sitepoint

rubysource.com

Rails Deep Dive

By GLENN GOODRICH

Build a Rails Application from Scratch

Summary of Contents

Preface xi
1. Ruby Version Manager 1
2. Installing Rails 7
3. App Generation 17
4. Application Setup: Loccasions 23
5. Home Page 29
6. Authentication 37
7. Spork, Events, and Authorization 49
8. Making Events 59
9. Pair Programming 71
10. Hiring a Foreman, Inheriting Resources, and Occasions 81
11. Going Client-side with Leaflet, Backbone, and Jasmine 89
12. Getting to Occasions 105
13. Bubbly Map Events 115
14. Retrospective 125

RAILS DEEP DIVE

BY **GLENN GOODRICH**

Rails Deep Dive

by Glenn Goodrich

Cover Illustrator: Matthew Magain **Cover Designer**: Alex Walker

Notice of Rights

Notice of Liability

The authors and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors, will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: business@sitepoint.com

ISBN 978-0-9872478-9-6

Table of Contents

Preface xi

What's in this book? xi

Code Samples xii

Chapter 1 Ruby Version Manager 1

Installing RVM 1

Chapter 2 Installing Rails 7

Selecting the Interpreter 7

Installing Rails 8

RubyGems 9

Other Gems Installed 10

MultiJSON 11

ActiveSupport 11

Builder 11

i18n 11

BCrypt Ruby 11

ActiveModel 12

The Rack Gems 12

Hike 12

Tilt 12

Sprockets 12

TZInfo 13

Erubis 13

ActionPack 13

Arel 13

ActiveRecord 13
ActiveResource 14
MIME Types 14
Polyglot 14
Treetop 14
Mail and ActionMailer 14
Thor 15
Rack SSL 15
RDoc 15
Railties 15
Bundler 15
Rails 16

Chapter 3 App Generation 17
Ruby Path (`-r, --ruby`) 18
Application Builder (`-b, --builder`) 19
Application Template (`-m, --template`) 19
Things You Can Skip 19
Specify a Database (`-d, --database`) 20
Specify a Rails Location 20
Specify a JavaScript library (`-j, --javascript=JAVASCRIPT`) 21
Runtime Options 21

Chapter 4 Application Setup: Loccasions 23
User Stories 23
Gems 24
Client-side Stuff 25
Testing 25
Source Control 26

Other Resources 26
The Starting Line 26

Chapter 5 Home Page 29
Mocking Up the Home Page 29
Prepare the Test Environment 30
Setup RSpec 30
Our First Test 32

Chapter 6 Authentication 37
Create a Branch 37
Write the Test 38
Set up Devise 39
Decision Point: User Names 43
Test Sign In 45

Chapter 7 Spork, Events, and Authorization 49
Event Model 50
Adding Spork 52
Back to Testing 53
Testing That a User Has Events 53
Events Controller 54
Wrap Up 58

Chapter 8 Making Events 59
CRUDdy Events 59
Creating Events 60

Clean up the Signed In Navigation 62
Adding More CRUD to Events 63
MUST DESTROY EVENTS 67

Chapter 9 Pair Programming 71
Let There Be (Evan) Light 72
Am I Worthy? 72
The Day Arrives 72
Revelations 73
Oh Yeah, We're Supposed to Program 74
Feature of the Day 76
Okay, Okay, the ACTUAL Code 77
Time Flies 78
Go and Pair 79

Chapter 10 Hiring a Foreman, Inheriting Resources, and Occasions 81
Hiring a Foreman 81
Occasions 83
Changing Our Spork Configuration 85
You Say ~~Potatoe~~ "Hurry up", and I Say ~~Potahtoe~~ "Occasions Controller" 86
Inherited Resources 86
Loccasions.map do { |its| about.time()} 88

Chapter 11 Going Client-side with Leaflet, Backbone, and Jasmine 89
Libraries, Frameworks, and Maps, OH MY! 89

Setup 91
Client-side Directory Structures, and the Women Who Love Them 94
Setup Complete, Now What? 95
Gentleman, Right Now on Stage 3, Put Your Hands Together for JAAASSSMMIIIIINE 96
I'm the Map[View]! 99
Do You Know the Way to Map, Jose? 100
Start Me Up 102
Update 102
My Blogger Went All Over the Place and All I Got Was This Lousy Map 103

Chapter 12 Getting to Occasions 105
Deleting Events 110
One Event at a Time 110
Finally, an Occasion for Occasions 112

Chapter 13 Bubbly Map Events 115
Responding to Map Clicks 115
Change the Event Show View 118
Remove the CreateOccasionView Call from EventRouter 118
Create a CreateOccasionView When the Map is Clicked 119
More Housekeeping 121
Basic Occasion Functionality 124

Chapter 14 Retrospective 125
What is a Retrospective? 126
What Went Wrong? 127
What Went Right? 128

How to Get Better? 129
What's the Plan? 129

Preface

This book started life simply, as a series of blog posts on Rubysource.com.[1] When I came up with the concept, I wanted a series about Rails that was beyond the "blog in 15 minutes" examples, dealing with the decisions, issues, and challenges that pop up when creating a "real" Rails application. Also, I wanted to level up, so to speak, in my own Rails development. At the time, I was much more on the beginner side of intermediate, which I felt was an advantage in writing the series. Once you've improved your knowledge of a technology, it's difficult to remember what beginners need to help them improve as well. In this way, I believe Loccasions and the Rails Deep Dive series was successful. After a year of using Rails in my day job, I am not sure I could write a beginner/intermediate series.

As with anything I write (code or articles) looking back on this series, I can only see the places it needs improvement. I was tempted, for this book, to almost rewrite each post to make it more accurate or better or whatever. However, I think that would remove the original goal of what I was trying to do, which is write a deeper Rails tutorial series from the perspective of someone who was learning (a great deal) along the way. As such, you may find issues or may disagree with an approach taken by the series. That's OK. Actually, that's great. Especially if you publish your approach to the problem. It is in this way that the Ruby and Rails community grows and learns together.

I'd like to thank the great folks and SitePoint and RubySource for being desperate enough for a Ruby writer to allow me to publish my thoughts. The experience has led to a metamorphosis of my career and life. I'd like to especially thank Aaron Osteraas for his never-ending patience, almost constant availability on Skype, and (what must have been difficult) much-needed encouragement.

What's in this book?

This book will guide you in creating a Rails application. It will focus on setting your system up properly (for those systems that support it) and will fly a little lower than the typical 50,000 foot level of many tutorials.

[1] http://rubysource.com

By the end of the book, you'll have learned how to:

- set up Ruby Version Manager (RVM) to maintain sandboxed[2] development environments
- install Ruby 1.9.3
- install Rails 3.1
- create a Rails application
- determine what Rails IDEs exist, as well as their pros and cons
- generate a resource for your application to create, retrieve, update, and delete
- modify a view template
- know what's next

While Rails is often touted as a good web development framework for beginners, there are rumblings in the community that Rails has outgrown that moniker;[3] the changes in Rails 3.1 are a result of a more mature community being in need of an advanced web framework.

We're going to focus on Rails 3.1 (RC4 at the time of writing), highlighting some of the changes at 3.1 as we go. I'll assume that you're comfortable on the command line; that is, "curl" is more than a Canadian verb.

Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

[2] http://en.wikipedia.org/wiki/Sandbox_%28software_development%29
[3] http://intridea.com/2011/6/16/what-if-rails-isnt-for-beginners-anymore

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➥ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/blogs/2007/05/28/user-style-she
➥ets-come-of-age/");
```

Chapter 1

Ruby Version Manager

In this chapter, we'll start from scratch and end up with a Rails application. Although there are many posts out there on this subject, basic Rails tutorials—especially in the wake of the Rails 3.1 changes—fall into a more-the-merrier category. So let's get started with the first step.

Installing RVM

I can't stress enough how invaluable Ruby Version Manager (RVM) is to Ruby and Rails development. In a nutshell, RVM basically allows you to create as many Ruby sandboxes as you need for development or projects or whatever. You can separate versions of Ruby as well as sets of gems (called, funnily enough, **gemsets**), so you can do this tutorial without hawking your base Ruby or gems. Then, you can just delete the gemset and/or the version of Ruby if it's no longer needed, or create a new Rails 3.0.8 application so that you can live in the present. Ruby development starts with RVM, so learning how to use it is a best practice you should pick up now. Unfortunately for my Windows friends, you don't have an RVM.

First, let's get some terminology out of the way. RVM refers to the different interpreters as "rubies". Each rubie has one or more gemsets associated with it. You cannot have a single gemset serving two different rubies, but you can import/export or copy gemsets between rubies. Here, we are going to use RVM to install the latest 1.9.2 rubie and create a gemset for our Rails applications.

Now that we are speaking the same language, let's install[1] RVM. Looking at the prerequisites[2], most of the things you'll need are core to Mac OS X and Linux. If you have not installed Git, then you should do so now[3], as Git is the source control of most open source and Rails developers. Also, you'll need the gcc compiler to allow RVM to compile different Ruby interpreters in your environment. For Mac users, this means installing XCode (you can install Xcode 3[4] for free or pay $5 for Xcode 4 in the Mac App Store. Either one is fine with RVM.). On Linux, make sure you have make and the C compiler, which you can install with:

```
sudo apt-get install build-essential
```

and:

```
curl sudo apt-get install curl
```

Okay, that should handle the prereqs.

There are a couple of ways to install RVM, either single-user or multi-user. We will install it in the single-user fashion, which is the way to go for developers. The multi-user install of RVM is more for server administrators, allowing for the system wide install of rubies and gemsets.

Installing RVM is just running a bash script at the command line. So, fire up your terminal and type:

```
bash < <(curl -s https://rvm.beginrescueend.com/install/rvm)
```

[1] https://rvm.io/rvm/install/
[2] https://rvm.beginrescueend.com/rvm/prerequisites/
[3] http://book.git-scm.com/2_installing_git.html
[4] http://developer.apple.com/devcenter/mac/

This will run the rvm install bash script in your current session, installing in your home directory at ~/.rvm. Also, the output of the script will have some instructions for your .bashrc (or .profile or .bash_profile) startup scripts. RVM has to load into your shell environment when you open a terminal, so add this to the end of your startup script:

```
[[ -s "$HOME/.rvm/scripts/rvm" ]] && . "$HOME/.rvm/scripts/rvm"
```

If you are interested there is a good explanation[5] of what that statement does on startup. Once you have modified the startup script, you can either reload your startup script:

```
source ~/.bash_profile
```

Or close your terminal and open a new one. Now type:

```
type rvm | head -1
```

And you should see:

```
rvm is a function
```

Now, we can go get some rubies (YAR! That makes me feel like a pirate!):

First, let's review our choices, which can be seen in Figure 1.1:

```
rvm list known
```

[5] [[-s "$HOME/.rvm/scripts/rvm"]] && . "$HOME/.rvm/scripts/rvm"

```
# MRI Rubies
[ruby-]1.8.6[-p399]
[ruby-]1.8.6-head
[ruby-]1.8.7[-p302]
[ruby-]1.8.7-head
[ruby-]1.9.1-p243
[ruby-]1.9.1[-p376]
[ruby-]1.9.1-p429
[ruby-]1.9.1-head
[ruby-]1.9.2-preview1
[ruby-]1.9.2-preview3
[ruby-]1.9.2-rc1
[ruby-]1.9.2-rc2
[ruby-]1.9.2[-p0]
[ruby-]1.9.2-head
ruby-head

# JRuby
jruby-1.2.0
jruby-1.3.1
jruby-1.4.0
jruby-1.5.1
jruby-1.5.2
jruby[-1.5.3]
jruby-head

# Rubinius
rbx-1.0.1
rbx[-1.1.0]
rbx-head

# Ruby Enterprise Edition
ree-1.8.6
ree[-1.8.7]
ree-1.8.6-head
ree-1.8.7-head

# MagLev
maglev[-24407]
maglev-head

# Shyouhei head, the default mput
mput[-head]

# Mac OS X Snow Leopard Only
macruby-0.6
macruby-[0.7.1] # the default macruby
macruby-nightly
macruby-head      # Build from the macruby git repository

# IronRuby -- Not implemented yet.
ironruby-0.9.3
ironruby-1.0-rc2
ironruby-head
```

Figure 1.1. RVM Known Interpreters

Wow. I had no clue there were that many. Let's install them all … BWA-HAHAHAHAHA … no, wait, (smooths back hair) let's just install one. I vote for 1.9.2, and my vote is the only one that counts:

```
rvm install 1.9.2
```

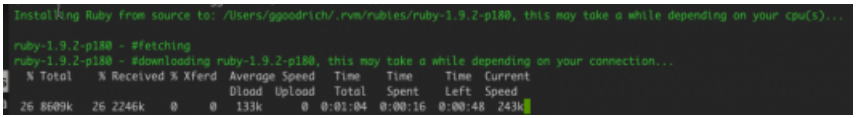

Figure 1.2. Installing Ruby 1.9.2

As you can see in Figure 1.2, this installs the latest patch level of MRI (Matz's Ruby Interpreter), which is 180 in this case. With RVM, you can target patch levels or the latest (head) stable build. Either one serves our purposes here, so p180 it is. When the install is complete, RVM will install the "default" gemsets, which you can define in `~/.rvm/gemsets/default.gems`. Currently, all I have in there is rake, but you can add others as needed.

We have to tell RVM that we want to use that newly loaded 1.9.2 Ruby interpreter. This is done with:

```
rvm use 1.9.2
```

Awesome. Now, if you type `gem list`, you should just see the default gems. My results are seen in Figure 1.3:

Figure 1.3. RVM Gem List

This validates that rake is the only gem in my current rubie. Obviously, we're going to want Rails installed, but before we do that, let's create a gemset for this tutorial called "rubysource":

```
rvm gemset create rubysource
```

RVM tells us that our gemset is created, now we have to use that. Can you guess how that's done? If you said:

```
rvm gemset use rubysource
```

… then you're a winner! So, as we're working, how do we know what rubie and gemset combination is the current one? If only RVM had an easy way to give us that kind of (hint, hint) info:

```
rvm info
```

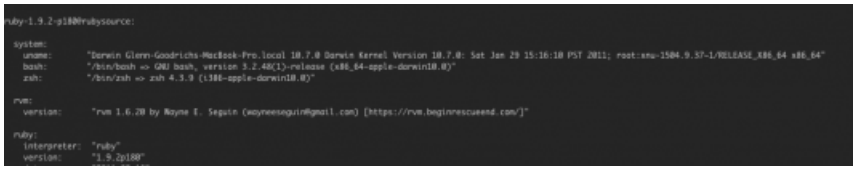

Figure 1.4. RVM Info

As seen in Figure 1.4, that command gives us all kinds of great information, like which interpreter we're using, the current gemset, where the binaries for the current rubie reside, and the relevent Ruby environment variables. It's worth noting that the syntax for indicating rubie and gemset is `rubie@gemset`, which you can also use as a shortcut when switching rubies/gemsets. For example, if you type:

```
rvm use 1.9.2@rubysource
```

… it will switch the current ruby to 1.9.2 and the current gemset to rubysource. For homework, go figure out how to use that shortcut to automatically create the gemset if it's yet to exist.

So, that's RVM in a nutshell. Next, we'll finish installing Rails 3.1, as well as create our Rails app. In the meantime, feel free to play with RVM and get comfortable using it for all your Ruby development.

Chapter 2

Installing Rails

This book attempts to go a bit deeper when starting with Rails, so we'll now cover some of the options available when first creating your world-changing Rails application, as well as the gems that are installed with Rails.

Selecting the Interpreter

First things first, make sure you open a terminal and switch to our RVM Ruby interpreter and gemset, which is MRI 1.9.2 and *rubysource*, respectively. We do that with:

```
rvm 1.9.2@rubysource
```

… and you can verify with a quick `rvm info`. As you probably know, the way to generate a new Rails application is by typing this at the command prompt:

```
rails new application_name
```

Installing Rails

What you may not know is where *Rails* executable lives. In fact, it may surprise you to know that the only thing the Rails gem includes is the *rails* executable. The Rails gem has many dependencies, which are satisfied by other gems, but the actual Rails gem is just an executable. Let's install it now:

```
gem install rails
```

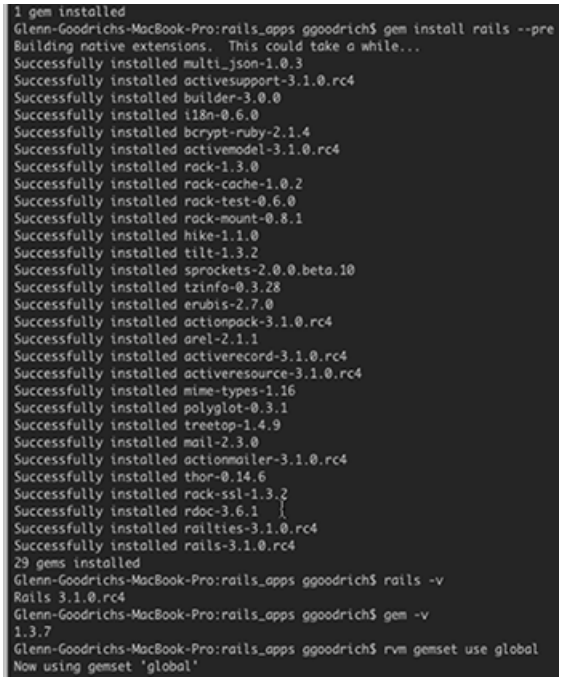

Figure 2.1. Filler

The `--pre` option tells RubyGems to install the latest prerelease gem, which is not a stable version of Rails. In my case, I got Rails 3.1 RC4. "RC4" stands for Release Candidate 4, which was the last of the release candidates before Rails became stable. We can see the general release cycle that Rails follows by looking at the tags on GitHub.[1]

[1] http://github.com/rails/rails

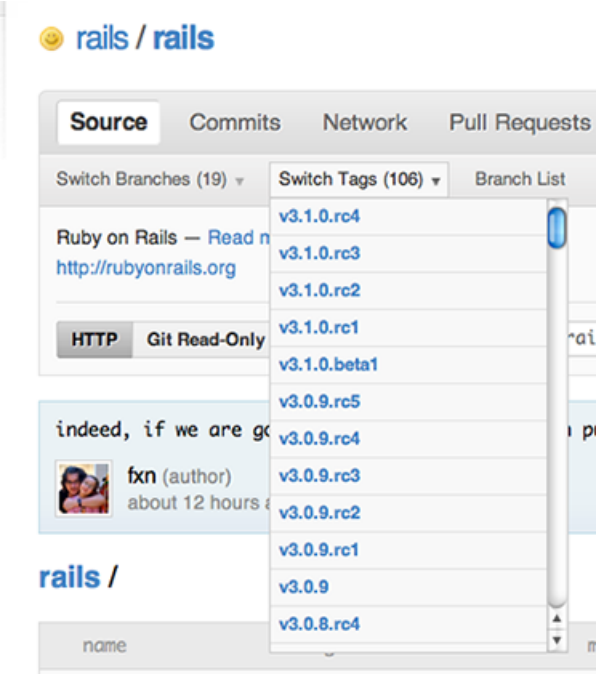

Figure 2.2. Rails 3.1 Release Candidates

Where we can see that Rails follows a pattern of starting with a "beta" release, followed by 4 or 5 release candidates, before going stable. Thanks to RVM, we can muck about with any of the pre-release software without contaminating the rest of our development environment.

Now you know what the "–pre" option does when installing the Rails gem, but what other options are there? "gem install" takes many options.[2]

RubyGems

RubyGems allows us to specify items, such as, a specific version (which we've seen), an install destination, whether or not to install documentation, whether or not to install dependencies, as well as specifying a source for searching for gems. Looking over the available options, it's easy to see how RVM leverages RubyGems to keep gemsets isolated. Finally, you can put any of these options into your **~/.gemrc** file if you find yourself typing the same options over and over again. As a special bonus,

[2] http://docs.rubygems.org/read/chapter/10#page33

here is a way to significantly speed up your gem installs by setting options in your Gem configuration file.[3]

Other Gems Installed

When we installed Rails, it also installed several other gems. What are those gems? What is their purpose?

```
Successfully installed multi_json-1.0.3
Successfully installed activesupport-3.1.0.rc4
Successfully installed builder-3.0.0
Successfully installed i18n-0.6.0
Successfully installed bcrypt-ruby-2.1.4
Successfully installed activemodel-3.1.0.rc4
Successfully installed rack-1.3.1
Successfully installed rack-cache-1.0.2
Successfully installed rack-test-0.6.0
Successfully installed rack-mount-0.8.1
Successfully installed hike-1.1.0
Successfully installed tilt-1.3.2
Successfully installed sprockets-2.0.0.beta.16
Successfully installed tzinfo-0.3.29
Successfully installed erubis-2.7.0
Successfully installed actionpack-3.1.0.rc4
Successfully installed arel-2.1.3
Successfully installed activerecord-3.1.0.rc4
Successfully installed activeresource-3.1.0.rc
Successfully installed mime-types-1.16
Successfully installed polyglot-0.3.1
Successfully installed treetop-1.4.9
Successfully installed mail-2.3.0
Successfully installed actionmailer-3.1.0.rc4
Successfully installed thor-0.14.6
Successfully installed rack-ssl-1.3.2
Successfully installed rdoc-3.8
Successfully installed railties-3.1.0.rc4
Successfully installed bundler-1.0.15
Successfully installed rails-3.1.0.rc4
30 gems installed
```

Figure 2.3. Gems Installed with Rails

Let's briefly run through each one.

[3] http://www.rubyinside.com/speed-up-gem-installs-significantly-1605.html

MultiJSON

MultiJSON[4] (by Intridea, who do great stuff) allows for multiple JSON backends, detecting and leveraging the best one. In the case of the vanilla Rails install, it uses the `json_pure` gem.

ActiveSupport

ActiveSupport[5] is "a collection of various utility classes and standard library extensions that were found useful for Rails. All these additions have hence been collected in this bundle as way to gather all that sugar that makes Ruby sweeter." This means, in essense, that ActiveSupport is THE building blocks of Rails, including abstractions for caching, JSON support, unicode support, and notifications. It also defines ActiveSupport::Railtie, which is one of the ways to extend your Rails application. The breadth of ActiveSupport is far too large to cover here, so check it out in your spare time.

Builder

Builder[6] provides a Domain Specific Language (DSL) for generating markup.

i18n

The i18n[7] gem provides all the localization support for Rails. As you can imagine, this is a large topic, and one worth studying as you grow in your Rails knowledge.

BCrypt Ruby

BCrypt[8] is new to Rails 3.1, providing encryption for securing persisted passwords.[9]

[4] http://intridea.com/2010/6/14/multi-json-the-swappable-json-handler?blog=company
[5] http://as.rubyonrails.org/
[6] http://ruby.about.com/od/gems/a/builder.htm
[7] https://github.com/svenfuchs/i18n
[8] http://bcrypt-ruby.rubyforge.org/
[9] http://bcardarella.com/post/4668842452/exploring-rails-3-1-activemodel-securepassword

ActiveModel

ActiveModel[10] provides the the interface for models in Rails. ActiveModel was provided starting in Rails 3.0, allowing the developer to go through the buffet line of the syntactic sugar provided to models in Rails and only put the desired bits on the plate.

The Rack Gems

Rack[11] provides the interface to the web server from Ruby applications. One of the items supported by Rack is *middleware*, allowing for bits of code to be dropped into the web request/response pipeline and provide functionality. rack-cache[12] is middleware providing HTTP caching. rack-test[13] provides an API for testing Rack apps (which is what Rails is) in the form of a nice DSL. rack-mount[14] provides the routing for Rails, which drives the nice RESTful interface of a standard Rails application.

Hike

Hike[15] handles the load and search paths for Rails.

Tilt

Tilt[16] provides an interface to different Ruby templating engines, like ERB and Haml.

Sprockets

Sprockets[17] is new at Rails 3.1, providing the new asset packaging pipeline for JavaScript and CoffeeScript, as well as SASS and CSS.

[10] https://github.com/rails/rails/tree/master/activemodel
[11] http://rack.rubyforge.org/
[12] http://rtomayko.github.com/rack-cache/
[13] https://github.com/brynary/rack-test
[14] http://rubydoc.info/gems/rack-mount/0.8.1/frames
[15] http://rubydoc.info/gems/hike/1.1.0/frames
[16] http://net.tutsplus.com/tutorials/ruby/ruby-for-newbies-the-tilt-gem/
[17] https://github.com/sstephenson/sprockets

TZInfo

TZInfo[18] is a timezone library for Ruby.

Erubis

Erubis[19] is an implementation of eRuby, which is the Ruby expressions between the `<% %>` in Ruby views.

ActionPack

ActionPack[20] is kind of a big deal. The rubydoc for ActionPack states that it "provides the view and controller layers in the MVC paradigm". You could argue that it is two-thirds of Rails. It includes ActionDispatch (routing and HTTP goodness, like caching, sessions, and cookies), ActionController (provides the base class for Rails controllers). ActionView (handles the views and rendering of formats like HTML or ATOM feeds).

If you want to really learn about Rails, doing a deep dive on ActionPack will take you far.

Arel

Arel[21] is what gives ActiveRecord its cool syntax. It is a "SQL AST manager," where "AST" meaning "Abstract Syntax Tree." An AST[22] is one of those super-nerd concepts that separates the Geniuses from the rest. Explaining an AST is *way* outside the scope of this article (not to mention the scope of my brain).

ActiveRecord

ActiveRecord[23] is the default object-relational mapper (ORM) used by Rails. Basically, it maps your model to the database structure. You can use other ORMs such as DataMapper[24] if you prefer.

---

[18] http://tzinfo.rubyforge.org/
[19] http://www.kuwata-lab.com/erubis/
[20] http://rubydoc.info/gems/actionpack/3.0.9/frames
[21] https://github.com/rails/arel
[22] http://en.wikipedia.org/wiki/Abstract_syntax_tree
[23] http://ar.rubyonrails.org/
[24] http://ar.rubyonrails.org/

ActiveResource

ActiveResource[25] is for mapping RESTful[26] resources as models in a Rails application. Understanding REST is key to being a good web developer. A great place to start learning REST is in the O'Reilly book, RESTful Web Services.[27]

MIME Types

The MIME Types[28] gem is used by Rails to identify the MIME type of a request, such as mapping `text/html` to HTML, etc.

Polyglot

Polyglot[29] registers file extensions to be used with Ruby *require* statements. So, if you wanted to load files with a **.funk** extension, the **.funk** extenstion can be registered with Polyglot. Then, `require 'wegotthe'` would find a file named **wegotthe.funk**.

Treetop

Treetop[30] is another gem that handles a guru-level concept, allowing the developer to create syntax and parse "expression grammars" easily. Treetop is used, in its most basic way, to add syntax to a language (I would explain more, but there isn't the room for it here).

Mail and ActionMailer

The Mail[31] and ActionMailer[32] gems focus on (you guessed it) sending mail from Rails. Mail handles the generation, sending, parsing of e-mail, while ActionMailer creates the "mailers" (think: controllers for e-mail) and views for mail templates.

[25] http://api.rubyonrails.org/classes/ActiveResource/Base.html
[26] http://en.wikipedia.org/wiki/Representational_State_Transfer
[27] http://oreilly.com/catalog/9780596529260
[28] http://mime-types.rubyforge.org/
[29] http://polyglot.rubyforge.org/
[30] http://treetop.rubyforge.org/
[31] https://github.com/mikel/mail
[32] http://api.rubyonrails.org/classes/ActionMailer/Base.html

Thor

Thor[33] helps build command line tools and utilities. It is one of those gems you would do well to research and incorporate into your Ruby arsenal (by Odin's Beard!).

Rack SSL

The Rack SSL[34] gem provides a middleware to support Transport Level Security (TLS), the most common is Secure Socket Layer (SSL). For more on its use, see Daniel Morrison's article, SSL with Rails.[35]

RDoc

RDoc[36] simplifies the creation of HTML-based documentation and it is used to document Rails itself.

Railties

Railties,[37] starting at 3.0, has been included in core Rails. Railties are (arguably) THE way to extend Rails, and many of the major components, like ActionMailer, ActionController, and ActionView are Railties. Understanding how Railties work is another key task for the budding Rails developer. Here is a good article on Railtie & Creating Plugins[38] to whet your appetite.

Bundler

Bundler[39] is the way Rails manages its gem dependencies. Bundler uses the Gemfile in the root of the Rails app to make sure all the necessary gems are available and any conflicts are identified. Spend some time on the Bundler site[40] to see how many options for loading gems are available.

[33] https://github.com/wycats/thor
[34] https://github.com/josh/rack-ssl
[35] http://collectiveidea.com/blog/archives/2010/11/29/ssl-with-rails/
[36] http://docs.seattlerb.org/rdoc/
[37] http://rubydoc.info/gems/railties/3.0.9/frames
[38] http://www.igvita.com/2010/08/04/rails-3-internals-railtie-creating-plugins/
[39] http://gembundler.com/
[40] http://gembundler.com/

Rails

Previously, I mentioned that the Rails[41] gem only includes the *Rails* executable, which is slightly misleading. If you inspect the source on GitHub, you can see that it includes many of the gems discussed in these two articles. You may want to start your code browsing here.

We've now covered gem dependencies in Rails 3.1. Even though Rails has a reputation of being a simple web framework, there is a lot of work that goes into building that simplicity.

Next, we'll go over each of the options for creating a new rails application and why you might want to use them. My goal is to generate our Rails application so that we have something tangible. I hope you're enjoying this more in-depth look at starting with Rails.

[41] http://github.com/rails/rails

Chapter 3

App Generation

The `rails new` command is probably the first part of the Rails command line we all learned. Give it an application name, and that command will create a *fully functional* (sic) Rails web application. Much of the convention over configuration plays out in the generated structure, with the app directory holding our model, view, controller, and (now, at Rails 3.1) client–side assets.

My added emphasis on "fully functional" in the previous statement exists to point out that the site is hardly production-ready. More often than not, there are database, security, or other concerns, but the generated app is a great foundation. So, how can we tweak this foundation to put us a bit further us down more specific development paths?

The options available to `rails new` are shown in Figure 3.1.

