master
```

If everything went okay, you should see success logs in the terminal and "Hello World" in the browser of your Windows Azure Web Site URL.

To push changes, just execute:

```
$ git add .
$ git commit -m "changing to hello azure"
$ git push azure master
```

A more meticulous guide can be found in the tutorial Build and deploy a Node.js web site to Windows Azure[24].

6.1.6 Deploying to Heroku

For Heroku deployment, we need to create 2 extra files: **Procfile** and **package.json**. You could get the source code from rpjs/hello[25] or write your own one.

The structure of the "Hello World" application looks like this:

[24] http://www.windowsazure.com/en-us/develop/nodejs/tutorials/create-a-website-(mac)/
[25] https://github.com/azat-co/rpjs/tree/master/hello

```
/hello
  -package.json
  -Procfile
  -server.js
```

Procfile is a mechanism for declaring what commands are run by your application's dynos on the Heroku platform. Basically, it tells Heroku what processes to run. Procfile has only one line in this case:

```
web: node server.js
```

For this example, we keep **package.json** simple:

```
{
  "name": "node-example",
  "version": "0.0.1",
  "dependencies": {
  },
  "engines": {
    "node": ">=0.6.x"
  }
}
```

After we have all of the files in the project folder, we can use Git to deploy the application. The commands are pretty much the same as with Windows Azure except that we need to add Git remote, and create Cedar stack with:

```
$ heroku create
```

After it's done, we push and update with:

```
$ git push heroku master
$ git add .
$ git commit -am "changes"
$ git push heroku master
```

If everything went okay, you should see success logs in the terminal and "Hello World" in the browser of your Heroku app URL.

6.2 Chat: Run-Time Memory Version

The first version of the Chat back-end application will store messages only in run-time memory storage for the sake of KISS[26]. That means that each time we start/reset the server, the data will be lost.

We'll start with a simple test case first to illustrate the Test-Driven Development approach. The full code is available under the rpjs/test[27] folder.

[26] http://en.wikipedia.org/wiki/KISS_principle
[27] https://github.com/azat-co/rpjs/tree/master/test

6.3 Test Case for Chat

We should have two methods:

1. Get all of the messages as an array of JSON objects for the GET `/message` endpoint using the `getMessages()` method
2. Add a new message with properties **name** and **message** for POST `/messages` route via the `addMessage()` function

We'll start by creating an empty **mb-server.js** file. After it's there, let's switch to tests and create the **test.js** file with the following content:

```
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"));
```

Please keep in mind that, this is a very simplified comparison of strings and not JavaScript objects. So every space, quote and case matters. You could make the comparison "smarter" by parsing a string into a JSON object with:

```
JSON.parse(str);
```

For testing our assumptions, we use core the Node.js module assert[28]. It provides a bunch of useful methods like `equal()`, `deepEqual()`, etc.

More advanced libraries include alternative interfaces with TDD and/or BDD approaches:

- Should[29]
- Expect[30]

[28] http://nodejs.org/api/assert.html
[29] https://github.com/visionmedia/should.js/
[30] https://github.com/LearnBoost/expect.js/

For more Test-Driven Development and cutting-edge automated testing, you could use the following libraries and modules:

- Mocha[31]
- NodeUnit[32]
- Jasmine[33]
- Vows[34]
- Chai[35]

You could copy the "Hello World" script into the **mb-server.js** file for now or even keep it empty. If we run **test.js** by the terminal command:

```
$ node test.js
```

We should see an error. Probably something like this one:

```
TypeError: Object #<Object> has no method 'getMessages'
```

That's totally fine, because we haven't written `getMessages()` method yet. So let's do it and make our application more useful by adding two new methods: to get the list of the messages for Chat and to add a new message to the collection.

mb-server.js file with global `exports` object:

```
exports.getMessages = function() {
  return JSON.stringify(messages);
};
exports.addMessage = function (data){
  messages.push(querystring.parse(data));
  return JSON.stringify(querystring.parse(data));
};
```

So far, nothing fancy, right? To store the list of messages, we'll use an array:

[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/

```
var messages=[];
//this array will hold our messages
messages.push({
  "name":"John",
  "message":"hi"
});
//sample message to test list method
```

Tip

Generally, fixtures like dummy data belong to the test/spec files and not to the main application codebase.

Our server code will look slightly more interesting. For getting the list of messages, according to REST methodology, we need to make a GET request. For creating/adding a new message, it should be a POST request. So in our createServer object, we should add `req.method()` and `req.url()` to check for an HTTP request type and a URL path.

Let's load the http module:

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

We'll need some of the handy functions from the util and querysting modules (to serialize and deserialize objects and query strings):

```
var util = require('util');
var querystring = require('querystring');
```

To create a server and expose it to outside modules (i.e., test.js):

```
exports.server=http.createServer(function (req, res) {
```

Inside of the request handler callback, we should check if the request method is POST and the URL is `messages/create.json`:

```
  if (req.method == "POST" &&
    req.url == "/messages/create.json") {
```

If the condition above is true, we add a message to the array. However, **data** must be converted to a string type (with encoding UTF-8) prior to the adding, because it is a type of Buffer:

```
    var message = '';
    req.on('data', function(data, message){
      console.log(data.toString('utf-8'));
      message = exports.addMessage(data.toString('utf-8'));
```

These logs will help us to monitor the server activity in the terminal:

```
    })
    console.log(util.inspect(message, true, null));
    console.log(util.inspect(messages, true, null));
```

The output should be in a text format with a status of 200 (okay):

```
    res.writeHead(200, {
      'Content-Type': 'text/plain'
     });
```

We output a message with a newly created object ID:

```
    res.end(message);
  }
```

If the method is GET and the URL is `/messages/list.json` output a list of messages:

```
  if (req.method == "GET" &&
    req.url == "/messages/list.json") {
```

Fetch a list of messages:

```
    var body = exports.getMessages();
```

The response body will hold our output:

```
    res.writeHead(200, {
      'Content-Length': body.length,
      'Content-Type': 'text/plain'
    });
    res.end(body);
  }
  else {
```

This sets the right header and status code:

```
    res.writeHead(200, {
      'Content-Type': 'text/plain'
    });
```

In case it's neither of the two endpoints above, we output a string with a line-end symbol:

```
    res.end('Hello World\n');
  };
  console.log(req.method);


}).listen(1337, "127.0.0.1");
```

Now, we should set a port and IP address of the server:

```
console.log('Server running at http://127.0.0.1:1337/');
```

We expose methods for the unit testing in **test.js** (exports keyword), and this function returns an array of messages as a string/text:

```
exports.getMessages = function() {
  return JSON.stringify(messages);
};
```

`addMessage()` converts a string into a JavaScript object with the parse/deserializer method from querystring:

```
exports.addMessage = function (data){
  messages.push(querystring.parse(data));
```

Also returning a new message in a JSON-as-a-string format:

```
  return JSON.stringify(querystring.parse(data));
};
```

Here is the full code of **mb-server.js**. It's also available at rpjs/test[36]:

[36] https://github.com/azat-co/rpjs/tree/master/test

```
var http = require('http');
//loads http module
var util = require('util');
//useful functions
var querystring = require('querystring');
//loads querystring module,
//we'll need it to serialize and
//deserialize objects and query strings

var messages=[];
//this array will hold our messages
messages.push({
  "name": "John",
  "message": "hi"
});
//sample message to test list method

exports.server=http.createServer(function (req, res) {
//creates server
  if (req.method == "POST" &&
    req.url == "/messages/create.json") {
    //if method is POST and
    //URL is messages/ add message to the array
    var message = '';
    req.on('data', function(data, message){
      console.log(data.toString('utf-8'));
      message = exports.addMessage(data.toString('utf-8'));
      //data is type of Buffer and
      //must be converted to string
      //with encoding UTF-8 first
      //adds message to the array
    })
    console.log(util.inspect(message, true, null));
    console.log(util.inspect(messages, true, null));
    //debugging output into the terminal
    res.writeHead(200, {
      'Content-Type': 'text/plain'
     });
    //sets the right header and status code
    res.end(message);
    //out put message, should add object id
  }
  if (req.method == "GET" &&
    req.url == "/messages/list.json") {
  //if method is GET and
```

```
  //URL is /messages output list of messages
    var body = exports.getMessages();
    //body will hold our output
    res.writeHead(200, {
      'Content-Length': body.length,
      'Content-Type': 'text/plain'
    });
    res.end(body);
  }
  else {
    res.writeHead(200, {
      'Content-Type': 'text/plain'
    });
    //sets the right header and status code
    res.end('Hello World\n');
  };
  console.log(req.method);
  //outputs string with line end symbol
}).listen(1337, "127.0.0.1");
//sets port and IP address of the server
console.log('Server running at http://127.0.0.1:1337/');

exports.getMessages = function() {
  return JSON.stringify(messages);
  //output array of messages as a string/text
};
exports.addMessage = function (data){
  messages.push(querystring.parse(data));
  //to convert string into
  //JavaScript object we use parse/deserializer
  return JSON.stringify(querystring.parse(data));
  //output new message in JSON as a string
};
```

To check it, go to localhost:1337/messages/list.json[37]. You should see an example message. Alternatively, you could use the terminal command:

```
$ curl http://127.0.0.1:1337/messages/list.json
```

To make the POST request by using a command line interface:

```
curl -d "name=BOB&message=test" http://127.0.0.1:1337/messages/create.json
```

[37] http://localhost:1337/messages/list.json

And you should get the output in a server terminal window and a new message "test" when you refresh localhost:1337/messages/list.json[38]. Needless to say, all three tests should pass.

Your application might grow bigger with more methods, URL paths to parse and conditions. That is where frameworks come in handy. They provide helpers to process requests and other nice things like static file support, sessions, etc. In this example, we intentionally didn't use any frameworks like Express (http://expressjs.com/) or Restify (http://mcavage.github.com/node-restify/). Other notable Node.js frameworks:

- Derby[39]: MVC framework making it easy to write real-time, collaborative applications that run in both Node.js and browsers
- Express.js[40]: the most robust, tested and used Node.js framework
- Restify[41]: lightweight framework for REST API servers
- Sails.js[42]: MVC Node.js framework
- hapi[43]: Node.js framework built on top of Express.js
- Connect[44]: a middleware framework for node, shipping with over 18 bundled middlewares and a rich selection of third-party middleware
- GeddyJS[45]: a simple, structured MVC web framework for Node
- CompoundJS[46] (ex-RailswayJS): Node.JS MVC framework based on ExpressJS
- Tower.js[47]: a full stack web framework for Node.js and the browser
- Meteor[48]: open-source platform for building top-quality web apps in a fraction of the time

Ways to improve the application:

- Improve existing test cases by adding object comparison instead of a string one
- Move the seed data to **test.js** from **mb-server.js**
- Add test cases to support your front-end, e.g., up-vote, user log in
- Add methods to support your front-end, e.g., up-vote, user log in
- Generate unique IDs for each message and store them in a Hash instead of an Array
- Install Mocha and re-factor test.js so it uses this library

So far we've been storing our messages in the application memory, so each time the application is restarted, we lose it. To fix it, we need to add a persistence, and one of the ways is to use a database like MongoDB.

[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

6.4 MongoDB

6.4.1 MongoDB Shell

If you haven't done so already, please install the latest version of MongoDB from mongodb.org/downloads[49]. For more instructions, please refer to the *Setup, Database: MongoDB* section.

Now from the folder where you unpacked the archive, launch the **mongod** service with:

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

You should be able to see information in your terminal and in the browser at localhost:28017[50].

For the MongoDB shell, or **mongo**, launch in a new terminal window (**important!**), and at the same folder this command:

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

You should see something like this, depending on your version of the MongoDB shell:

```
MongoDB shell version: 2.0.6
connecting to: test
```

To test the database, use the JavaScript-like interface and commands **save** and **find**:

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

More detailed step-by-step instructions are available at Setup, Database: MongoDB.

Some other useful MongoDB shell commands:

```
> 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"});
```

[49] http://www.mongodb.org/downloads
[50] http://localhost:28017

A full overview of the MongoDB interactive shell is available at mongodb.org: Overview - The MongoDB Interactive Shell[51].

6.4.2 MongoDB Native Driver

We'll use Node.js Native Driver for MongoDB (https://github.com/christkv/node-mongodb-native) to access MongoDB from Node.js applications. Full documentation is also available at http://mongodb.github.com/node-mongodb-native/api-generated/db.html.

To install MongoDB Native driver for Node.js, use:

```
$ npm install mongodb
```

More details are at http://www.mongodb.org/display/DOCS/node.JS.

Don't forget to include the dependency in the **package.json** file as well:

```
{
  "name": "node-example",
  "version": "0.0.1",
  "dependencies": {
        "mongodb":"",
        ...
  },
  "engines": {
    "node": ">=0.6.x"
  }
}
```

Alternatively, for your own development you could use other mappers, which are available as an extension of the Native Driver:

- Mongoskin[52]: the future layer for node-mongodb-native
- Mongoose[53]: asynchronous JavaScript driver with optional support for modeling
- Mongolia[54]: lightweight MongoDB ORM/driver wrapper
- Monk[55]: Monk is a tiny layer that provides simple yet substantial usability improvements for MongoDB usage within Node.js

This small example will test if we can connect to local MongoDB instance from a Node.js script.

After we installed the library, we can include the **mongodb** library in our **db.js** file:

[51] http://www.mongodb.org/display/DOCS/Overview+-+The+MongoDB+Interactive+Shell
[52] https://github.com/guileen/node-mongoskin
[53] http://mongoosejs.com/
[54] https://github.com/masylum/mongolia
[55] https://github.com/LearnBoost/monk

```
var util = require('util');
var mongodb = require ('mongodb');
```

This is one of the ways to establish connection to the MongoDB server in which the db variable will hold reference to the database at a specified host and port:

```
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, {}));
```

To actually open a connection:

```
db.open(function(e,c){
        //do something with the database here
  // console.log (util.inspect(db));
  console.log(db._state);
  db.close();
});
```

This code snippet is available under the rpjs/db/db.js[56] folder. If we run it, it should output "**connected**" in the terminal. When you're in doubt and need to check the properties of an object, there is a useful method in the **util** module:

```
console.log(util.inspect(db));
```

6.4.3 MongoDB on Heroku: MongoHQ

After you made your application that displays 'connected' work locally, it's time to slightly modify it and deploy to the platform as a service, i.e., Heroku.

We recommend using the MongoHQ add-on[57], which is a part of MongoHQ[58] technology. It provides a browser-based GUI to look up and manipulate the data and collections. More information is available at https://devcenter.heroku.com/articles/mongohq.

Note

You might have to provide your credit card information to use MongoHQ even if you select the free version. You should not be charged, though.

[56] https://github.com/azat-co/rpjs/blob/master/db/db.js
[57] https://addons.heroku.com/mongohq
[58] https://www.mongohq.com/home

In order to connect to the database server, there is a database connection URL (a.k.a. MongoHQ URL/URI), which is a way to transfer all of the necessary information to make a connection to the database in one string.

The database connection string MONGOHQ_URL has the following format:

```
mongodb://user:pass@server.mongohq.com/db_name
```

You could either copy the MongoHQ URL string from the Heroku website (and hard-code it) or get the string from the Node.js **process.env** object:

```
process.env.MONGOHQ_URL;
```

or

```
var connectionUri = url.parse(process.env.MONGOHQ_URL);
```

Tip

The global object process gives access to environment variables via process.env. Those variables conventionally used to pass database host names and ports, passwords, API keys, port numbers, and other system information that shouldn't be hard-coded into the main logic.

To make our code work both locally and on Heroku, we can use the logical OR operator `||` and assign a local host and port if environment variables are undefined:

```
var port = process.env.PORT || 1337;
var dbConnUrl = process.env.MONGOHQ_URL ||
  'mongodb://@127.0.0.1:27017';
```

Here is out updated cross-environment ready **db.js** file:

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

Following the modification of **db.js** by addition of **MONGOHQ_URL**, we can now initialize Git repository, create a Heroku app, add the MongoHQ add-on to it and deploy the app with Git.

Utilize the same steps as in the previous examples to create a new git repository:

```
git init
git add .
git commit -am 'initial commit'
```

Create the Cedar stack Heroku app:

```
$ heroku create
```

If everything went well you should be able to see a message that tell you the new Heroku app name (and URL) along with a message that remote was added. Having remote in your local git is crucial, you can always check a list of remote by:

```
git remote show
```

To install free MongoHQ on the existing Heroku app (add-ons work on a per app basis), use:

```
$ heroku addons:add mongohq:sandbox
```

Or log on to addons.heroku.com/mongohq[59] with your Heroku credentials and choose MongoHQ Free for that particular Heroku app, if you know the name of that app.

If you get **db.js** and modified **db.js** files working, let's enhance by adding a HTTP server, so the 'connected' message will be displayed in the browser instead of the terminal window. To do so, we'll wrap the server object instanciation in a database connection callback:

[59] https://addons.heroku.com/mongohq

```
...
db.open(function(e,c){
        // console.log (util.inspect(db));
        var server = http.createServer(function (req, res) {
    //creates server
          res.writeHead(200, {'Content-Type': 'text/plain'});
    //sets the right header and status code
          res.end(db._state);
    //outputs string with line end symbol
        });
        server.listen(port, function() {
                console.log('Server is running at %s:%s '
    , server.address().address
    , server.address().port);
    //sets port and IP address of the server
        });
        db.close();
});
...
```

The final deployment-ready file **app.js** from 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();
});
```

After the deployment you should be able to open the URL provided by Heroku and see the message 'connected'.

Here is the manual on how to use MongoDB from Node.js code: mongodb.github.com/node-mongodb-native/api-articles/nodekoarticle1.html[61].

Another approach is to use the MongoHQ Module, which is available at github.com/MongoHQ/mongohq-nodejs[62].

This example illustrates a different use of the **mongodb** library by outputting collections and a document count. The full source code from 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

```
    //output all collection names
    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});

    //output a count of all documents found
    documents.count(function(error, count){
      log("  " + count + " documents(s) found");
      log("====================");

      // output the first 5 documents
      documents.toArray(function(error, docs) {
        if(error) throw error;

        docs.forEach(function(doc){
          log(doc);
        });

        // close the connection
        client.close();
      });
    });
  });
});
```

We've used a shortcut for `console.log()` with `var log = console.log;` and `return` as a control flow via `if (!lastCollection) return;`.

6.4.4 BSON

Binary JSON, or BSON, it is a special data type which MongoDB utilizes. It is like to JSON in notation, but has support for additional more sophisticated data types.

Warning

A word of caution about BSON: ObjectId in MongoDB is an equivalent to ObjectID in MongoDB Native Node.js Driver, i.e., make sure to use the proper case. Otherwise you'll get an error. More on the types: ObjectId in MongoDB[64] vs Data Types in MongoDB Native Node.js Drier[65]. Example of Node.js code with `mongodb.ObjectID()`: `collection.findOne({_id: new ObjectID(idString)}, console.log) // ok`. On the other hand, in the MongoDB shell, we employ: `db.messages.findOne({_id:ObjectId(idStr)});`.

6.5 Chat: MongoDB Version

We should have everything set up for writing the Node.js application which will work both locally and on Heroku. The source code is available under rpjs/mongo[66]. The structure of the application is simple:

```
/mongo
  -web.js
  -Procfile
  -package.json
```

This is what *web.js* looks like; first we include our libraries:

```
var http = require('http');
var util = require('util');
var querystring = require('querystring');
var mongo = require('mongodb');
```

Then put out a magic string to connect to 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
```

Note

The URI/URL format contains the optional database name in which our collection will be stored. Feel free to change it to something else, for example 'rpjs' or 'test'.

We put all the logic inside of an open connection in the form of a callback function:

[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);
})
```

Note

We don't have to use additional words after the collection/entity name, i.e., instead of `/messages/list.json` and `/messages/create.json` it's perfectly fine to have just `/messages` for all the HTTP methods such as GET, POST, PUT, DELETE. If you change them in your application code make sure to use the updated CURL commands and front-end code.

To test via CURL terminal commands run:

```
curl http://localhost:5000/messages/list.json
```

Or open your browser at the http://locahost:5000/messages/list.json location.

It should give you an empty array: `[]` which is fine. Then POST a new message:

```
curl  -d "username=BOB&message=test" http://localhost:5000/messages/create.json
```

Now we must see a response containing an ObjectID of a newly created element, for example: `[{"username":"BOB","message":id":"51edcad45862430000000001"}]`. Your ObjectId might vary.

If everything works as it should locally, try to deploy it to Heroku.

To test the application on Heroku, you could use the same CURL[67] commands, substituting `http://localhost/` or "http://127.0.0.1" with your unique Heroku app's host/URL:

```
$ 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
```

It's also nice to double check the database either via Mongo shell: `$ mongo` terminal command and then `use twitter-clone` and `db.messages.find()`; or via MongoHub[68], mongoui[69], mongo-express[70] or in case of MongoHQ through its web interface accessible at heroku.com website.

If you would like to use another domain name instead of http://your-app-name.herokuapp.com, you'll need to do two things:

1. Tell Heroku your domain name:

   ```
   $ heroku domains:add www.your-domain-name.com
   ```

2. Add the CNAME DNS record in your DNS manager to point to http://your-app-name.herokuapp.com.

More information on custom domains can be found at devcenter.heroku.com/articles/custom-domains[71].

Tip

For more productive and efficient development we should automate as much as possible, i.e., use tests instead of CURL commands. There is an article on the Mocha library in the BONUS chapter which, along with the superagent or request libraries, is a timesaver for such tasks.

[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 Putting It All Together

Summary: descriptions of different deployment approaches, final version of Chat application and its deployment.

> *"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."* — Brian W. Kernighan[1]

Now, it would be good if we could put our front-end and back-end applications so they could work together. There are a few ways to do it:

- Different domains (Heroku apps) for front-end and back-end apps: make sure there are no cross-domain issues by using CORS or JSONP. This approach is covered in detail later.
- Same domain deployment: make sure Node.js process static resources and assets for front-end application — not recommended for serious production applications.

7.1 Different Domain Deployment

This is, so far, the best practice for the production environment. Back-end applications are usually deployed at the *http://app.* or *http://api.* subdomains.

One way to make a different domain deployment work is to overcome the same-domain limitation of AJAX technology with JSONP:

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

The other, and better, way to do it is to add the OPTIONS method, and special headers, which are called CORS, to the Node.js server app before the output:

[1] http://en.wikipedia.org/wiki/Brian_Kernighan

```
...
response.writeHead(200,{
  'Access-Control-Allow-Origin': origin,
  'Content-Type':'text/plain',
  'Content-Length':body.length
});
...
```

or

```
...
res.writeHead(200, {
  'Access-Control-Allow-Origin', 'your-domain-name',
  ...
});
...
```

The need for the OPTIONS method is outlined in HTTP access control (CORS)[2]. The OPTIONS request can be dealt with in the following manner:

```
...
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, // Seconds.
    "Content-Length": 0
  });
  response.end();
};
...
```

7.2 Changing Endpoints

Our front-end application used Parse.com as a replacement for a back-end application. Now we can switch to our own back-end replacing the endpoints (yes, it's that painless!). The front-end app source code is in the rpjs/board[3] GitHub folder.

In the beginning of the **app.js** file, uncomment the first line for running locally, or replace the URL values with your Heroku or Windows Azure back-end application public URLs:

[2] https://developer.mozilla.org/en-US/docs/HTTP_access_control
[3] https://github.com/azat-co/rpjs/tree/master/board

```
// var URL = "http://localhost:5000/";
var URL ="http://your-app-name.herokuapp.com/";
```

As you can see, most of the code in **app.js** and the folder structure remained intact with the exception of replacing Parse.com models and collections with original Backbone.js ones:

```
Message = Backbone.Model.extend({
  url: URL + "messages/create.json"
})
MessageBoard = Backbone.Collection.extend ({
  model: Message,
  url: URL + "messages/list.json"
});
```

Those are the places where Backbone.js looks up for REST API URLs corresponding to the specific collection and model.

Here is the full source code of the rpjs/board/app.js[4] file:

```
/*
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'],

function(
headerTpl,
homeTpl,
footerTpl) {

  var ApplicationRouter = Backbone.Router.extend({
    routes: {
      "": "home",
      "*actions": "home"
```

[4] https://github.com/azat-co/rpjs/blob/master/board/app.js

```
  },
  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() {
    $(this.el).html(_.template(this.template));
  }
});

FooterView = Backbone.View.extend({
  el: "#footer",
  template: footerTpl,
  render: function() {
    this.$el.html(_.template(this.template));
  }
});
Message = Backbone.Model.extend({
  url: URL + "messages/create.json"
})
MessageBoard = Backbone.Collection.extend({
  model: Message,
  url: URL + "messages/list.json"
});

HomeView = Backbone.View.extend({
  el: "#content",
  template: homeTpl,
  events: {
    "click #send": "saveMessage"
  },
```

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

  app = new ApplicationRouter();
  Backbone.history.start();
});
```

7.3 Chat Application

The back-end Node.js application source code is in the rpjs/node[5] GitHub folder, which has this structure:

[5] https://github.com/azat-co/rpjs/tree/master/node

```
/node
        -web.js
        -Procfile
        -package.json
```

Here is a source code of web.js, our Node.js application implemented with CORS headers:

```
/*
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",
        "Access-Control-Allow-Headers": "content-type, accept",
        "Access-Control-Max-Age": 10, // Seconds.
        "Content-Length": 0
      });
      response.end();
    };
    if (request.method==="GET"&&
      request.url==="/messages/list.json") {
      collection.find().toArray(function(error,results) {
        var body = JSON.stringify(results);
        response.writeHead(200,{
```

```
          'Access-Control-Allow-Origin': origin,
          'Content-Type':'text/plain',
          'Content-Length':body.length
        });
        console.log("LIST OF OBJECTS: ");
        console.dir(results);
        response.end(body);
      });
    };
    if (request.method === "POST" &&
      request.url === "/messages/create.json") {
      request.on('data', function(data) {
        console.log("RECEIVED DATA:")
        console.log(data.toString('utf-8'));
        collection.insert(JSON.parse(data.toString('utf-8')),
        {safe:true}, function(error, obj) {
          if (error) throw error;
          console.log("OBJECT IS SAVED: ")
          console.log(JSON.stringify(obj))
          var body = JSON.stringify(obj);
          response.writeHead(200,{
                'Access-Control-Allow-Origin': origin,
            'Content-Type':'text/plain',
            'Content-Length':body.length
          });
          response.end(body);
        })
      })

    };

  });
  var port = process.env.PORT || 5000;
  app.listen(port);
})
```

7.4 Deployment

For your convenience, we have the front-end app in the rpjs/board[6] folder and the back-end app with CORS under rpjs/node[7]. By now, you probably know what to do, but as a reference, below are the steps to deploy these examples to Heroku.

In the **node** folder execute:

[6] https://github.com/azat-co/rpjs/tree/master/board
[7] https://github.com/azat-co/rpjs/tree/master/node

```
$ git init
$ git add .
$ git commit -am "first commit"
$ heroku create
$ heroku addons:add mongohq:sandbox
$ git push heroku master
```

Copy the URL and paste it into the **board/app.js** file, assigning the value to the URL variable. Then, in **board** folder, execute:

```
$ git init
$ git add .
$ git commit -am "first commit"
$ heroku create
$ git push heroku master
$ heroku open
```

7.5 Same Domain Deployment

Same domain deployment is *not recommended* for serious production applications, because static assets are better served with web servers like nginx (not Node.js I/O engine), and separating API makes for less complicated testing, increased robustness, and quicker troubleshooting/monitoring. However, the same app/domain approach could be used for staging, testing, development environments and/or tiny apps.

This is an example of a static Node.js server:

```
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");
```

Note

Another, more elegant way is to use Node.js frameworks as Connect (http://www.senchalabs.org/connect/static.html), or Express (http://expressjs.com/guide.html); because there is a special **static** middleware for JS and CSS assets.

8 BONUS: Webapplog Articles

Summary: articles on the essence of asynchronocity in Node.js, TDD with Mocha; introduction to Express.js, Monk, Wintersmith, Derby frameworks/libraries.

"Don't worry about failure; you only have to be right once." — Drew Houston[1]

For your convenience we included some of the Node.js posts from Webapplog.com[2] — a publicly accessible blog about web development — in this chapter.

8.1 Asynchronicity in Node

8.1.1 Non-Blocking I/O

One of the biggest advantages of using Node.js over Python or Ruby is that Node has a non-blocking I/O mechanism. To illustrate this, let me use an example of a line in a Starbucks coffeeshop. Let's pretend that each person standing in line for a drink is a task, and everything behind the counter — cashier, register, barista — is a server or server application. Whether we order a cup of regular drip coffee, like Pike Place, or hot tea, like Earl Grey, the barista makes it. The whole line waits while that drink is made, and each person is charged the appropriate amount.

Of course, we know the aforementioned drinks (a.k.a., time-consuming bottlenecks) are easy to make; just pour the liquid and it's done. But what about those fancy choco-mocha-frappe-latte-soy-decafs? What if everybody in line decides to order these time-consuming drinks? The line will be held up, and in turn, grow longer and longer. The manager of the coffeeshop will have to add more registers and put more baristas to work (or even stand behind the register him/herself).

This is not good, right? But this is how virtually all server-side technologies work, except Node.js, which is like a real Starbucks. When you order something, the barista yells the order to the other employee, and you leave the register. Another person gives their order while you wait for your state-of-the-art eye-opener in a paper cup. The line moves, the processes are executed asynchronously and without blocking the queue.

This is why Node.js blows everything else away (except maybe low-level C++) in terms of performance and scalability. With Node.js, you just don't need that many CPUs and servers to handle the load.

[1] http://en.wikipedia.org/wiki/Drew_Houston
[2] http://webapplog.com

8.1.2 Asynchronous Way of Coding

Asynchronicity requires a different way of thinking for programmers familiar with Python, PHP, C or Ruby. It's easy to introduce a bug unintentionally by forgetting to end the execution of the code with a proper **return** expression.

Here is a simple example illustrating this scenario:

```
var test = function (callback) {
  return callback();
  console.log('test') //shouldn't be printed
}

var test2 = function(callback){
  callback();
  console.log('test2') //printed 3rd
}

test(function(){
  console.log('callback1') //printed first
  test2(function(){
  console.log('callback2') //printed 2nd
  })
});
```

If we don't use return callback() and just use callback() our string test2 will be printed (test is not printed).

```
callback1
callback2
tes2
```

For fun I've added a `setTimeout()` delay for the callback2 string, and now the order has changed:

```
var test = function (callback) {
  return callback();
  console.log('test') //shouldn't be printed
}

var test2 = function(callback){
  callback();
  console.log('test2') //printed 2nd
}

test(function(){
  console.log('callback1') //printed first
```

```
  test2(function(){
    setTimeout(function(){
      console.log('callback2') //printed 3rd
    },100)
  })
});
```

Prints:

```
callback1
tes2
callback2
```

The last example illustrates that the two functions are independent of each other and run in parallel. The faster function will finish sooner than the slower one. Going back to our Starbucks examples, you might get your drink faster than the other person who was in front of you in the line. Better for people, and better for programs! :-)

8.2 MongoDB Migration with Monk

Recently one of our top users complained that his Storify[3] account was inaccessible. We've checked the production database, and it appears that the account might have been compromised and maliciously deleted by somebody using the user's account credentials. Thanks to a great MongoHQ service, we had a backup database in less than 15 minutes. There were two options to proceed with the migration:

1. Mongo shell script
2. Node.js program

Because Storify user account deletion involves deletion of all related objects — identities, relationships (followers, subscriptions), likes, stories — we've decided to proceed with the latter option. It worked perfectly, and here is a simplified version which you can use as a boilerplate for MongoDB migration (also at gist.github.com/4516139[4]).

Let's load all of the modules we need: Monk[5], Progress[6], Async[7], and MongoDB:

[3] http://storify.com
[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

```
var async = require('async');
var ProgressBar = require('progress');
var monk = require('monk');
var ObjectId=require('mongodb').ObjectID;
```

By the way, Monk, made by LeanBoost[8], is a "tiny layer that provides simple yet substantial usability improvements for MongoDB usage within Node.js".

Monk takes connection string in the following format:

```
username:password@dbhost:port/database
```

So we can create the following objects:

```
var dest = monk('localhost:27017/storify_localhost');
var backup = monk('localhost:27017/storify_backup');
```

We need to know the object ID which we want to restore:

```
var userId = ObjectId(YOUR-OBJECT-ID);
```

This is a handy `restore()` function which we can reuse to restore objects from related collections by specifying the query (for more on MongoDB queries, go to the post Querying 20M-Record MongoDB Collection[9]). To call it, just pass a name of the collection as a string, e.g., `"stories"` and a query which associates objects from this collection with your main object, e.g., `{userId:user.id}`. The progress bar is needed to show us nice visuals in the terminal:

```
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 {
```

[8] https://www.learnboost.com/

[9] http://www.webapplog.com/querying-20m-record-mongodb-collection/

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

Now we can use async to call the `restore()` function mentioned above:

```
async.series({
  restoreUser: function(callback){   // import user element
    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);
});
```

The full code is available at gist.github.com/4516139[10] and here:

```
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 should have auto casting but we need it for queries

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();
```

[10] https://gist.github.com/4516139

```
        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){   // import user element
    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);
});
```

To launch it, run `npm install`/`npm update` and change the hard-coded database values.

8.3 TDD in Node.js with Mocha

8.3.1 Who Needs Test-Driven Development?

Imagine that you need to implement a complex feature on top of an existing interface, e.g., a 'like' button on a comment. Without tests, you'll have to manually create a user, log in, create a post, create a different user, log in with a different user and like the post. Tiresome? What if you'll need to do it 10 or 20 times to find and fix some nasty bug? What if your feature breaks existing functionality, but you notice it six months after the release because there was no test?!

Don't waste time writing tests for throwaway scripts, but please adapt the habit of Test-Driven Development for the main code base. With a little time spent in the beginning, you and your team will save time later and have confidence when rolling out new releases. Test Driven Development is a really, really, really good thing.

8.3.2 Quick Start Guide

Follow this quick guide to set up your Test-Driven Development process in Node.js with Mocha[11].

Install Mocha[12] globally by executing this command:

```
$ sudo npm install -g mocha
```

We'll also use two libraries, Superagent[13] and expect.js[14] by LearnBoost[15]. To install them, fire up NPM[16] commands in your project folder like this:

```
$ npm install superagent
$ npm install expect.js
```

Open a new file with `.js` extension and type:

[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/

```
var request = require('superagent');
var expect = require('expect.js');
```

So far we've included two libraries. The structure of the test suite going to look like this:

```
describe('Suite one', function(){
  it(function(done){
  ...
  });
  it(function(done){
  ...
  });
});
describe('Suite two', function(){
  it(function(done){
  ...
  });
});
```

Inside of this closure, we can write a request to our server, which should be running at localhost:8080[17]:

```
...
it (function(done){
  request.post('localhost:8080').end(function(res){
    //TODO check that response is okay
  });
});
...
```

Expect will give us handy functions to check any condition we can think of:

```
...
expect(res).to.exist;
expect(res.status).to.equal(200);
expect(res.body).to.contain('world');
...
```

Lastly, we need to add the **done()** call to notify Mocha that the asynchronous test has finished its work. And the full code of our first test looks like this:

[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();
  });
});
```

If we want to get fancy, we can add **before** and **beforeEach** hooks which will, according to their names, execute once before the test (or suite) or each time before the test (or suite):

```
before(function(){
  //TODO seed the database
});
describe('suite one ',function(){
  beforeEach(function(){
    //todo log in test user
  });
  it('test one', function(done){
  ...
  });
});
```

Note that before and beforeEach can be placed inside or outside of the describe construction.

To run our test, simply execute:

```
$ mocha test.js
```

To use a different report type:

```
$ mocha test.js -R list
$ mocah test.js -R spec
```

8.4 Wintersmith — Static Site Generator

For this book's one-page website —rapidprototypingwithjs.com[18] — I used Wintersmith[19] to learn something new and to ship fast. Wintersmith is a Node.js static site generator. It greatly impressed me with its flexibility

[18] http://rapidprototypingwithjs.com
[19] http://jnordberg.github.com/wintersmith/

and ease of development. In addition, I could stick to my favorite tools such as Markdown[20], Jade and Underscore[21].

Why Static Site Generators

Here is a good article on why using a static site generator is a good idea in general: An Introduction to Static Site Generators[22]. It basically boils down to a few main things:

Templates

You can use template engines such as Jade[23]. Jade uses whitespaces to structure nested elements, and its syntax is similar to Ruby on Rail's Haml markup.

Markdown

I've copied markdown text from my book's Introduction chapter and used it without any modifications. Wintersmith comes with a marked[24] parser by default. More on why Markdown is great in my old post: Markdown Goodness[25].

Simple Deployment

Everything is HTML, CSS and JavaScript so you just upload the files with an FTP client, e.g., Transmit[26] (by Panic) or Cyberduck[27].

Basic Hosting

Due to the fact that any static web server will work well, there is no need for Heroku or Nodejitsu PaaS solutions, or even PHP/MySQL hosting.

Performance

There are no database calls, no server-side API calls, and no CPU/RAM overhead.

Flexibility

Wintersmith allows for different plugins for contents and templates, and you can even write your own plugins[28].

8.4.1 Getting Started with Wintersmith

There is a quick getting started guide on github.com/jnordberg/wintersmith[29].

To install Wintersmith globally, run NPM with -g and sudo:

[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

```
$ sudo npm install wintersmith -g
```

Then run to use the default blog template:

```
$ wintersmith new <path>
```

or for an empty site:

```
$ wintersmith new <path> -template basic
```

or use a shortcut:

```
$ wintersmith new <path> -T basic
```

Similar to Ruby on Rails scaffolding, Wintersmith will generate a basic skeleton with **contents** and **templates** folders. To preview a website, run these commands:

```
$ cd <path>
$ wintersmith preview
$ open http://localhost:8080
```

Most of the changes will be updates automatically in the preview mode, except for the config.json file[30].