```
Usage:
  rails new APP_PATH [options]

Options:
  -r, [--ruby=PATH]              # Path to the Ruby binary of your choice
                                 # Default: /Users/ggoodrich/.rvm/rubies/ruby-1.9.2-p290/bin/ruby
  -b, [--builder=BUILDER]        # Path to a application builder (can be a filesystem path or URL)
  -m, [--template=TEMPLATE]      # Path to an application template (can be a filesystem path or URL)
      [--skip-gemfile]           # Don't create a Gemfile
      [--skip-bundle]            # Don't run bundle install
  -G, [--skip-git]               # Skip Git ignores and keeps
  -O, [--skip-active-record]     # Skip Active Record files
  -d, [--database=DATABASE]      # Preconfigure for selected database (options: mysql/oracle/postgresql/
                                  sqlite3/sqlite3/frontbase/ibm_db/jdbcmysql/jdbcsqlite3/jdbcpostgresql)
                                 # Default: sqlite3
  -j, [--javascript=JAVASCRIPT]  # Preconfigure for selected JavaScript library
                                 # Default: jquery
  -J, [--skip-javascript]        # Skip JavaScript files
      [--dev]                    # Setup the application with Gemfile pointing to your Rails checkout
      [--edge]                   # Setup the application with Gemfile pointing to Rails repository
  -T, [--skip-test-unit]         # Skip Test::Unit files
      [--old-style-hash]         # Force using old style hash (:foo => 'bar') on Ruby >= 1.9

Runtime options:
  -f, [--force]    # Overwrite files that already exist
  -p, [--pretend]  # Run but do not make any changes
  -q, [--quiet]    # Supress status output
  -s, [--skip]     # Skip files that already exist

Rails options:
  -h, [--help]     # Show this help message and quit
  -v, [--version]  # Show Rails version number and quit

Description:
    The 'rails new' command creates a new Rails application with a default
    directory structure and configuration at the path you specify.

Example:
    rails new ~/Code/Ruby/weblog

    This generates a skeletal Rails installation in ~/Code/Ruby/weblog.
```

Figure 3.1. Options for rails new

Let's go through each one and discuss what it does, and why you may want to use it.

Ruby Path (`-r, --ruby`)

The PATH to the Ruby binary that will be used for this Rails application. You might use this to test your app against another version of Ruby or lock it into certain version. On the development side, this is unnecessary due to tools like RVM, but your production or staging environments may have multiple Ruby versions.

Application Builder (`-b, --builder`)

In the last section, I briefly mentioned the ability to specify an application builder. The builder is responsible for creating the application structure, so providing your own builder allows you to change that structure and content as you see fit. In short, you create a class that inherits from `Rails::AppBuilder` and overrides the things you want to change. You can specify a different Test framework (say, RSpec) or automatically include your favorite gems or rake tasks … well, you get the idea. Here is the best post I could find on the process[1] (thanks Mike Barinek!).

Application Template (`-m, --template`)

Application Templates are another way to change how the application is generated. In this case, the parameter you pass to `-m` is a Ruby file that allows the addition of gems or initializers to a generated Rails application. The major difference between using a builder and using a template is *when* the customizations occur. Using a builder, you could modify the structure of the app as it is generated (for example, I could call the lib directory "bibloteca"), which you can't (read: shouldn't) do with a template.

Application Templates are more about selecting the right gems, running rake tasks, and adding initializers to the base application structure. Templates seem to be the much more popular method for customizing Rails application generation, and there is even RailsWizard[2] to make creating templates a breeze. Also, I feel any mention of Rails application templates is incomplete without highlighting the awesome work of Daniel Kehoe and his RailsApp[3] repository.

Things You Can Skip

Many of the options to the `rails new` command allow you to NOT do something.

- (`--skip-gemfile`): Do not create the Gemfile, because I am bringing my own or I am not using Bundler.

[1] http://pivotallabs.com/users/mbarinek/blog/articles/1437-rails-3-application-builders
[2] http://railswizard.org/
[3] http://railsapps.github.com/

- (`--skip-bundle`): Do not run `bundle install` after generating the app, because I want to do something before Bundler does its thing.
- (`-G, --skip-active-record`): Don't include ActiveRecord, because I am using a different ORM or a NoSQL database.
- (`-J, --skip-javascript`): Don't supply the default JavaScript files, because I am bringing my own.
- (`-T, --skip-test-unit`): Don't create the default Test::Unit test files (BTW, this doesn't even create the *test* directory), because I am using a different test framework (like RSpec).
- (`-F, --skip-git`): Don't create Git files (**.gitignore** and **.gitkeep**) because I am using a different source control system or (NOOOO!) none at all. It's interesting to note, that, in the official guides,[4] a `--git` option is mentioned. From what I can tell, this option is not valid (and is ignored by the command) but must be a hangover from when the Git files were not created by default.

Specify a Database (`-d, --database`)

By default, Rails presumes a new application will be using a database and that database will be SQLite. This allows for easy spiking of Rails apps without brining in the overhead of a typical RDBMS. However, the cases for which you want to use a different RDBMS, you can specify one of them using this option and supplying one of the supported parameters. Those supported parameters are: `mysql/oracle/postgresql/sqlite3/frontbase/ibm_db/jdbcmysql/jdbcsqlite3/jdbcpostgresql`. Supply one of these and your Gemfile and **config/database.yml** will be generated appropriately.

Specify a Rails Location

There are two options (`--dev` and `--edge`) that allow you to point to a particular version of Rails. Using `--dev` that allows you to point to a local Rails Git repository. In this case, the version of Rails from your local repository either needs to be in the

[4] http://guides.rubyonrails.org/command_line.html#rails-with-databases-and-scm

PATH or needs to be specified as the full path when generating the application. For example:

```
ruby /path/to/rails/bin/rails new theapp --dev # from
➥ rubyonrails.org
```

The `--edge` option will point to the HEAD version of the main Rails GitHub repository. In either case, you want to freeze your Rails version.

Specify a JavaScript library (`-j, --javascript=JAVASCRIPT`)

If you don't want to use a different option beside jQuery, you can specify 'prototype' to the `-j` option and get an app with the Prototype library.

Runtime Options

The various runtime options affect the feedback of the command, as well as how to treat existing files. The parameters `-f` and `--force` will overwrite any existing files. You might want to do this if you've corrupted some of the base files and (for shame!) aren't using source control. The parameters `-s` and `--skip` are the opposite of *force* and will not overwrite any existing files. Perhaps my favorite command line switch is `-p, --pretend`, which doesn't actually create anything, but still emits the output of the command so you can see what it *would* do. And finally, `-q, --quiet` suppresses all output. I like to run `rails new existential_app -p -q` and wonder aloud if it every really existed … To finish up, let's generate an application to be used by the remainder of this book. Sadly, we'll just use all the defaults.

```
rails new deep_dive
```

The output of this command can be seen in this GitHub gist.[5]

Before we go, let's make sure we are up and running. Change into the **deep_dive** directory and type `rails s`. You should see output similar to Figure 3.2:

[5] https://gist.github.com/1117147

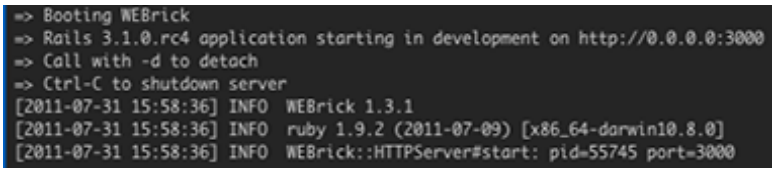

Figure 3.2. Rails Server Output

Now if you open http://localhost:3000, you should see the familiar “Ruby on Rails: Welcome Aboard” page. We’ll be deleting this with extreme prejudice in the next chapter, where we’ll really start creating our own app.

Chapter 4

Application Setup: Loccasions

Up to this point, we've focused on digging down into the entrails of the framework, attempting to uncover some of the ways that Rails accomplishes its magic. Going forward, we want to create a Rails 3.1 application, focusing on how we'd set up the application, perform the development, and deploy the application.

Our application will be called *Loccasions*. The purpose of Loccasions is to allow users to create *Events and Occasions*. An Event might be "I cleaned my room" or "It rained" or "A comet sighting." Events contain Occasions, marking a time and place where the Event occurred. The application will present the occasions on a map, allowing the user to see how often and where an Event occurs. The idea is simple and the use case specific, so creating the app should be a snap (he says, knowing he will hit roadblocks …)

User Stories

When creating a new Rails application (or any application, really) it's a good idea to have some user stories[1] to direct the application and ensure we are staying on

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

task. Normally, you would meet with the client and generate the high level user stories together. The key with user stories is to capture just enough detail to start working, avoiding the "analysis paralysis" that can cripple progress. For Loccasions, we will keep the user stories pretty high level, adding more as we go. Our first stories are:

- As an unregistered user, I want to see the home/landing page
- As an administrator, I want to be able to invite users to Loccasions
- As an invited user, I want to be able to create an account
- As a registered user, I want to be able to create Events
- As a registered user, I want to be able to create Occasions
- As a registered user, I want to see Occasions on a map

I think that is a good start.

Gems

The next decision concerns the gems we are going to leverage to take care of some of our functional needs. Obviously, Loccasions will need some kind of authentication, and the community has great gems in this area. Probably the most well known authentication gem is Devise[2] written by Jose Valim and the incredible folks at Plataformatec. I think using Devise gives us a well-tested gem and a fantastic community for support.

One of the decisions I have made for Loccasions is how persistence will be handled. Rather than go the standard relational database route, like PostgreSQL or MySQL, I have chosen MongoDB for our back-end persistence store. First, I think the Event ==> Occasions model makes a good document db model. Second, I am relatively certain that Loccasions will use the spatial functionality[3] that MongoDB provides. Also, if I am being honest, I really want to use MongoDB in a "realish" Rails app and this is opportunity knocking.

[2] https://github.com/plataformatec/devise
[3] http://www.mongodb.org/display/DOCS/Geospatial+Indexing

The use of MongoDB leads to another area where gems can help. In this case, I looked at MongoMapper[4] and Mongoid[5] and settled on Mongoid because it seems to have slightly better support for the spatial parts of MongoDB, as well as the existence of mongoid_spacial.[6]

It's worth noting that this conclusion is based on a few minutes of looking at both sets of docs, so there may be better options. However, this is how decisions are made, sometimes, when starting an application. Pick a direction and go. Also, it is likely that we'll run into version issues between gem dependencies. If this happens, we may have to either sacrifice a gem or fork it and fix the issue ourselves—in any case, you might learn something.

Client-side Stuff

I am relatively sure that we'll use a decent amount of JavaScript in Loccasions. My initial vision sees each Event page as a Single Page Application, allowing the user to create Occasions and add them to the map. The map and list of Occasions will stay in sync, which means we have two "views with our view". This vision pushes me toward a client-side framework, and my current favorite is Backbone.js.[7] The rails-backbone gem[8] simplifies using Backbone with Rails, so we'll put that gem on the list as well. (Note: There are two Backbone gems that are very closely named, ensure you are using codebrew's gem.)

Also, I have become a fan of Haml[9] so I think we'll use Haml instead of ERB for our view templates.

Testing

We will, as much as possible, employ a test-driven approach to creating Loccasions. In essence, this means we'll write tests first to drive the design and implementation of the app. With that in mind, we need to select a testing approach, and I've decided

[4] http://mongomapper.com/
[5] http://mongoid.org/
[6] https://github.com/ryanong/mongoid_spacial
[7] http://documentcloud.github.com/backbone/
[8] https://github.com/codebrew/backbone-rails
[9] http://rubysource.com/an-introduction-to-haml/

on RSpec and Capybara. Also, there is a gem that integrates Mongoid and Rspec (mongoid-rspec) that will simplify our testing.

The test-driven approach extends to the client-side of the application, as well, and using something like Jasmine[10] keeps the specification approach consistent.

Source Control

I will be creating a GitHub repository for the Loccasions source. Before you start any development process, you should have a plan for source control. Git makes it criminally easy to get going with SCM, so there is no excuse.

Other Resources

One of the best tools in your Rails toolbelt is the Internet and standing on the shoulders of those that came before. For example, my inspiration for the Devise and Mongoid setup is one of Daniel Kehoe's fantastic tutorials.[11] I am sure we will be scouring the web for help and resources, and I hope to highlight what we find.

The Starting Line

Alright, I think that is enough planning. Time to stop dipping our toes in the water and jump in up to our necks. Of course, we need MongoDB running locally. Go install MongoDB[12] on your platform … I'll wait.

Done? Hopefully you have a default MongoDB setup ready to go. The last thing to do before we generate the application is to create a GitHub repository[13] for our app (see my GitHub repository[14]).

We are getting closer. I am using Rails 3.1 RC5 and Ruby 1.9.2. Also, I am using RVM (see Chapter 1) and I *strongly* recommend you set up RVM and a gemset before continuing:

[10] http://pivotal.github.com/jasmine/
[11] https://github.com/RailsApps/rails3-devise-rspec-cucumber
[12] http://www.mongodb.org/display/DOCS/Quickstart
[13] http://learn.github.com/p/setup.html
[14] https://github.com/ruprict/loccasions