Images, CSS, JavaScript and other files go into the **contents** folder. The Wintersmith generator has the following logic:

1. looks for *.md files in the contents folder
2. reads metadata[31] such as the template name
3. processes *.jade templates[32] per metadata in *.md files

When you're done with your static site, just run:

```
$ wintersmith build
```

[30] https://github.com/jnordberg/wintersmith#config
[31] https://github.com/jnordberg/wintersmith#the-page-plugin
[32] https://github.com/jnordberg/wintersmith#templates

8.4.2 Other Static Site Generators

Here are some of the other Node.js static site generators:

- DocPad[33]
- Blacksmith[34]
- Scotch[35]
- Wheat[36]
- Petrify[37]

A more detailed overview of these static site generators is available in the post Node.js Based Static Site Generators[38].

For other languages and frameworks like Rails and PHP, take a look at Static Site Generators by GitHub Watcher Count[39] and the "mother of all site generator lists[40]".

8.5 Intro to Express.js: Simple REST API app with Monk and MongoDB

8.5.1 REST API app with Express.js and Monk

This app is a start of a mongoui[41] project — a phpMyAdmin counterpart for MongoDB written in Node.js. The goal is to provide a module with a nice web admin user interface. It will be something like Parse.com[42], Firebase.com[43], MongoHQ[44] or MongoLab[45] has, but without tying it to any particular service. Why do we have to type `db.users.findOne({'_id':ObjectId('...')})` anytime we want to look up the user information? The alternative MongoHub[46] Mac app is nice (and free) but clunky to use and not web-based.

Ruby enthusiasts like to compare Express to the Sinatra[47] framework. It's similarly flexible in the way developers can build their apps. Application routes are set up in a similar manner, i.e., `app.get('/products/:id', showProduct);`. Currently Express.js is at version number 3.1. In addition to Express, we'll use the Monk[48] module.

We'll use Node Package Manager[49], which usually comes with a Node.js installation. If you don't have it

[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
[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

already, you can get it at npmjs.org[50].

Create a new folder and NPM configuration file, **package.json**, in it with the following content:

```
{
  "name": "mongoui",
  "version": "0.0.1",
  "engines": {
    "node": ">= v0.6"
  },
  "dependencies": {
    "mongodb":"1.2.14",
    "monk": "0.7.1",
    "express": "3.1.0"
  }
}
```

Now run `npm install` to download and install modules into the **node_module** folder. If everything went okay you'll see bunch of folders in **node_modules** folders. All of the code for our application will be in one file, **index.js**, to 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)
```

[50] http://npmjs.org

Let's break down the code piece by piece. Module declaration:

```
var mongo = require('mongodb');
var express = require('express');
var monk = require('monk');
```

Database and Express application instantiation:

```
var db =  monk('localhost:27017/test');
var app = new express();
```

Tell Express application to load and server static files (if there are any) from the public folder:

```
app.use(express.static(__dirname + '/public'));
```

Home page, a.k.a. root route, set up:

```
app.get('/',function(req,res){
  db.driver.admin.listDatabases(function(e,dbs){
      res.json(dbs);
  });
});
```

`get()` function just takes two parameters: string and function. The string can have slashes and colons — for example, `product/:id`. The function must have two parameters: request and response. Request has all of the information like query string parameters, session and headers, and response is an object to which we output the results. In this case, we do it by calling the `res.json()` function.

`db.driver.admin.listDatabases()`, as you might guess, gives us a list of databases in an asynchronous manner.

Two other routes are set up in a similar manner with the `get()` function:

```
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 conveniently supports other HTTP verbs like post and update. In the case of setting up a post route, we write this:

```
app.post('product/:id',function(req,res) {...});
```

Express also has support for middleware. Middleware is just a request function handler with three parameters: `request`, `response`, and `next`. For example:

```
app.post('product/:id',
  authenticateUser,
  validateProduct,
  addProduct
);

function authenticateUser(req,res, next) {
  //check req.session for authentication
  next();
}

function validateProduct (req, res, next) {
   //validate submitted data
   next();
}

function addProduct (req, res) {
  //save data to database
}
```

validateProduct and authenticateProduct are middlewares. They are usually put into separate file (or files) in big projects.

Another way to set up middleware in the Express application is to utilize the `use()` function. For example, earlier we did this for static assets:

```
app.use(express.static(__dirname + '/public'));
```

We can also do it for error handlers:

```
app.use(errorHandler);
```

Assuming you have mongoDB installed, this app will connect to it (localhost:27017[51]) and display the collection name and items in collections. To start the mongo server:

```
$ mongod
```

to run the app (keep the **mongod** terminal window open):

[51] http://localhost:27017

```
$ node .
```

or

```
$ node index.js
```

To see the app working, open localhost:3000[52] in Chrome with the JSONViewer[53] extension (to render JSON nicely).

8.6 Intro to Express.js: Parameters, Error Handling and Other Middleware

8.6.1 Request Handlers

Express.js is a node.js framework that, among other things, provides a way to organize routes. Each route is defined via a method call on an application object with a URL patter as a first parameter (RegExp is also supported) — for example:

```
app.get('api/v1/stories/', function(res, req){
  ...
})
```

or, for a POST method:

```
app.post('/api/v1/stories'function(req,res){
  ...
})
```

Needless to say, DELETE and PUT methods are supported as well[54].

The callbacks that we pass to the `get()` or `post()` methods are called request handlers because they take requests (`req`), process them, and write to response (`res`) objects. For example:

```
app.get('/about', function(req,res){
  res.send('About Us: ...');
});
```

We can have multiple request handlers — hence the name *middleware*. They accept a third parameter, `next`, calling which (`next()`) will switch the execution flow to the next handler:

[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) {
  //do authorization
  //if not authorized or there is an error
  // return next(error);
  //if authorized and no errors
  return next();
}), function(req,res, next) {
  //extract id and fetch the object from the database
  //assuming no errors, save story in the request object
  req.story = story;
  return next();
}), function(req,res) {
  //output the result of the database search
  res.send(res.story);
});
```

The ID of a story in URL patter is a query string parameter which we need for finding matching items in the database.

8.6.2 Parameters Middleware

Parameters are values passed in a query string of a URL of the request. If we didn't have Express.js or a similar library and had to use just the core Node.js modules, we'd have to extract parameters from HTTP.request[55] object via some `require('querystring').parse(url)` or `require('url').parse(url, true)` functions trickery.

Thanks to Connect framework[56] and the people at VisionMedia[57], Express.js already has support for parameters, error handling and many other important features in the form of middlewares. This is how we can plug param middleware in our app:

```
app.param('id', function(req,res, next, id){
  //do something with id
  //store id or other info in req object
  //call next when done
  next();
});

app.get('/api/v1/stories/:id',function(req,res){
  //param middleware will be execute before and
  //we expect req object already have needed info
  //output something
  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

For example:

```
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);
});
```

Or we can use multiple request handlers, but the concept remains the same: we can expect to have the `req.story` object or an error thrown prior to the execution of this code, so we abstract the common code/logic of getting parameters and their respective objects:

```
app.get('/api/v1/stories/:id', function(req,res, next) {
  //do authorization
  }),
  //we have an object in req.story so no work is needed here
  function(req,res) {
  //output the result of the database search
  res.send(story);
});
```

Authorization and input sanitation are also good candidates for residing in the middlewares.

The function `param()` is especially cool because we can combine different keys, e.g.:

```
app.get('/api/v1/stories/:storyId/elements/:elementId',
  function(req,res){
    res.send(req.element);
  }
);
```

8.6.3 Error Handling

Error handling is typically used across the whole application, so it's best to implement it as a middleware. It has the same parameters plus one more, `error`:

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  res.send(500);
})
```

In fact, the response can be anything:

JSON string

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  res.send(500, {status:500,
    message: 'internal error',
    type:'internal'}
  );
})
```

Text message

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  res.send(500, 'internal server error');
})
```

Error page

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  //assuming that template engine is plugged in
  res.render('500');
})
```

Redirect to error page

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  res.redirect('/public/500.html');
})
```

Error HTTP response status (401, 400, 500, etc.)

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  res.end(500);
})
```

By the way, logging should also be abstracted in a middleware!

To trigger an error from within your request handlers and middleware, you can just call:

```
next(error);
```

or

```
next(new Error('Something went wrong :-(');
```

You can also have multiple error handlers and use named instead of anonymous functions, as its shows in the Express.js Error handling guide[58].

8.6.4 Other Middleware

In addition to extracting parameters, it can be used for many things, like authorization, error handling, sessions, output, and others.

`res.json()` is one of them. It conveniently outputs the JavaScript/Node.js object as a JSON. For example:

```
app.get('/api/v1/stories/:id', function(req,res){
  res.json(req.story);
});
```

is equivalent to (if `req.story` is an Array and Object):

```
app.get('/api/v1/stories/:id', function(req,res){
  res.send(req.story);
});
```

or

[58] http://expressjs.com/guide.html#error-handling

```
app.get('api/v1/stories/:id',function(req,res){
  res.set({
    'Content-Type': 'application/json'
  });
  res.send(req.story);
});
```

8.6.5 Abstraction

Middleware is flexible. You can use anonymous or named functions, but the best thing is to abstract request handlers into external modules based on the functionality:

```
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){
};
```

You can use some functional programming tricks, like this:

```
function requiredParamHandler(param){
  //do something with a param, e.g.,
  //check that it's present in a query string
  return function (req,res, next) {
    //use param, e.g., if token is valid proceed with next();
    next();
  });
}

app.get('/api/v1/stories/:id',
  requiredParamHandler('token'),
  story.show
);

var story  = {
  show: function (req, res, next) {
    //do some logic, e.g., restrict fields to output
    return res.send();
  }
}
```

As you can see, middleware is a powerful concept for keeping code organized. The best practice is to keep the router lean and thin by moving all of the logic into corresponding external modules/files. This way, important server configuration parameters will be neatly in one place when you need them! :-)

8.7 JSON REST API server with Node.js and MongoDB using Mongoskin and Express.js

This tutorial will walk you through writing test using the Mocha[59] and Super Agent[60] libraries and then use them in a test-driven development manner to build a Node.js[61] free JSON REST API server utilizing Express.js[62] framework and Mongoskin[63] library for MongoDB[64]. In this REST API server, we'll perform **create**, **update**, **remove and delete** (CRUD) operations and harness Express.js middleware[65] concept with `app.param()` and `app.use()` methods.

[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

8.7.1 Test Coverage

Before anything else let's write functional tests that make HTTP requests to our soon-to-be-created REST API server. If you know how to use Mocha[66] or just want to jump straight to the Express.js app implementation feel free to do so. You can use CURL terminal commands for testing too.

Assuming we already have Node.js, NPM[67] and MongoDB installed, let's create a *new* folder (or if you wrote the tests use that folder):

```
mkdir rest-api
cd rest-api
```

We'll use Mocha[68], Expect.js[69] and Super Agent[70] libraries. To install them run these command from the project folder:

```
$ npm install mocha
$ npm install expect.js
$ npm install superagent
```

Now let's create **express.test.js** file in the same folder which will have six suites:

- creating a new object
- retrieving an object by its ID
- retrieving the whole collection
- updating an object by its ID
- checking an updated object by its ID
- removing an object by its ID

HTTP requests are just a breeze with Super Agent's chained functions which we'll put inside of each test suite. Here is the full source code for the **express.test.js** file:

[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/

```
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){
        // console.log(res.body)
        expect(e).to.eql(null)
        expect(res.body.length).to.be.above(1)
        expect(res.body.map(function (item){
          return item._id
        })).to.contain(id)
        done()
      })
  })
```

```
  it('updates an object', function(done){
    superagent.put('http://localhost:3000/collections/test/'+id)
      .send({name: 'Peter'
        , email: 'peter@yahoo.com'})
      .end(function(e, res){
        // console.log(res.body)
        expect(e).to.eql(null)
        expect(typeof res.body).to.eql('object')
        expect(res.body.msg).to.eql('success')
        done()
      })
  })

  it('checks an updated 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)
        expect(res.body.name).to.eql('Peter')
        done()
      })
  })

  it('removes an object', function(done){
    superagent.del('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.msg).to.eql('success')
        done()
      })
  })
})
```

To run the tests we can use the `$ mocha express.test.js` command.

8.7.2 Dependencies

In this tutorial we'll utilize Mongoskin[71], a MongoDB library which is a better alternative to the plain good old native MongoDB driver for Node.js[72]. In addition Mongoskin is more light-weight than Mongoose and schema-less. For more insight please check out Mongoskin comparison blurb[73].

Express.js[74] is a wrapper for the core Node.js HTTP module[75] objects. The Express.js framework is build on top of Connect[76] middleware and provided tons of convenience. Some people compare the framework to Ruby's Sinatra in terms of how it's non-opinionated and configurable.

If you've create a `rest-api` folder in the previous section *Test Coverage*, simply run these commands to install modules for the application:

```
npm install express
npm install mongoskin
```

8.7.3 Implementation

First things first, so let's define our dependencies:

```
var express = require('express')
  , mongoskin = require('mongoskin')
```

After the version 3.x, Express streamlines the instantiation of its app instance, in a way that this line will give us a server object:

```
var app = express()
```

To extract params from the body of the requests we'll use `bodyParser()` middleware which looks more like a configuration statement:

```
app.use(express.bodyParser())
```

Middleware (in this[77] and other forms[78]) is a powerful and convenient pattern in Express.js and Connect[79] to organize and re-use code.

As with the `bodyParser()` method that saves us from the hurdles of parsing a body object of HTTP request, Mongoskin makes possible to connect to the MongoDB database in one effortless line of code:

[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
[77] http://expressjs.com/api.html#app.use
[78] http://expressjs.com/api.html#middleware
[79] https://github.com/senchalabs/connect

```
var db = mongoskin.db('localhost:27017/test', {safe:true});
```

Note: *If you wish to connect to a remote database, e.g., MongoHQ[80] instance, substitute the string with your username, password, host and port values. Here is the format of the URI string:* `mongodb://[username:password@]host1[:port1`

The `app.param()` method is another Express.js middleware. It basically says "*do something every time there is this value in the URL pattern of the request handler.*" In our case we select a particular collection when request pattern contains a sting `collectionName` prefixed with a colon (you'll see it later in the routes):

```
app.param('collectionName',
  function(req, res, next, collectionName) {
    req.collection = db.collection(collectionName)
    return next()
  }
)
```

Merely to be user-friendly, let's put a root route with a message:

```
app.get('/', function(req, res) {
  res.send('please select a collection, e.g., /collections/messages')
})
```

Now the real work begins. Here is how we retrieve a list of items sorted by `_id` and which has a limit of 10:

```
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)
      }
    )
  }
)
```

Have you noticed a `:collectionName` string in the URL pattern parameter? This and the previous `app.param()` middleware is what gives us the `req.collection` object which points to a specified collection in our database.

The object creating endpoint is slightly easier to grasp since we just pass the whole payload to the MongoDB (method a.k.a. free JSON REST API):

[80] https://www.mongohq.com/home

```
app.post('/collections/:collectionName', function(req, res) {
  req.collection.insert(req.body, {}, function(e, results){
    if (e) return next(e)
    res.send(results)
  })
})
```

Single object retrieval functions are faster than `find()`, but they use different interface (they return object directly instead of a cursor), so please be aware of that. In addition, we're extracting the ID from `:id` part of the path with `req.params.id` Express.js magic:

```
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)
    }
  )
})
```

PUT request handler gets more interesting because `update()` doesn't return the augmented object, instead it returns us a count of affected objects.

Also `{$set:req.body}` is a special MongoDB operator (operators tend to start with a dollar sign) that sets values.

The second ‘ {safe:true, multi:false}‘ parameter is an object with options that tell MongoDB to wait for the execution before running the callback function and to process only one (first) item.

```
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'})
    }
  )
})
```

Finally, the DELETE method which also output a custom JSON message:

```
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'})
    }
  )
})
```

Note: *The `delete` is an operator in JavaScript, so Express.js uses `app.del` instead.*

The last line that actually starts the server on port 3000 in this case:

```
app.listen(3000)
```

Just in case something is not working quite well here is the full code of **express.js** file:

```
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'})
    }
  )
})

app.listen(3000)
```

Exit your editor and run this in your terminal:

```
$ node express.js
```

And in a different window (without closing the first one):

```
$ mocha express.test.js
```

If you really don't like Mocha and/or BDD, CURL is always there for you. :-)

For example, CURL data to make a POST request:

```
$ curl -d "" http://localhost:3000
```

GET requests also work in the browser, for example http://localhost:3000/test.

In this tutorial our tests are longer than the app code itself so abandoning test-driven development might be tempting, but believe me **the good habits of TDD will save you hours and hours** during any serious development when the complexity of the applications you work one is big.

8.7.4 Conclusion

The Express.js and Mongoskin libraries are great when you need to build a simple REST API server in a few line of code. Later, if you need to expand the libraries they also provide a way to configure and organize your code.

NoSQL databases like MongoDB are good at free-REST APIs where we don't have to define schemas and can throw any data and it'll be saved.

The full code of both test and app files: https://gist.github.com/azat-co/6075685.

If you like to learn more about Express.js and other JavaScript libraries take a look at the series Intro to Express.js tutorials[81].

Note: *In this example I'm using semi-colon less style. Semi-colons in JavaScript are absolutely optional[82] except in two cases: in the for loop and before expression/statement that starts with parenthesis (e.g., Immediately-Invoked Function Expression[83]).

8.8 Node.js MVC: Express.js + Derby Hello World Tutorial

8.8.1 Node MVC Framework

Express.js[84] is a popular node framework which uses the middleware concept to enhance the functionality of applications. Derby[85] is a new sophisticated Model View Controller (MVC[86]) framework which is designed to be used with Express[87] as its middleware. Derby also comes with the support of Racer[88], data synchronization engine, and a Handlebars[89]-like template engine, among many other features[90].

[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
[84] http://expressjs.com
[85] http://derbyjs.com
[86] http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
[87] http://expressjs.com
[88] https://github.com/codeparty/racer
[89] https://github.com/wycats/handlebars.js/
[90] http://derbyjs.com/#features

8.8.2 Derby Installation

Let's set up a basic Derby application architecture without the use of scaffolding. Usually project generators are confusing when people just start to learn a new comprehensive framework. This is a bare minimum "Hello World" application tutorial that still illustrates the Derby skeleton and demonstrates live-templates with websockets.

Of course, we'll need Node.js[91] and NPM[92], which can be obtained at nodejs.org[93]. To install Derby globally, run:

```
$ npm install -g derby
```

To check the installation:

```
$ derby -V
```

My version, as of April 2013, is 0.3.15. We should be good to go to creating our first app!

8.8.3 File Structure

This is the project folder structure:

```
project/
  -package.json
  -index.js
  -derby-app.js
  views/
    derby-app.html
  styles/
    derby-app.less
```

8.8.4 Dependencies

Let's include dependencies and other basic information in the **package.json** file:

[91] http://nodejs.org
[92] http://npmjs.org
[93] http://nodejs.org

```
 {
  "name": "DerbyTutorial",
  "description": "",
  "version": "0.0.0",
  "main": "./server.js",
  "dependencies": {
    "derby": "*",
    "express": "3.x"
  },
  "private": true
}
```

Now we can run `npm install`, which will download our dependencies into **node_modules** folder.

8.8.5 Views

Views must be in the **views** folder, and they must be either in **index.html** under a folder which has the same name as your derby app JavaScript file, i.e., `views/derby-app/index.html`, or be inside of a file which has the same name as your derby app JS file, i.e., **derby-app.html**.