```
rvm use 1.9.2@loccasions --create
# will create and switch to loccasions gemset and Ruby 1.9.2
```

We have a clean gemset, so we need to install a couple of gems before we can get to Rails:

```
gem install bundler
gem install rails --version=3.1.0.rc5 # We want 3.1
```

Remember, we are using MongoDB, so we don't need any ActiveRecord pieces (`-O`) (we won't be using migrations). Also, we are using RSPec, so no need to generate the Test::Unit files (`-T`):

```
rails new loccasions -O -T
cd loccasions
```

Now that we finally have an application structure, we need to pull in the aforementioned gems. Open **Gemfile** in your favorite editor (I use vim, b/c it is fantasmic) and make it look like:

```
source 'http://rubygems.org'
gem 'rails', '3.1.0.rc5'
gem 'devise', "~> 1.4.2"
gem 'mongoid', "~> 2.1.8"
gem 'mongoid_spacial', "~> 0.2.13"
gem 'haml', '~> 3.1.2'
gem 'bson_ext', '~> 1.3.1'
gem 'rails-backbone', "~> 0.5.3"
# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails', "~> 3.1.0.rc"
  gem 'coffee-rails', "~> 3.1.0.rc"
  gem 'uglifier'
end
gem 'jquery-rails'
group :test, :development do
  gem 'rspec-rails', '~> 2.6.1'
  gem 'mongoid-rspec', '~> 1.4.4'
  gem 'capybara', '~> 1.0.1'
```

```
  gem 'factory_girl_rails', '~> 1.1.0'
  gem 'database_cleaner', '~> 0.6.7'
  gem 'jasmine', '~> 1.0.2.1'
end
```

A quick `bundle install` and we are ready to attack our first user story. Before we do that, let's do the initial commit to our Git repository and push it up to GitHub. First, edit the **.gitignore** and make sure it makes sense:

```
.bundle
db/*.sqlite3
log/*.log
tmp/
.sass-cache/
*.swp
.DS_Store
```

I added the `*.swp` and `.DS_Store` lines so that my vim buffers and Mac artifacts don't get added to the repository:

```
git add .
git commit -m "Initial commit"
```

Now, add your GitHub remote repository as 'origin':

```
git remote add origin
➥ https://ruprict@github.com/ruprict/loccasions.git
git push origin master
```

Note: you will need to adjust the origin accordingly. The minute I did that, I realized I had forgotten to create a `.rvmrc` file, so let's do that and push it up as well:

```
rvm --rvmrc --create ruby-1.9.2-p290@loccasions
```

Now, `cd ..` and then `cd loccasions` to make the **.rvmrc** file trusted. It will prompt you to review the file, then type `yes`.

In the next chapter, we'll start with the "unregistered user" story, which should lead us to make decisions about how we'll lay out the app.

Chapter 5

Home Page

We have our application structure. Now, we'll start with our first user story "As an unregistered user, I want to see the home/landing page." There are a few items to point out in this short user story. First, we've identified a role of the application in the form of an *unregistered* user. Additionally, the story tells us that we need to have a home page that is not protected by authorization.

Mocking Up the Home Page

This page is the front door of our application, and anyone can knock on it. At this point, it'd be nice to have a talented web designer on the team who could crank out a simple, elegant page. Unfortunately, we just have me, and my design skills are, um, raw. *coughs*.

I like to sketch out wireframes when someone can't afford a real designer, er, I mean, when I am designing a page. There are a plethora of mockup tools on the interwebs (and there are paper and pencils virtually everywhere) and I've settled on MockupBuilder[1] because my usual tool (Pencil) is on sabbatical or something.

[1] http://mockupbuilder.com/

I have cranked out a quick mockup shown in Figure 5.1.

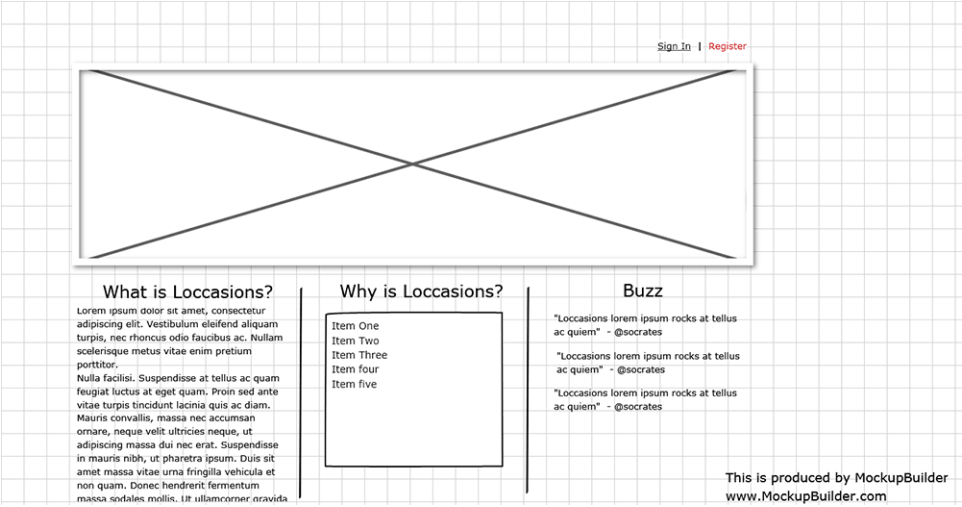

Figure 5.1. Our blueprint

It's a simple, if criminally familiar, home page. There are links to sign in and register, a largish image to promote the site, and a divided area below to put information about the awesome of Loccasions. The layout feeds into a grid system nicely, which is no mistake.

Prepare the Test Environment

Before we take that mockup through the looking glass, we need to setup our test environment and write a test (or two) to validate our page. I mentioned Daniel Kehoe's (@rails_apps[2]) excellent tutorials[3] in the last chapter, and I am relying heavily on them to set up the test environment for Loccasions.

Setup RSpec

RSpec comes with generators to set up the environment, so open up a terminal, `cd` into the **loccasions** directory and type:

```
rails g rspec:install
```

[2] http://twitter.com/rails_apps
[3] https://github.com/railsapps

The output to this command complained that the Mongoid config didn't exist (ugh, first mistake), so we need to generate it:

```
rails g mongoid:config
```

That gives us a **config/mongoid.yml** that will drive the creation of our databases in MongoDB. Re-running `rails g rspec:install` and rspec is happy (unless, MongoDB is not running) again.

We aren't going to use ActiveRecord in Loccasions, so we have to comment out a couple of lines in the **spec/spec_helper.rb** file. Comment out these lines:

```
# config.fixture_path = "#{::Rails.root}/spec/fixtures"
# config.use_transactional_fixtures = true
```

In between test suites, we want RSpec to clean up the database. The `database_cleaner` gem performs this task for us. Add the following to the **spec/spec_helper.rb** file:

```
# Clean up the database
require 'database_cleaner'
config.before(:suite) do
  DatabaseCleaner.strategy = :truncation
  DatabaseCleaner.orm = "mongoid"
end
config.before(:each) do
  DatabaseCleaner.clean
end
```

Also in the **spec_helper.rb** file, add:

```
require 'capybara/rspec'</code>
```

… to get the capybara RSpec matchers.

The `mongoid-rspec` gem gives us some nice RSpec matchers to help our tests communicate their purpose more effectively. These matchers are added to RSpec by creating a **spec/support/mongoid.rb** (you'll have to create the **support** directory too) file with the following:

```
RSpec.configure do |config|
  config.include Mongoid::Matchers
end
```

That gets our test environment into a state that we can get going. There are still a couple of things to set up, but we'll do that when it's more pressing. Just to make sure that we haven't broken anything, run `rake -T` at the command prompt and make sure you see the RSpec tasks. If all is well, run `rake spec` and RSpec should tell you that it has nothing to do, which is a good thing.

Our First Test

Looking at our current user story, an unregistered user needs to see our home page. How do we measure that we've met this story? What will the home page have that we can confirm and know we're okay? My first thoughts on what to measure include that:

- the page has a **Sign In** link
- the page title is **Loccasions: Home**
- the page has a **What is Loccasions?** section on it

We'll start by using Capybara's new feature syntax[4] in our first *acceptance* test. Create a file called **spec/acceptance/home_page_spec.rb** (you'll have to create the **acceptance** directory as well) with the following content:

```
require 'spec_helper'
feature 'Home Page', %q{
  As an unregistered user
  I want to see the home/landing page
} do
  background do
    # Nothing to do here
  end
  scenario "Home page" do
    visit "/"
    page.should have_link('Sign In')
```

[4] http://jeffkreeftmeijer.com/2011/acceptance-testing-using-capybaras-new-rspec-dsl/

```
    page.should have_selector('title', :content => "Loccasions")
    page.should have_content('What is Loccasions?')
  end
end
```

This spec is very easy to read and looks for the items we are using to measure success. By starting with an acceptance test, we are driving the design from the user's perspective. Running `rake spec` results in the output shown in Figure 5.2:

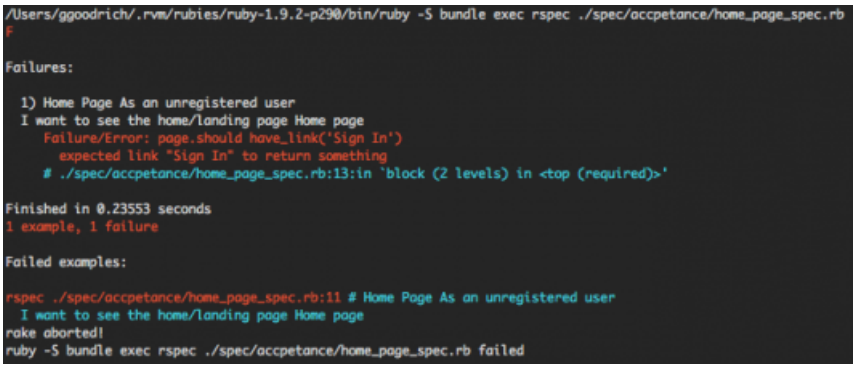

Figure 5.2. Awww ... our first spec ...

As expected, the spec fails trying to find a "Sign In" link. Right now, we are still using the default **index.html** in the *public* directory, so let's delete it. Running the spec again, we now see a "Routing error" which, again, makes sense as we have no root route. Based on previous Rails apps and convention, let's make a "Home" controller with an "index" action:

```
rails g controller Home index
```

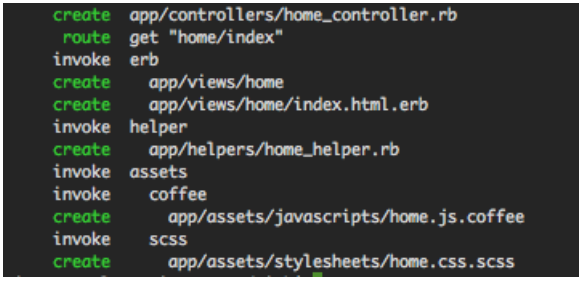

Figure 5.3. ERB? Que pasa?

Hmmm… something is not right. The generator created an ERB template instead of a Haml template. I swear I've read that, at 3.1, including Haml in your Gemfile will automatically make it the default template_engine. Seems I am incorrect. Oh well, the quick solution is to `add gem "haml-rails", "~> 0.3.4"` to our Gemfile and `bundle install`. Now, rerun the `rails g controller Home index` and voila!, we have a Haml template. (Oh, and go ahead and remove the **app/views/home/index.html.erb** file.)

Next, we need to point our root path to the index method on our new controller. Open up **config/routes.rb**, remove the `get "home#index"` line and replace it with `root :to => "home#index"`. Running the spec again brings the return of the no "Sign In" link error.

In the interests of time and effort, I'm going to jump ahead and create the layout for our home page. I'm using Skeleton,[5] and have modified the **app/views/layouts/application.html.haml** (I migrated the ERB layout file to Haml) and the **app/views/home/index.html.haml** files accordingly. Here is what you need to do to catch up:

- Download and uncompress Skeleton.
- Copy the files from the JavaScript and stylesheets directories to those same directories in `app/assets/`.
- Replace the contents of your **app/views/layouts/application.html.haml** file with the content here.
- Replace the contents of your **app/views/home/index.html.haml** file with the content here.
- Replace the contents of your **app/assets/stylesheets/home.css.sass** file with the content here.
- Copy Figure 5.4 to your app/assets/images directory.

[5] http://getskeleton.com/

Figure 5.4. Loccasions header 150 x 150 pixels

If you rerun the spec, it passes as in Figure 5.5.

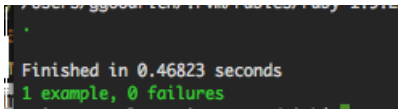

Figure 5.5. Pass me on the left-hand side

Just like that, our first user story is complete. It's Miller Time!

Feel free to look over the layout and index Haml files and see how I met our spec. Once you are comfortable with Haml's syntax, the details of our layout is much less interesting. Don't forget:

```
git add .
git commit -am "Added home page and layout"
git push origin master
```

In the next chapter, we'll start with another user story and see where that takes Loccasions.

By the way, the font used for the Loccasions header is Eight One by glue.[6]

[6] http://glue.deviantart.com/art/Eight-One-45198536

Chapter 6

Authentication

In the previous chapter, we finished our first user story. That user story was fairly simple, but it flushed out the design of our home page. The next user story, "As an administrator, I want to invite users to Loccasions" is not quite so simple.

The implications from this user story are big: First, we have a new role, administrator, which brings our roles to two (unregistered user and administrator). Second, the administrator role brings out the idea of *authorization*, where functionality of the site is restricted based on the user's role. Of course, this points us to *authentication*, because we need to know who the user is before we can figure out what the user can do. By the end of this user story, we should have authentication and authorization set up and ready to go.

Create a Branch

Something I neglected in the last chapter was to create a Git branch for our new story. This keeps our code isolated to the branch and allows us to make unrelated changes to master if needed. Here is a good article on Git workflow.[1]

[1] http://thinkvitamin.com/code/source-control/git/our-simple-git-workflow/

Git makes branching easy:

```
git checkout -b inviting_users
```

… and we're in our new branch, ready to go.

Write the Test

Continuing with our test-driven approach, let's write a test for signing in to the application. Here, we will start to break our user story into smaller stories to facilitate testing. Breaking a problem or piece of functionality down into smaller parts makes the bigger problem easier to tackle. Here's our first sub user story: *As an administrator, I want to sign in to Loccasions*.

Since we are writing specs from the user's perspective, each story (and sub story) has implications. In this story, the act of "signing in" can mean many things, so how do we measure it to match what we/our client wants? In this case, I conferred with the client who wants a cool drop down to come down (a la Twitter) with a sign-in form. While I agreed that is cool, I talked the client into progressive enhancement,[2] allowing us to develop a separate page for the sign-in form, for now, and return to make it sexier later.

With our client expectations in hand, we can make our sign-in form test. At first, we'll just make sure the sign-in page has a form and a title:

```
require 'spec_helper'
feature 'Sign In', %q {
  As an administrator
  I want to sign in to Loccasions
} do
  background do
    visit "/"
  end
  scenario "Click Sign In" do
    click_link "Sign In"
    page.should have_selector("title",
                              :text => "Loccasions: Sign In")
```

[2] http://www.alistapart.com/articles/understandingprogressiveenhancement

```
    page.should have_selector('form')
  end
end
```

(Remember to fire up mongodb before running your specs.) This spec fails, complaining about the title not matching. Also, I noticed I was getting the following message when I ran my specs:

```
NOTE: Gem.available? is deprecated, use
 Specification::find_by_name. It will be removed on or after
 2011-11-01.
Gem.available? called from /Users/ggoodrich/.rvm/gems/ruby-1.9.2-
➥p290@loccasions/gems/jasmine-1.0.2.1/lib/jasmine/base.rb:64.
```

HMMMM … I don't like that. A quick trip to the `jasmine-gem` GitHub repo[3] shows they are on version 1.1.0.rc3. For now, I will bump the version in my Gemfile and hope it works when we get to client-side testing. Bumping the version fixes that warning, so immediate needs met.

Here, I rely on experience to drive my next step. I am keen on using Devise for authentication, and I know it has it's own views for signup, signin, etc. In other words, it's time to set up Devise.

Set up Devise

Tipping my hat to the awesome RailsApps[4] yet again, let's prepare RSpec for Devise. Create a **spec/support/devise.rb** file with:

```
RSpec.configure do |config|
  config.include Devise::TestHelpers, :type => :controller
end
```

Now, on to the typical Devise setup:

```
rails g devise:install
```

[3] https://github.com/pivotal/jasmine-gem
[4] https://github.com/RailsApps/rails3-mongoid-devise/wiki/Tutorial

This creates **config/initializers/devise.rb** and **config/locales/devise.en.yml**. If we look in the initializer file, we can see `require 'devise/orm/mongoid'`, so we know that Devise is aware of our choice to use Mongoid. The output of the `devise:install` generator gives some instructions:

- Set up `default_url_options` for ActionMailer
- Set up a default route (we have done this already)
- Make sure we handle our flash/notice messages in our layout.

Let's do this stuff while it's fresh. I added:

```
config.action_mailer.default_url_options = {
  :host => 'localhost:3000'
}
```

… to **config/environments/development.rb**. Also, I added:

```
%p.notice = notice
%p.alert = alert
```

… to **app/views/layouts/application.html.haml** just above the call to `yield`. It may not stay there, but we aren't worried about that right now.

Devise will generate a User model for us:

```
rails g devise User
```

The output of this command shows that we get a model (User), a user_spec, and a new route (`devise_for :users`). If we do a quick `rake routes` at the command line, we see:

```
new_user_session GET    /users/sign_in(.:format)
{:action=>"new", :controller=>"devise/sessions"}
```

… which is where we want our "Sign In" link to go. Let's change it in the application layout:

```
#sign_in.sixteen.columns
  %a(href= new_user_session_path ) Sign In
```

Rerunning our spec, and we still have the same error. At this point, let's fire up the server and see what is happening.

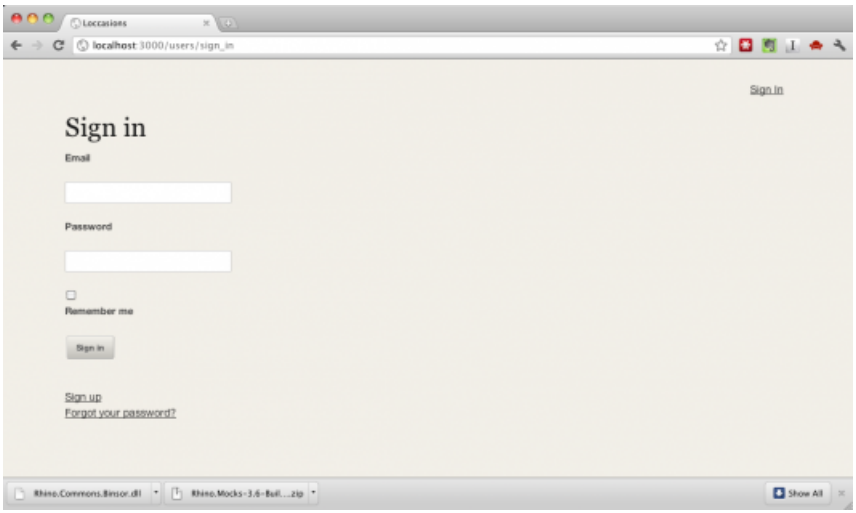

Figure 6.1. Sign Up page

Wow … that looks pretty good. However, the title isn't what we want, and it has a 'Sign Up' link, which we may want later, but not yet. We need to customize the Devise views and, thankfully, Devise gives us an easy way to do just that:

```
rails g devise:views
```

```
      invoke  Devise::Generators::SharedViewsGenerator
      create    app/views/devise/mailer
      create    app/views/devise/mailer/confirmation_instructions.html.erb
      create    app/views/devise/mailer/reset_password_instructions.html.erb
      create    app/views/devise/mailer/unlock_instructions.html.erb
      create    app/views/devise/shared
      create    app/views/devise/shared/_links.erb
      invoke  form_for
      create    app/views/devise/confirmations
      create    app/views/devise/confirmations/new.html.erb
      create    app/views/devise/passwords
      create    app/views/devise/passwords/edit.html.erb
      create    app/views/devise/passwords/new.html.erb
      create    app/views/devise/registrations
      create    app/views/devise/registrations/edit.html.erb
      create    app/views/devise/registrations/new.html.erb
      create    app/views/devise/sessions
      create    app/views/devise/sessions/new.html.erb
      create    app/views/devise/unlocks
      create    app/views/devise/unlocks/new.html.erb
```

Figure 6.2. Output of Devise Views

That creates quite a few views, and they are all ERB not Haml … UGH. Googling around, I found this on the Devise wiki[5] detailing how to get Haml views for Devise. Check it out.

First, let's change the title. Since the Devise views will use the same application layout, we need a way to change the title for each page. Enter the `ApplicationHelper`. Add this method to **app/helpers/application_helper.rb**:

```
def title(page_title)
  content_for(:title) { page_title }
end
```

Now, replace the title tag in the application layout with:

```
%title= "Loccasions: #{content_for?(:title) ? content_for(:title) :
 ➥'Home' }"
```

Finally, add this to the top of **app/views/devise/session/new.html.haml**:

```
- title('Sign In')
```

[5] https://github.com/plataformatec/devise/wiki/How-To:-Create-Haml-and-Slim-Views

Now, the specs all pass.

Decision Point: User Names

After some deliberation with the client, we are going to add a `name` attribute to the users. Let's add a test for our new attribute. Devise was kind enough to create a user_spec, so let's create a test for `name` (in **spec/models/user_spec.rb**):

```
describe User do
  it "should have a 'name' attribute" do
    user = User.new
    user.should respond_to(:name)
    user.should respond_to(:name=)
  end
end
```

That spec fails, as expected. We can make it pass by adding this to **app/models/user.rb**:

```
field :name
attr_accessible :name
```

I want name to be unique and required. Tests (with a bit of refactoring):

```
describe User do
  describe "the 'Name' attribute" do
    before(:each) do
      @user = Factory.build(:user)
    end
    it "should exist on the User model" do
      @user.should respond_to(:name)
      @user.should respond_to(:name=)
    end
    it "should be unique" do
      @user.save
      user2 = Factory.build(:user, :email=>'diff@example.com')
      user2.valid?.should be_false
      user2.errors[:name].should include("is already taken")
    end
    it "should be required" do
      @user.name=nil
      @user.valid?.should be_false
```

```
      @user.errors[:name].should include("can't be blank")
    end
  end
end
```

The Alert Reader has notice the calls to `Factory` in the refactored spec. We need a user for this test, and we'll turn to Factory Girl to get one. Add the file **spec/factories.rb** with:

```
require 'factory_girl'
FactoryGirl.define do
  factory :user do
    name 'Testy'
    email 'testy@test.com'
    password 'password'
  end
end
```

Running the spec gives us 2 failures:

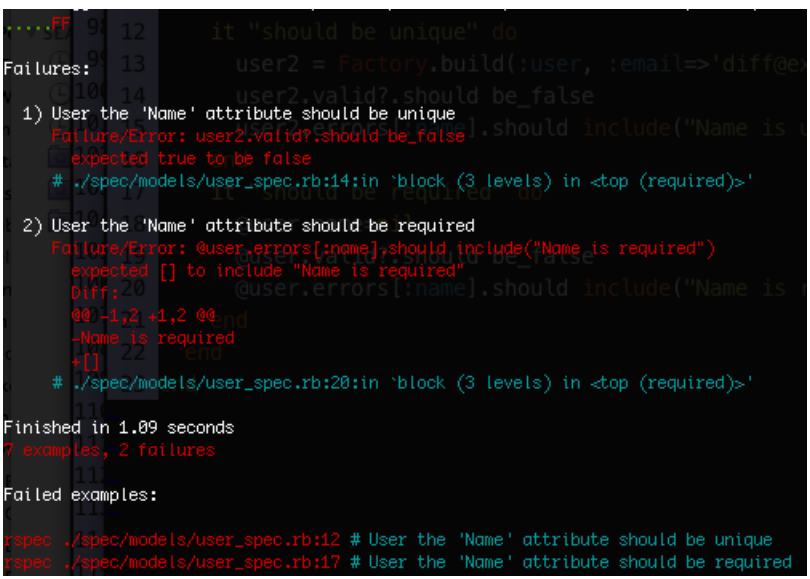

Figure 6.3. User Name Spec Fails

Add some quick validation to our `name` field:

```
validates :name, :presence => true, :uniqueness => true
```

… and all our specs pass. We can now move on to actually testing sign in.

Test Sign In

The first item to determine for our sign-in test is, what happens when a user successfully signs in?

The customer thinks that the user should be redirected to their individual "home" page. What is on the user home page, then? We know our main business objects are Event and Occasion, and that Occasions live inside Events. The user home page, then, should probably list the user's events, to start. The spec, then, should fill out and submit the form, then redirect to the user home page.

Before we write this spec, I want to make the spec task the default Rake task (I am tired of typing `rake spec`) so add this to the bottom of your *Rakefile*:

```
Rake::Task[:default].prerequisites.clear
task :default => [:spec]
```

Now, we can just type `rake` and our specs will run. AAAAAH, that's better.

Here is our sign-in spec:

```
scenario "Successful Sign In" do
  click_sign_in
  fill_in 'Email', :with => 'testy@test.com'
  fill_in 'Password', :with => 'password'
  click_on('Sign in')
  current_path.should == user_root_path # used by Devise
end
```

Notice the `click_sign_in` method? I made a quick helper (**spec/support/request_helpers.rb**) so I didn't need to keep typing the lines to click get to the sign in page:

```
module RequestHelpers
  module Helpers
    def click_sign_in
      visit "/"
      click_link "Sign In"
    end
  end
end
RSpec.configure.include (RequestHelpers::Helpers,
                         :type => :acceptance,
                         :example_group => {
  :file_path => config.escaped_path(%w[spec acceptance])
})
```

This will only include our helper in the acceptance tests, meaning, any specs in *spec/acceptance*. (Note: RSpec defines a bunch of spec "types", such as *request, controller, models*, etc. Here, we are just adding *acceptance*.)

Running `rake` (Yay! Isn't that better?) and we get an expected error about `user_root_path` being undefined. Just to get the test passing, add this to **config/routes.rb**:

```
match 'events' => 'home#index', :as => :user_root
```

We'll call the route `/events`, since we know events will be the main course of the user home page. The spec now fails because the URLs don't match. After we submit the form, the URL is unchanged. This is because we have no users in the database. Add this to the "Successful Sign In" scenario, just after `click_sign_in`:

```
FactoryGirl.create(:user)
```

Yay! The spec now passes. The `user_root` route, by the way, is a Devise convention to override where the user is redirected after successful sign in. We haven't fully tested authentication, but it's working. For completeness sake, let's make sure a bad login fails. Add this under the "Successful Sign In" scenario:

```
scenario "Unsuccessful Sign In" do
  click_sign_in
  fill_in 'Name', :with => "BadUser"
```

```
  fill_in 'Email', :with => 'hacker@getyou.com'
  fill_in 'Password', :with => 'badpassword'
  click_on 'Sign in'
  current_path.should == user_session_path
  page.should have_content("Invalid email or password")
end
```

Yup, the new scenario passes, as expected. Let's go ahead and push this to GitHub:

```
git add .
git commit -m "Basic authentication"
git checkout master
git merge inviting_users
```

Run your specs here, just to make sure everything is okay, then:

```
git push origin master
git branch -d inviting_users
```

This is becoming a bit too long, so we'll stop here and pick up with our user Events page in the next chapter.

Chapter 7

Spork, Events, and Authorization

Previously, we ended with very basic authentication working. However, we're faking out the `events_path` in our `sign_in` spec, which is where we'll start. A successful sign-in redirects to the user events page which, presumably, has a list of the events owned by that user. Let's go back to Mockbuilder and crank out a layout for our events page.

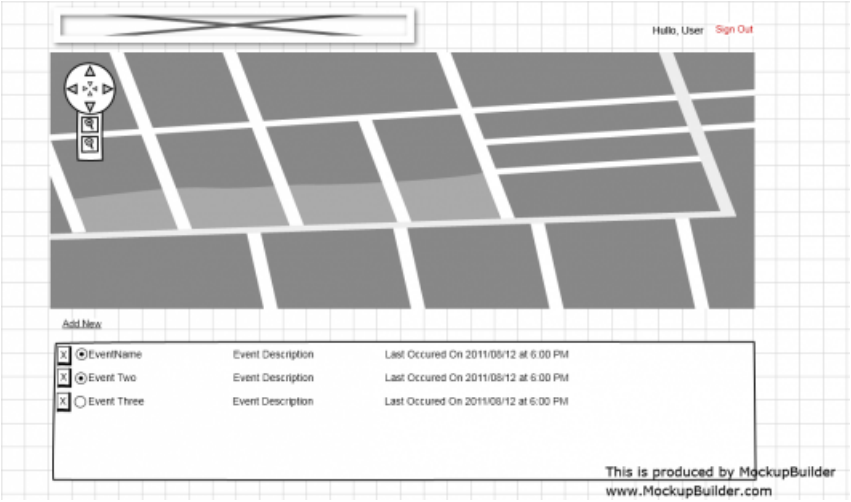

Figure 7.1. Events Page Mockup

The page has a different layout than the home page, with the main content being a map. The latest occasions will be visible on the map, and the users' Events will be listed below the map. It's another very simple layout, helping us drive the implementation of the site. At this point, we can identify Events as a *resource* and start developing the items that will represent this resource. Of course, we'll need an Event model and an Event controller.

Event Model

At this stage, it might be a good idea to solidify what the client wants to track on each Event. This calls for the addition of some more user stories:

> "As a registered user, I would like to see my Events by Name, Description, and last occurrence."

That story lists our attributes explicitly, so we have enough to generate our model. First, let's make a Git branch:

```
git checkout -b adding_events
```

Before we generate the model, a quick thought on the "last occurrence" attribute. Knowing that Events will have Occurrences, it seems to me that we won't actually

store a value for last_occurrence, but will grab the latest date from the Occurrences. This makes last occurrence a "virtual attribute", as it's generated when you ask for it. However, part of me worries about the need to query on this attribute. Another design decision: Create a Virtual Attribute or a Real Attribute that has to be updated whenever an occurrence is created. Well, premature optimization is the root of all evil. We'll go with a virtual attribute … for now.

```
rails g model Event name:string description:string user:references
```

… which generates a model and spec. We have a couple of tests to write for Event. Events belong to users (thus, the `user:references` in our generator) and can't exist without a Name. I went ahead and added a Factory for events:

```
factory :event do
name "Test Event"
user
end
```

Here is the test for user:

```
describe Event do
  it "should belong to a user" do
    event = Factory.build(:event, :user=>nil)
    event.valid?.should be_false
    event.errors[:user].should include("can't be blank")
    event.user = User.new
    event.valid?.should be_true
  end
end
```

… which fails, until we add this to our Event model:

```
validates :user, :presence => true
```

Before we continue with the Event specs, I am getting frustrated with how long it is taking each time I run my specs. The spin-up time is too long, so I want to do something to speed this up. My instincts tell me that the specs have to load the Rails application environment, and that is what is taking the most time. I tried just

running the model specs (`rake spec:models`), but the startup time is still bothering me. After much googling, I am going to try adding Spork[1] to our test environment.

Adding Spork

Add `gem 'spork-rails'` to the `development` portion of your Gemfile and `bundle install`. Now, we need to setup the spec environment for Spork. Thankfully, Spork has some helpers for this. From your project root, type `spork --bootstrap`, which will add some instructions into your **spec/spec_helper.rb** file, so open that file. Basically, we need to split our `spec_helper` into two blocks, `prefork` and `each_run`. Anything we only need to run once for each spec run goes in the `prefork` block. For now, I am putting everything in the `prefork` block, so my **spec/spec_helper.rb** file looks like:

```
require 'rubygems'
require 'spork'
Spork.prefork do
  ⋮
  ENV["RAILS_ENV"] ||= 'test'
  require File.expand_path("../../config/environment", __FILE__)
  require 'rspec/rails'
  require 'capybara/rspec'
  ⋮
  Dir[Rails.root.join("spec/support/**/*.rb")].each {|f| require f}
  RSpec.configure do |config|
    ⋮
    config.mock_with :rspec
    ⋮
    require 'database_cleaner'
    config.before(:suite) do
      DatabaseCleaner.strategy = :truncation
      DatabaseCleaner.orm = "mongoid"
    end
    config.before(:each) do
      DatabaseCleaner.clean
    end
  end
end
```

[1] http://github.com/timcharper/spork

```
Spork.each_run do
  # This code will be run each time you run your specs.
end
```

We may run into issues having to restart Spork to pick up changes, but we'll deal with that if it happens. Running a quick time check on before and after (`time rake vs time rspec --drb spec/`), my spec run time dropped by 20 SECONDS! Wow…that is a worthy of a nerdgasm. To finish our Spork changes, add `--drb` to your **.rspec** file so that it uses Spork by default. By the way, since Spork loads up the Rails environment, it will be necessary to restart Spork when that environment changes (new route added, etc.) Something to bear in mind.

Back to Testing

Okay, now we can finish our Event specs. Let's add a quick test making sure 'name' is a required attribute.

```
it "should require a name" do
  event = Factory.build(:event, :name=>nil)
  event.valid?.should be_false
  event.errors[:name].should include("can't be blank")
  event.name ="Event Name"
  event.valid?.should be_true
end
```

This fails, because we are not validating the name. Change the line we added to validate the user to:

```
validates :user, :name, :presence => true
```

… and the spec passes.

Testing That a User Has Events

Since we want to be able to build events for Users, let's put in some tests to make sure that works:

```
describe "User Event" do
  it "can be built for a user"  do
    lambda {
     @user.events.build(:name=>"A new event")
    }.should change(@user.events, :length).by(1)
  end
  it "can be removed from a user" do
    @user.events.build(:name => "A short event")
    lambda {
      @user.events.first.destroy
    }.should change(@user.events, :length).by(-1)  end
end
```

These specs pass without further coding, so, um, yay?

Wait!

This is not true … Alert Reader Nicolas rightly points out that we still have two things to do:

1. Add `embeds_many :events` to the User model.
2. Either restart spork or add `ActiveSupport::Dependencies.clear` to our `Spork.each` section in **spec_helper.rb**.

Events Controller

Now we need to display our events on the user events page. We'll need an EventsController and an index action/view. Generators, ho!

```
rails g controller events index
```

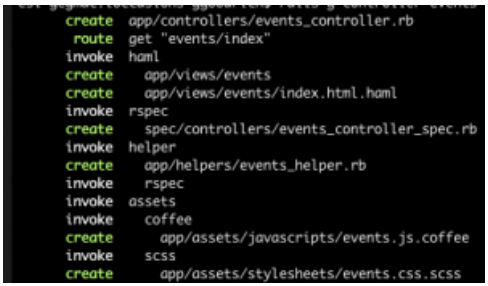

Figure 7.2. EventsController and index action/view

Let's undo our route cheat from the last chapter, and point the events_path to our events#index action. Delete `get "events#index"` and change the 'events' route to `match 'events' => 'events#index', :as => :events` and run the specs. HMMM … they fail.

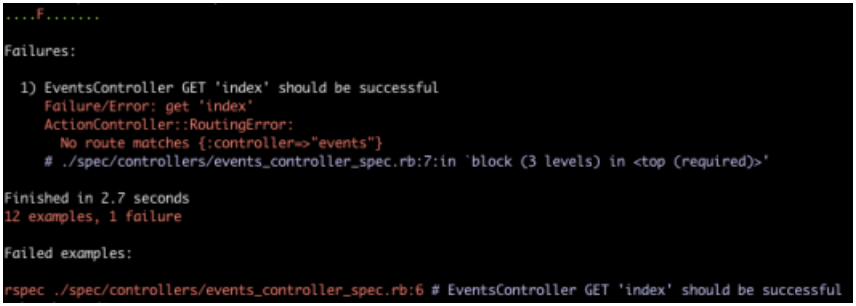

Figure 7.3. Events spec fails!

I wasn't expecting that to fail. Wait a sec, I didn't write that `events_controller_spec`. Stupid generators … I don't want that spec. Delete the `spec/controllers` directory with extreme prejudice. There, our specs are passing again. As I mentioned in the setup, we are, as much as possible, driving the testing through acceptance tests. Because of that, we won't have specific controller/view specs.

Create the file **specs/acceptance/user_events_spec.rb** to hold the specs for our user events page. In order to see the user events page, we'll have to sign in from our spec. Since we are using request specs, we need to, basically, simulate the posting of credentials to the server. We can do that with a quick mixin, which we'll include in our feature. Add this to **spec/support/request_helpers.rb**:

```
def login_user(user)
  visit new_user_session_path
  fill_in "Email", :with => "testy@test.com"
  fill_in "Password", :with => "password"
  click_button "Sign in"
end
```

As you can see, we are literally signing into the application.

Now, let's create the **spec/acceptance/user_events_spec.rb** file with:

```
require 'spec_helper'
feature 'User Events Page', %q{
  As a signed in user
  I want to see my events
  on my events page
} do
  background do
    @user = Factory(:user)
    @event = Factory(:event, :user => @user)
  end
  scenario "User is not signed in" do
    visit events_path
    current_path.should == new_user_session_path
  end
end
```

We are going to test the negative path first, meaning, what happens when we don't sign in and try to go to the user events page. In this case, the app should redirect to the Sign In page (which is the `new_user_session_path`) Run this spec, and the path points to a (rather ugly, make a note to fix that) events URL, so it's not redirecting. We need to tell the *EventsController* to authenticate the user. Thanks to Devise, all we have to do is add this (put it right under the `class` statement):

```
before_filter :authenticate_user!
```

... and the spec passes. Now, let's create the positive path:

```
feature 'Signed In User Events Page', %q{
  As a signed in user
  I want to see my events
  on my events page
} do
  background do
    @user = Factory(:user)
    @event = Factory(:event, :user => @user)
    login_user(@user)
  end
  scenario "User is signed in" do
    visit events_path
    page.should have_content(@user.name)
    page.should have_content(@event.name)
  end
end
```

We are measuring success here by simply making sure the user name and event name are on the page. It's likely that we'll have to strengthen this test later.

Run the spec, and it will complain that the user's name was not found on the page. Let's open up the **app/views/events/index.html.haml** file and see what's up. That view is still the basic, generated view, so we need to match it to our mockup as much as possible. First off, the "sign in area" in the mockup has a greeting and the user name. That bit is in the **app/views/layouts/application.html.haml** file. I changed the `#sign_in` div to:

```
#sign_in.sixteen.columns
  %span
    -if user_signed_in?
      Hullo #{current_user.name}
      |
      = link_to "Sign Out", destroy_user_session_path, :method =>
➥:delete
    - else
      = link_to "Sign In", new_user_session_path
```

Run the spec, and now it complains about the event name. Progress. Here, we'll open up the **app/views/events/index.html.haml** view and change it to

```
%ul#events
  - for event in @events
    %li= event.name
```

Which leads to the spec complaining about `You have a nil object when you didn't expect it!` (I hate that error, it has caused my hours of frustration) because we are looping over an `@events` object that does not exist. To the controller!

```
class EventsController < ApplicationController
  before_filter :authenticate_user!
  def index
    @events = current_user.events
  end
end
```

We're back to passing specs.

Wrap Up

Although we're still just coming out of the gate with Loccasions, we're picking up speed. The next chapter will, I hope, allow us to flush out the rest of the Events page, and enable us to create and modify events.

Chapter 8

Making Events

The last chapter flushed out the Events model and created a *very* basic home page. By the end of this chapter, we'll be able to add, modify, and delete events from our user home page.

CRUDdy Events

Unless you've just been unfrozen from a decades long, icy slumber, you know what CRUDifying a model entails. In Rails land, more often than not, a model is transformed into a REST[1]ful resource. (Note: If you don't have a great grasp on REST, I would HIGHLY recommend the O'Reilly book RESTful Web Services,[2] I won't be covering REST in detail in this book.) For the Event model, we'll need HTTP endpoints that allow us to add (POST), modify (PUT), retrieve (GET), and delete (Um, DELETE) events.

[1] http://en.wikipedia.org/wiki/Representational_state_transfer
[2] http://oreilly.com/catalog/9780596529260/

Creating Events

Since an Event is such a simple object, we can place the form for creating them right on our user home page. With the agility of a cat, I jump back into MockupBuilder and change our events page to add a form at the bottom.

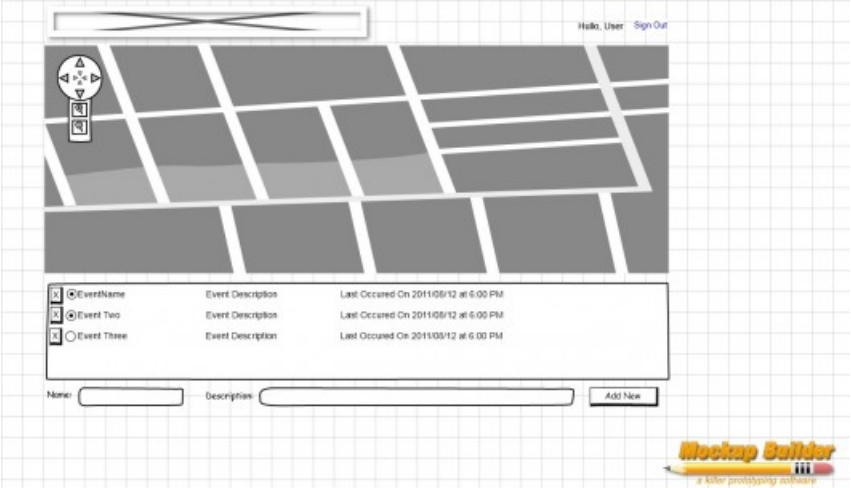

Figure 8.1. Add Event

Time to write a test to fill in that form. I have added a **spec/acceptance/add_events_spec.rb** with:

```
require 'spec_helper'
feature 'Add Events', %q{
  As a registered user
  I want to add Events
} do
  background do
    login_user Factory(:user)
  end
  scenario "Add Basic Event" do
    fill_in "Name", :with => "New Event"
    fill_in "Description", :with => "This is my new event"
    click_button "Create Event"
    page.should have_content("New Event")
    page.should have_content("This is my new event")
```

```
    page.should have_selector("ul > li")
  end
end
```

Running this spec results in complaints about ElementNotFound because we haven't created our form yet. So, let's add a form to our **app/views/events/index.html.haml** (add to the end of the file):

```
= form_for Event.new do |f|
  = f.label :name
  = f.text_field :name
  = f.label :description
  = f.text_field :description
  = f.submit
```

Since we want this form "inline" we need to override the base styles. In the **app/assets/stylesheets/events.css.scss** file add:

```
.new_event label, input[type='text'] {
  display: inline;
}
input#event_description {
  width: 500px;
}
```

(Note: Rails adds the `new_event` class to the form). Now, the spec complains about there being no `create` action on `EventsController`. Okey dokey, we can add that (in **app/controlles/events_controller.rb**):

```
def create
  event = current_user.events.build(params[:event])
  event.save
end
```

Now the spec says we don't have a `events/create` template, which is true. But, we don't want that template, we want the `create` action to just render the home page. So, for now, let's redirect to the events index page. I added:

```
page.current_path.should == events_path
```

… to our spec and:

```
redirect_to events_path
```

… to the `EventsController#create` method. The spec complains about the description text not being on the page. Oops, looks like I neglected that when I created the list of events. Change the **app/views/events/index.html.haml** to:

```
%h2 Your Events
map.sixteen_columns
%ul#events
  - for event in @events
    %li
      %span.event_name= event.name
      %span.event_description= event.description
= form_for Event.new do |f|
  = f.label :name
  = f.text_field :name
  = f.label :description
  = f.text_field :description
  = f.submit
```

… and the spec passes.

At points like this, I like to fire up the server (`rails s`) and look around. The first thing I notice is, when I am signed in and on the home page, there is no clear navigation to the user events page. Also, when I am on the *user events* page, the list of events looks like crap. I am not too hung up on the latter issue, but I'd like to get some visual clues in there to make them a bit nicer for now. Finally, the space where our map is going to go is a big void, but I am not going to deal with that until we get to creating Occasions.

Clean up the Signed In Navigation

Dealing with the first issue from above, when a user is signed in there should be a link to get to the user's events. I am going to call it "My Events". First, though, we need to write a test to make sure it's there. In fact, we can just add it to the "Successful Sign In" scenario in **spec/acceptance/sign_in_spec.rb**:

```
current_path.should == user_root_path # This line already exists
page.should have_selector("a", :text => "My Events",
                               :href => user_root_path)
```

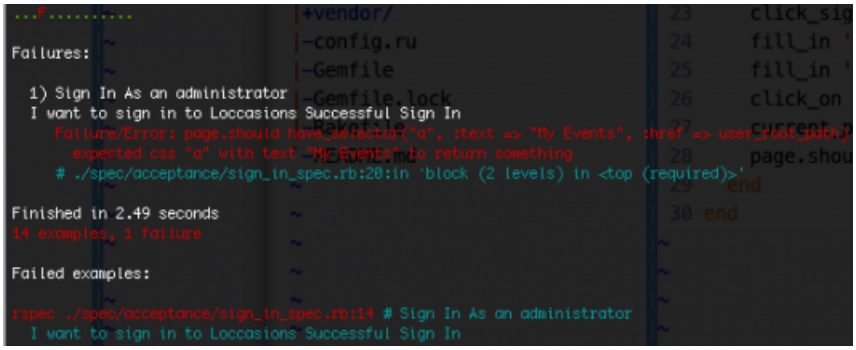

Figure 8.2. My Events Link Spec

Making this spec pass is simply a matter of adding this "My Events" link to the sign_in conditional in the **app/views/layout/application.html.haml** file. (Note: just add the lines commented with `#Add this line`):

```
-if user_signed_in?
      Hullo #{current_user.name}
      | <br>
      = link_to "My Events", user_root_path   #Add this line
      |                                       #Add this line
      = link_to "Sign Out", destroy_user_session_path, :method =>
➥:delete
    - else
      = link_to "Sign In", new_user_session_path
```

… and the spec passes. If you want to fire up the server and look at our new sign-in area, go for it.

Adding More CRUD to Events

We can create and retrieve Events, but we can't edit or delete events. Let's add the ability to do that now. My initial thoughts with editing are to just load the "selected" event into the same form we are using to create events. I doubt it will stay this way, but it gives us a quick way to test the update capabilities. Using this workflow, the

`edit` action of our `EventsController` will fetch the selected Event from the database and hydrate a `@event` instance variable. The *edit view* will then render that Event information into the form. A significant takeaway, then, is that our user events page is also going to be our edit view. Also, the user will need a way to "select" a particular event. For now, we'll make the Event name in the list of events a hyperlink that fires the event view. Let's write some more tests to flush this out. I have created a **spec/acceptance/edit_events_spec.rb** file with:

```
require 'spec_helper'
feature 'Select Event', %q {
  As a registered user
  I want to select an Event
} do
  background do
    @user = Factory(:user)
    @event = Factory(:event, :user => @user )
    login_user @user
  end
scenario "Select Event" do
    page.should have_selector("a", :text=> @event.name)
    click_link @event.name
    page.should have_selector("li.selected", :text=> @event.name)
    page.should have_selector("input[name='event[name]']",
                              :value => @event.name)
    page.should have_selector("input[name='event[description]']",
                              :value => @event.description)
  end
end
```

Of course, the spec fails because there is no link with "Test Event" (remember, that's our factory Event object name) on the page. Open **app/views/events/index.html.haml** and change:

```
%span.event_name= event.name
```

… to:

```
%span.event_name
  = link_to event.name, edit_event_path(event)
```

The spec complains `The action 'edit' could not be found for EventsController`, which makes sense. So, add an `edit` method to the EventsController:

```
def edit
end
```

And now, the spec complains about the `edit` template missing. As a quick tangent, Rails tells you where it looked, and you can see that Devise has modified our views search path … pretty cool.

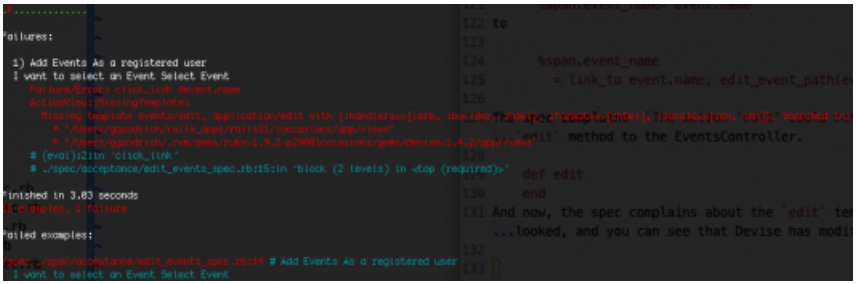

Figure 8.3. Where's the `edit_template`?

As I previously mentioned, we aren't going to have a separate edit template, but rather, we are going to use our existing events index template and just load the selected event into an instance variable:

```
def edit
  @events = current_user.events
  @event = @events.find(params[:id])
  render 'index'
end
```

The next issue is that there is no `li` with a class `selected`, so open up the events index template and change it to:

```
%h2 Your Events
map.sixteen_columns
%ul#events
  - for event in @events
    %li{:class => @event == event ? :selected : nil}  #FIRST
```

```
      %span.event_name
        = link_to event.name, edit_event_path(event)
      %span.event_description= event.description
= form_for @event || Event.new do |f|          #SECOND
  = f.label :name
  = f.text_field :name
  = f.label :description
  = f.text_field :description
  = f.submit
```

In the interest of time, I've made all the changes to make our spec pass. First, the events loop checks to see if the current event matches our `@event` instance variable and adds the `selected` CSS class name to the list item when it does. Second, we have the `form_for` test for the existence of the `@event` instance variable, and fall back to a plain `Event.new` if it's not there.

The spec now passes. We can select an event, which loads it into the form. Test HO! (By "HO!" I mean, add this feature to the same **edit_event_spec.rb** file):

```
feature 'Edit Event', %q {
  As a registered user
  I want to edit a selected Event
} do
  background do
    @user = Factory(:user)
    @event = Factory(:event, :user => @user )
    login_user @user
    click_link @event.name
  end
  scenario "Edit Event" do
    fill_in "Name", :with=> "Edited Event"
    click_button "Update Event"
    page.should have_selector("a", :text => "Edited Event")
  end
end
```

You can see that we select our event in the `background` block (sniff, sniff, I smell helper …), followed by the scenario of changing the Event's name. This spec fails because there is no `update` action on `EventsController`. Just like the `edit` action, we need to add our new action to the controller. However, before we do that, I want to point out something cool that Rails just did for us, free of charge. Notice in the

"Edit Event" scenario, we look for an "Update Event" button. However, we haven't put any code in the view to differentiate between a create form and an update form. Rails and `form_for` do this for us, making the form do the right thing based on the object passed into it. Some of the little things Rails does, like this, makes me want to give it a great big hug.

Adding the `update` action follows the same steps as adding the `edit` action. Add the empty `update` method to `EventsController`, watch the spec complain about the missing update template, then add the guts to the `update` method, redirect to the *index view*, and rock out. The redirect is a slight difference, because after an update we just want to go back to the events page to refresh our updated event in the events list:

```
def update
  event = current_user.events.find(params[:id])
  event.update_attributes(params[:event])
  event.save
  redirect_to events_path
end
```

Specs pass and we can create and update Events. Progress is fun.

MUST DESTROY EVENTS

Once Loccasions can destroy events, we'll be done with the manipulation of events. First, we'll need something for the user to indicate that an event is a goner. A "Delete" button sounds like a good start, as does writing a test for said button. Here's the spec:

```
require 'spec_helper'
feature "Delete Event", %q{
  As a registered user,
  I want to delete an event
} do
  background do
    Capybara.current_driver = :selenium   #FIRST
    @user = Factory(:user)
    @event = Factory(:event, :user => @user,
                     :name => "Dead Event Walking")
    login_user @user
```

```
  end
  after do                        #afterFIRST
    Capybara.use_default_driver
  end
  scenario "Delete Event" do
    page.should have_content("Dead Event Walking")
    page.should have_selector("form[action='/events/#{@event.id}']
input[value='delete']") #SECOND
    # auto confirm the dialog
    page.execute_script(
      'window.confirm = function() {return true;}'
    )
    click_button "X"
    page.should_not have_content("Dead Event Walking")
  end
end
```

I've done that thing where I jump ahead a bit with the "Delete Event" spec, so I'll try to explain what is happening. FIRST, I am switching the test_driver for Capybara to http://seleniumhq.org/, telling the page to just auto confirm all dialogs, and then switching back to the default test driver. SECOND, you might be wondering why I am testing for the existing of a form with those strange attributes. Due to Rails RESTful nature, the destroy route for a resource requires the use of the HTTP DELETE method. The only way to perform an HTTP request that does not use GET is to use a form. However, support for the HTTP DELETE method amongst browsers is spotty and http://www.w3.org/Bugs/Public/show_bug.cgi?id=10671, so we need a convention. A current convention, and what Rails will do for you, is to create a POST form including a hidden input called `_method` that has the value of the HTTP verb we want to use. The Rails routing middleware then checks for that parameter and routes the request appropriately. That's why I wrote this spec that way, even if it goes a bit deeper than a normal accepetance test might. Again, this test will likely change down the road. The spec, of course, will complain about the view not having a form with those attributes. Here's our new *index view*:

```
%h2 Your Events
map.sixteen_columns
%ul#events
  - for event in @events
    %li{:class => @event == event ? :selected : nil}
      %span.del_form
```