In this example, the "Hello World" app, we'll use `<Body:>` template and `{message}` variable. Derby uses mustach[94]-handlebars-like syntax for reactive binding. **index.html** looks like this:

```
<Body:>
  <input value="{message}"><h1>{message}</h1>
```

Same thing with Stylus/LESS files; in our example, index.css has just one line:

```
h1 {
  color: blue;
}
```

To find out more about those wonderful CSS preprocessors, check out the documentation at Stylus[95] and LESS[96].

8.8.6 Main Server

index.js is our main server file, and we begin it with an inclusion of dependencies with the `require()` function:

[94] http://mustache.github.io/
[95] http://learnboost.github.io/stylus/
[96] http://lesscss.org/

```
var http = require('http'),
  express = require('express'),
  derby = require('derby'),
  derbyApp = require('./derby-app');
```

The last line is our derby application file **derby-app.js**.

Now we're creating the Express.js application (v3.x has significant differences between 2.x) and an HTTP server:

```
var expressApp = new express(),
  server = http.createServer(expressApp);
```

Derby[97] uses the Racer[98] data synchronization library, which we create like this:

```
var store = derby.createStore({
  listen: server
});
```

To fetch some data from back-end to the front-end, we instantiate the model object:

```
var model = store.createModel();
```

Most importantly we need to pass the model and routes as middlewares to the Express.js app. We need to expose the public folder for socket.io to work properly.

```
expressApp.
  use(store.modelMiddleware()).
  use(express.static(__dirname + '/public')).
  use(derbyApp.router()).
  use(expressApp.router);
```

Now we can start the server on port 3001 (or any other):

```
server.listen(3001, function(){
  model.set('message', 'Hello World!');
});
```

Full code of **index.js** file:

[97] http://derbyjs.com
[98] https://github.com/codeparty/racer

```
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 Application

Finally, the Derby app file, which contains code for both a front-end and a back-end. Front-end only code is inside of the `app.ready()` callback. To start, let's require and create an app. Derby uses unusual construction (not the same familiar good old `module.exports = app`):

```
var derby = require('derby'),
  app = derby.createApp(module);
```

To make socket.io magic work, we need to subscribe a model attribute to its visual representation — in other words, bind data and view. We can do it in the root route, and this is how we define it (patter is `/`, a.k.a. root):

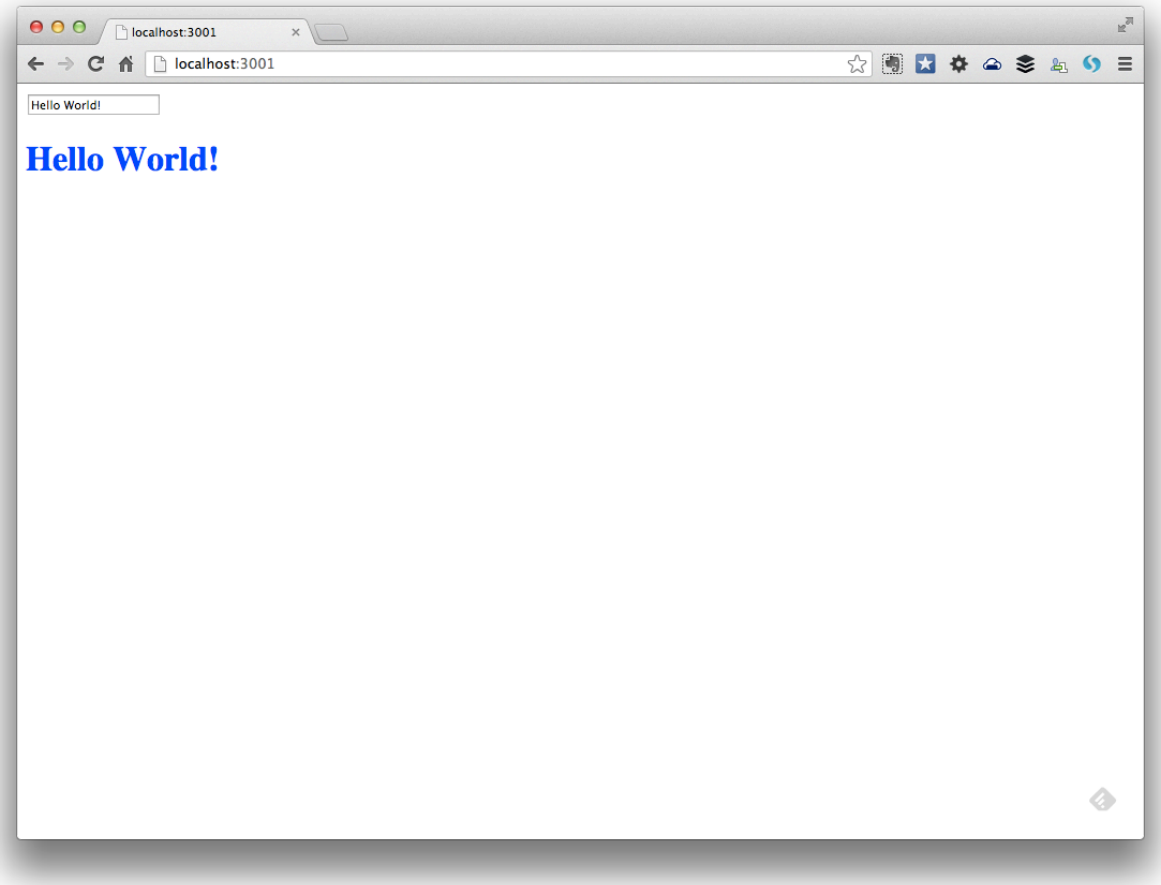
```
app.get('/', function(page, model, params) {
  model.subscribe('message', function() {
    page.render();
  });
});
```

Full code of **derby-app.js** file:

```
var derby = require('derby'),
  app = derby.createApp(module);

app.get('/', function(page, model, params) {
  model.subscribe('message', function() {
    page.render();
  });
});
```

8.8.8 Launching Hello World App

Now everything should be ready to boot our server. Execute `node .` or `node index.js` and open a browser at localhost:3001[99]. You should be able to see something like this:



Derby + Express.js Hello World App

[99] http://localhost:3001

8.8.9 Passing Values to Back-End

Of course, the static data is not much, so we can slightly modify our app to make back-end and front-end pieces talks with each other.

In the server file **index.js**, add `store.afterDb` to listen to `set` events on the **message** attribute:

```
server.listen(3001, function(){
  model.set('message', 'Hello World!');
  store.afterDb('set','message', function(txn, doc, prevDoc, done){
    console.log(txn)
    done();
  });
});
```

Full code of **index.js** after modifications:

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

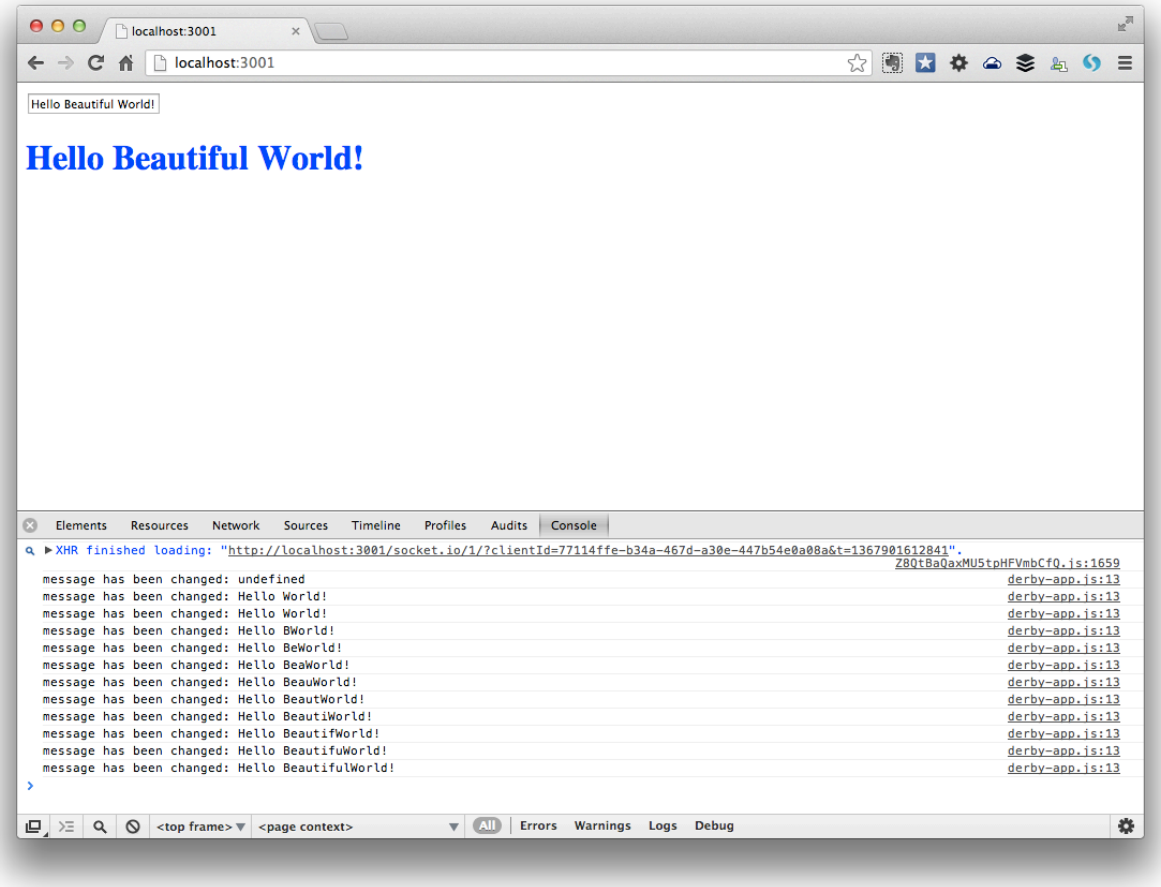
In the Derby application file **derby-app.js**, add `model.on()` to `app.ready()`:

```
  app.ready(function(model){
            model.on('set', 'message',function(path, object){
            console.log('message has been changed: '+ object);
          })
  });
```

Full **derby-app.js** file after modifications:

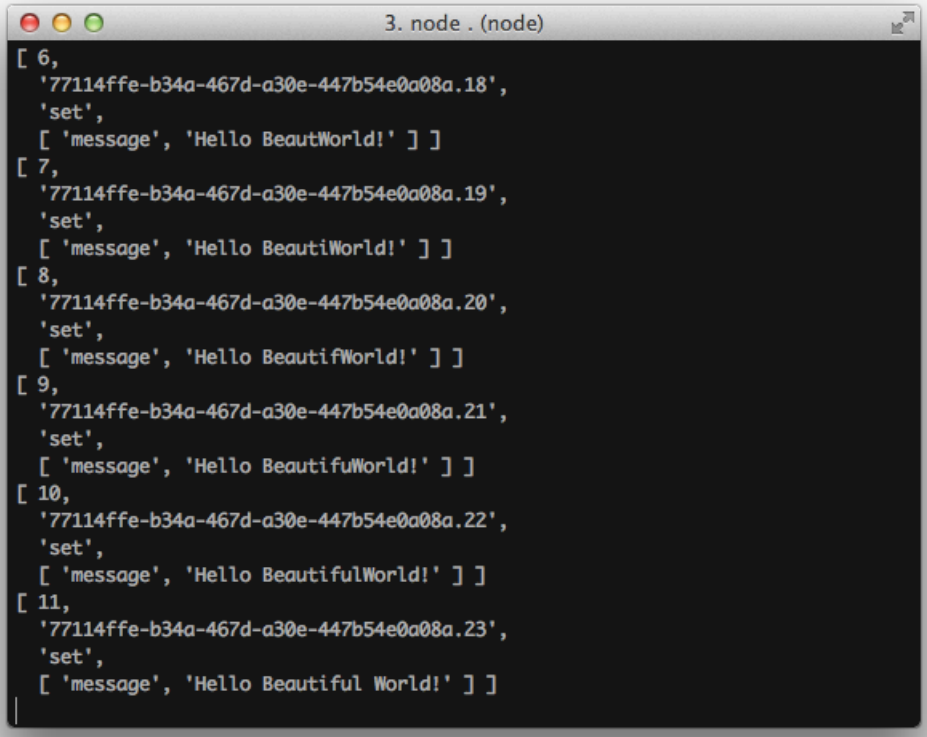
```
var derby = require('derby'),
  app = derby.createApp(module);

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);
  })
});
```

Now we'll see logs both in the terminal window and in the browser Developer Tools console. The end result should look like this in the browser:



Hello World App: Browser Console Logs

And like this in the terminal:



Hello World App: Terminal Console Logs

For more magic in the persistence area, check out Racer's db property[100]. With it you can set up an automatic synch between views and database!

The full code of all of the files in this Express.js + Derby Hello World app is available as a gist at gist.github.com/azat-co/5530311[101].

[100] http://derbyjs.com/#persistence
[101] https://gist.github.com/azat-co/5530311

Conclusion and Further Reading

Summary: the book's conclusion, lists of JavaScript blog posts, articles, ebooks, books and other resources.

Conclusion

We hope you've enjoyed this book. It was intended to be small on theory but big on practice, and give you an overview of multiple technologies, frameworks and techniques used in modern agile web development. Rapid Prototyping with JS touched topics such as:

- 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

If you need in-depth knowledge or references, they are usually one click or Google search away.

Practical aspect included building multiple versions of the Chat app:

- jQuery + Parse.com JS REST API
- Backbone and Parse.com JS SDK
- Backbone and Node.js
- Backbone and Node.js + MongoDB

The Chat application has all the foundation of a typical web/mobile application: fetching data, displaying it, submitting new data. Other examples include:

- 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"

Please submit a GitHub issue, if you have any feedback, comments, suggestions, or you've found typos, bugs, mistakes or other errata: https://github.com/azat-co/rpjs/issues.

Other ways to connect are via: @azat_co[102], http://webapplog.com, http://azat.co.

In case you enjoyed Node.js and want to find out more about building production web services with Express.js — a de factor standard for Node.js web apps — take a look at my new book Express.js Guide: The Most Popular Node.js Framework Manual[103].

Further Reading

Here is a list of resources, courses, books and blogs for further reading.

JavaScript resources and free ebooks

- Oh My JS[104]: Hand-picked collection of the best JavaScript articles
- JavaScript For Cats[105]: An introduction for new programmers
- Eloquent JavaScript[106]: A modern introduction to programming
- Superhero.js[107]: comprehensive collection of JS resources
- JavaScript Guide[108] by Mozilla Developer Network
- JavaScript Reference[109] by Mozilla Developer Network
- Why Use Closure[110]: practical uses of a closure in event based programming
- Prototypal Inheritance[111]: objects with inherited and local properties

[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/en-US/docs/JavaScript/Guide
[109] https://developer.mozilla.org/en-US/docs/JavaScript/Reference
[110] http://howtonode.org/why-use-closure
[111] http://howtonode.org/prototypical-inheritance

- Control Flow in Node[112]: parallel vs serial flows
- Truthy and Falsey Values[113]
- How to Write Asynchronous Code[114]
- Smooth CoffeeScript[115]: free interactive HTML5 book and collection of quick references and other goodies
- Developing Backbone.js Applications[116]: free early release book By Addy Osmani and O'Reilly
- Step by step from jQuery to Backbone[117]
- Open Web Platform Daily Digest[118]: JS daily digest
- DISTILLED HYPE[119]: JS blog/newsletter

JavaScript books

- JavaScript: The Good Parts[120]
- JavaScript: The Definitive Guide[121]
- Secrets of the JavaScript Ninja[122]
- Pro JavaScript Techniques[123]

Node.js resources and free ebooks

- Felix's Node.js Beginners Guide[124]
- Felix's Node.js Style Guide[125]
- Felix's Node.js Convincing the boss guide[126]
- Introduction to NPM[127]
- NPM Cheatsheet[128]
- Interactive Package.json Cheatsheet[129]
- Official Node.js Documentation[130]
- Node Guide[131]

[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://nodeguide.com/beginner.html
[125] http://nodeguide.com/style.html
[126] http://nodeguide.com/convincing_the_boss.html
[127] http://howtonode.org/introduction-to-npm
[128] http://blog.nodejitsu.com/npm-cheatsheet
[129] http://package.json.nodejitsu.com/
[130] http://nodejs.org/api/
[131] http://nodeguide.com/

- Node Tuts[132]
- What Is Node?[133]: free Kindle edition
- Mastering Node.js[134]: open source node ebook
- Mixu's Node book[135]: A book about using Node.js
- Learn Node.js Completely and with Confidence[136]: guide to learning JavaScript in 2 weeks
- How to Node[137]: The zen of coding in node.js

Node.js books

- The Node Beginner Book[138]
- Hands-on Node.js[139]
- Backbone Tutorials[140]
- Smashing Node.js[141]
- The Node Beginner Book[142]
- Hands-on Node.js[143]
- Node: Up and Running[144]
- Node.js in Action[145]
- Node: Up and Running[146]: Scalable Server-Side Code with JavaScript
- Node Web Development[147]: A practical introduction to Node
- Node Cookbook[148]

Interactive online classes and courses

- Cody Academy[149]: interactive programming cources
- Programr[150]
- LearnStreet[151]

[132] http://nodetuts.com/
[133] http://www.amazon.com/What-Is-Node-ebook/dp/B005ISQ7JC
[134] http://visionmedia.github.com/masteringnode/
[135] http://book.mixu.net/
[136] http://javascriptissexy.com/learn-node-js-completely-and-with-confidence/
[137] http://howtonode.org/
[138] https://leanpub.com/nodebeginner
[139] https://leanpub.com/hands-on-nodejs
[140] https://leanpub.com/backbonetutorials
[141] http://www.amazon.com/Smashing-Node-js-JavaScript-Everywhere-Magazine/dp/1119962595/
[142] http://www.nodebeginner.org/
[143] http://nodetuts.com/handson-nodejs-book.html
[144] http://shop.oreilly.com/product/0636920015956.do
[145] http://www.manning.com/cantelon/
[146] http://www.amazon.com/Node-Running-Scalable-Server-Side-JavaScript/dp/1449398588
[147] http://www.amazon.com/Node-Web-Development-David-Herron/dp/184951514X
[148] http://www.amazon.com/Node-Cookbook-David-Mark-Clements/dp/1849517185/
[149] http://www.codecademy.com/
[150] http://www.programr.com/
[151] http://www.learnstreet.com/

- Treehouse[152]
- lynda.com[153]: software, creative and business courses
- Udacity[154]: Massive open online courses
- Coursera[155]

Startup books and blogs

- Hackers & Painters[156]
- The Lean Startup[157]
- The Startup Owner's Manual[158]
- The Entrepreneur's Guide to Customer Development[159]
- Venture Hacks[160]
- WebAppLog[161]

[152] http://teamtreehouse.com/
[153] http://www.lynda.com/
[154] https://www.udacity.com/
[155] https://www.coursera.org/
[156] http://www.amazon.com/Hackers-Painters-Big-Ideas-Computer/dp/1449389554
[157] http://theleanstartup.com/book
[158] http://www.amazon.com/Startup-Owners-Manual-Step-Step/dp/0984999302
[159] http://www.amazon.com/The-Entrepreneurs-Guide-Customer-Development/dp/0982743602/
[160] http://venturehacks.com/
[161] http://webapplog.com