```
        =button_to "X", event, :confirm => "Are you sure?", :method
➥=> :delete
      %span.event_name
        = link_to event.name, edit_event_path(event)
      %span.event_description= event.description
      %div.clear
= form_for @event || Event.new do |f|
  = f.label :name
  = f.text_field :name
  = f.label :description
  = f.text_field :description
  = f.submit
```

If you ran `rails s` now, signed in, and went to the user events page, you could see (provided an Event exists) the delete button. Viewing the source of that page shows the delete form:

```
<span class='del_form'>
  <form method="post" action="/events/4e67812841574e0462000002"
                      class="button_to">
    <div>
      <input name="_method" type="hidden" value="delete" />
      <input data-confirm="Are you sure?" type="submit" value="X" />
      <input name="authenticity_token" type="hidden"
             value="..elided.." />
    </div>
  </form>
</span>
```

Rails is giving us all kinds of help here: the hidden `_method` input, the `data-confirm` attribute for our confirmation box, and an `authenticity_token` to help avoid cross-site scripting attacks. And what did YOU get Rails? Nothing, eh?

Run the spec, and we get the familiar complaint about `EventsController` missing an action, `destroy` in this case. At this point, you should know what's coming. Add the blank method, watch it fail, add the destroy and redirect logic, watch it pass, and, finally, feel good about yourself. Once you've added the `destroy` method:

```
def destroy
  event = current_user.events.find(params[:id])
  event.destroy
  redirect_to events_path
end
```

… all specs will pass. You may have been a bit startled by the browser popping up when you ran the specs, eh? That's Selenium, and it's awesome. (But, we'll probably get rid of it later …) So, if you fire up the server, you should be able to add, modify, and destroy events. Next time, we'll add Occasions and, maybe (DUN DUN DU-UUUUN) the map. Oh, and don't forget:

```
git add .
git commit -am "CRUDed events"
git push origin adding_events
git checkout master
git merge adding_events
git push origin master
```

Chapter 9

Pair Programming

On a day like any other, I was scanning through my Twitter feed searching for knowledge and inspiration. Although I didn't know it yet, this day of tweet fishing would prove fruitful and change the direction of Loccasions forever. (You should imagine some dramatic music here, complete with a cutaway of a Loccasions screen shot fading to black.)

The tweet that would change everything was about rubypair.com.[1] RubyPair.com is a "searchable directory of developers who are looking to pair, in-person or remotely, on topics that they've expressed an interest in through their profile." In other words, it's a way to find someone to pair program[2] on your (or their or some open-source) Ruby project. I read this tweet, and it struck me like a thunderbolt! I need to pair program on Loccasions! I mean, I had one (arguable) programmer, I just needed to find another. Where would I find such a soul? Well, I thought that the creator of RubyPair.com would surely be keen. I'd offer up free publicity on RubySource.com,[3] THE (read: a) place to read about all things Ruby.

[1] http://rubypair.com/
[2] http://rubysource.com/loccasions-pair-programming/linky
[3] http://rubysource.com

Looking on the About[4] page, I saw Evan Light's name. I've followed Evan on Twitter (@elight[5]) for awhile now, and I know that he holds "office hours"[6] where programmers seeking pairing can sign up. I sprang at the opportunity and snagged Evan's Office Hours on September 20th. Now, all I had was the crazy anticipation of waiting for that day, like it was my own Ruby Christmas.

Let There Be (Evan) Light

Evan Light has been doing software development a long time. You can tell by reading his highly informative blog[7] and his bio[8]. Evan hasn't always done Ruby, and in fact took a great risk to get to his current station in life.[9] Evan, in my opinion, exemplifies why the Ruby community is, how-you-say?, super-fantastic. He is smart, approachable, and passionate about helping others learn Ruby. If you need an awesome Ruby resource, he is available for hire (with all the blessings of Rubysource) at TripleDogDare.[10] He is one cool cat (that loves cats).

Am I Worthy?

Being honest, I was a bit apprehensive. I've never *really* paired before. Especially not with a stranger. Especially not with a stranger that is more experienced/smarter/better at this Ruby stuff than yours truly. It was a distinct possibilty, in my brain, that the session would be highly frustrating for him and that I, quite possibly, would cry. (Continuing with the dramatic theme …) I stood firm, though, knowing that sacrificing for your art is a sign of greatness.

The Day Arrives

As 9:50pm I start the stare-blankly-at-my-screen-worrying-about-my-impending-embarrassment trance at my desk. Ten minutes later, the Skype ring jolts me back to reality. It's Evan. I answer, "Hi."

[4] http://rubypair.com/about
[5] http://twitter.com/elight
[6] http://goo.gl/0Txzn
[7] http://evan.tiggerplace.com/
[8] http://www.tripledogdare.net/
[9] http://evan.tiggerpalace.com/articles/2011/08/28/getting-free/
[10] http://tripledogdare.net/

"Hello, Glenn. Are you ready to go?" he asks.

"Um, sort of. I have to be honest, here, I've never done this before."

"What? Programmed?" he chuckles. "It's okay, I'll get us going."

I had previously told Evan about Loccasions, and he had cloned my GitHub repository. He told me that all he needed from me was my SSH public key. I sent him the key, and he tells me to ssh to his box (he set me up a user and he has a cool DynDNS-type domain), which I do. My next instruction was to type `tmux -S /tmp/glenn attach`. I had not heard of `tmux` before, but I threw caution to the wind and typed the command.

Revelations

So, `tmux` is insanely fantastic. The command I entered connected me to a `tmux` session that Evan had started. The terminal window was split into a command line and an editor (vim) window with the Loccasions source in it. The kicker? We both can control/type/etc the session in real time.

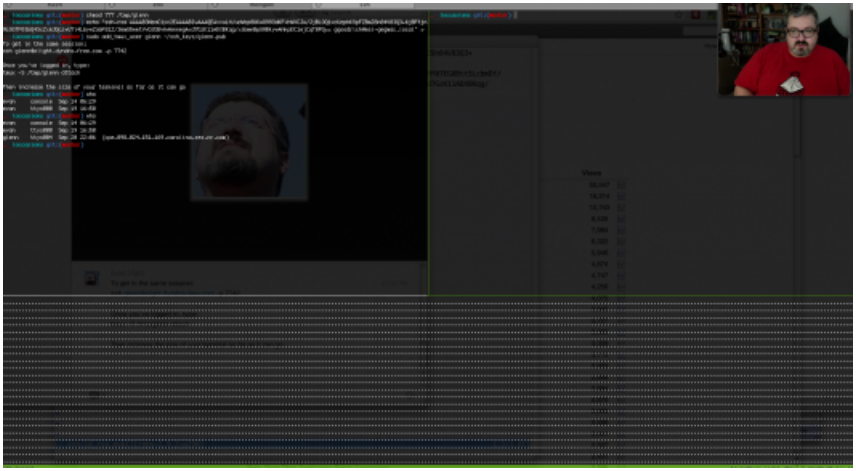

Figure 9.1. TMUX Session Window

There is no handing control back and forth. It's real-time editing and hacking as if you were sitting next to each other. Actually, it's better than that, because we don't have to swap seats. You can split the window as much as needed, so we can run

spork in one window, vim in another, and the command-line in another. I can't relay how freaking cool this is for remote code-sharing/pairing.

To make it even cooler, Evan forwarded a port to his development rails server on his box, so when he ran `rails s`, I could open up a local browser and see our changes to the site, again, in real time. I was blown away. Evan, who must be used to seeing mouth-agape, gaffawing dopes like me in a Skype video chat window, shrugs like it's no big deal. The pairing session was already worth it, and we hadn't touched any code. I told him that he needs to blog about this pairing approach. The masses must be informed.

Oh Yeah, We're Supposed to Program

With considerable effort, Evan talked me down off my cloud and put us on task. He had, as I mentioned, cloned the repository, but had not run `bundle install` yet. Here is where we hit our first issue. I hadn't, yet, upgraded Loccasions to Rails 3.1 final (it started life on rc5), so that was step one. Evan cranked open the Gemfile and complimented me for using "twiddle-waka" (~>) the gems in the file. Being honest, I've always done that, but couldn't *really* explain why, but Evan can: Twiddle-waka is usually a safe choice in Gemfiles because tiny releases (The Z in X.Y.Z) are supposed to be safe updates that won't cause breakage as I understand it. Apparently, that's THE thing to do with your Gemfile, and my tension about being a moron eased a bit. Maybe that's why Evan mentioned it, maybe not. Either way, I ceased to worry about the possibility of being told I was worthless and weak and started to get really excited about the session.

(I realize that the approach to gem versioning with Bundler is somewhat debatable. For another approach, read Gem Versioning and Bundler: Doing it Right.[11])

Evan changed the `gem rails` line in the Gemfile to `gem rails ~>'3.1.0'`, which caused Bundler to complain about the `sass-rails` gem. To make Bundler happy again, Sass-rails and coffee-rails also had to point to their respective 3.1.0 versions. The final version change for Rails 3.1.0 was Devise, moving from 1.4.2 to 1.4.6.

Mongoid had recently released 2.2.0, so we changed that too. That last change brought on messages about `bson_ext` needing to be 1.4.0, not 1.3.1. You have to

[11] http://yehudakatz.com/2011/05/30/gem-versioning-and-bundler-doing-it-right/

luuuuuv Bundler (and I do). A quick change of the `bson_ext` version and `bundle install` finished successfully. We're at Rails 3.1.0 Final, progress in pair programming.

The next suggestion Evan had was to use the guard_rspec[12] (based on guard[13]) gem to have the tests automatically run as files are modified. I had been toying with the idea of using autotest, so this was good timing. Neither of us were sure if spork and guard would play nicely, and after our session I went and found guard_spork.[14] Long story short, add:

```
gem 'guard-rspec', '~> 0.4.5'
gem 'guard-spork', '~> 0.2.1'
```

… to the development group of your Gemfile. Finally, update spork to 0.9.0.rc9 (`gem 'spork', '~> 0.9.0.rc9'`) or you'll get a C error when spork builds and `bundle install`. [Note: I am not adding the Mac OS X specific stuff (the `fsevents` and `growl_notify`) gems to the Gemfile. Please read the README at the guard GitHub repository and install the file system events for your system.] With guard installed, we need to tell it about spork and rspec, and we do that with a Guardfile in the root of the app:

```
guard 'spork', :cucumber_env => { 'RAILS_ENV' => 'test' },
               :rspec_env => { 'RAILS_ENV' => 'test' } do
  watch('config/application.rb')
  watch('config/environment.rb')
  watch(%r{^config/environments/.+.rb$})
  watch(%r{^config/initializers/.+.rb$})
  watch('spec/spec_helper.rb')
end
guard 'rspec', :version => 2 do
  watch(%r{^spec/.+_spec.rb$})
  watch(%r{^lib/(.+).rb$})     { |m| "spec/lib/#{m[1]}_spec.rb" }
  watch('spec/spec_helper.rb')  { "spec" }
# Rails example
  watch(%r{^spec/.+_spec.rb$})
  watch(%r{^app/(.+).rb$}) { |m| "spec/#{m[1]}_spec.rb" }
```

[12] https://github.com/guard/guard-rspec
[13] https://github.com/guard/guard
[14] https://github.com/guard/guard-spork

```
  watch(%r{^lib/(.+).rb$}) { |m| "spec/lib/#{m[1]}_spec.rb" }
  watch(%r{^app/controllers/(.+)_(controller).rb$})  { |m|
    ["spec/routing/#{m[1]}_routing_spec.rb",
    "spec/#{m[2]}s/#{m[1]}_#{m[2]}_spec.rb",
    "spec/acceptance/#{m[1]}_spec.rb"] }
  watch(%r{^spec/support/(.+).rb$}) { "spec" }
  watch('spec/spec_helper.rb') { "spec" }
  watch('config/routes.rb') { "spec/routing" }
  watch('app/controllers/application_controller.rb')
➥{ "spec/controllers" }
  # Capybara request specs
  watch(%r{^app/views/(.+)/.*.(erb|haml)$}) { |m|
"spec/requests/#{m[1]}_spec.rb" }
end
```

Opening up a new terminal and typing guard should fire up a guard (and spork) session.

Figure 9.2. On Guard!

Now, as we modify test and application files, the specs should run automatically. I like that feature – another notch on the stick for pair programming. (After our session, I noticed that my Selenium tests were unable to connect to Firefox 7, so I updated Capybara to 1.1.1 in the Gemfile as well. Do that too.)

Feature of the Day

Once we had the environment up and running with guard, Evan asked me what feature I wanted to add during our session. I had very ambitious ideas of what we could do, but we'd start with creating the page to show a singular Event. In Rails terms, I wanted to create the events#show sequence (here I am using "sequence" to refer to the controller => view needs to satisfy the *show* ing of an event). So, I moved

to the *spec* directory and created an **acceptence/show_event_spec.rb**. This was Evan's first chance to see how I was, basically, relying entirely on acceptance tests to exercise the app. He told me how he was moving away from acceptance tests for performance and syntactical reasons. Evan wants something like Cucumber, but not Cucumber. He actually wrote a DSL, called Couda[15] that emphasizes the Given-When-Then approach to testing, but doesn't rely on regular expressions like Cucumber. Check it out.

Evan told me of experiences of having hundreds of specs to run, where he had to use things like parallel_tests[16] to get the tests to run within 15 minutes. In fact, we talked about all kinds of stuff, which is probably the biggest takeaway of the session. Sure, we added a feature to Loccasions, but it was the chatter around what we were doing and around Ruby/Rails that I found fascinating.

Okay, Okay, the ACTUAL Code

Rather than step through the actual sequence of events that led to creating the Show Event page, I'll simply put the code in here. At a high level, Evan did some coding, I did some coding (he was really good about making me do some of the work, as I was content to sit back and watch) and we ended up with a basic page. Here is the spec for the page:

```
require 'spec_helper'
feature 'Show Event', %q{
  As a registered user
  I want to see an Event
  so I can see my Event Details
} do
background do
    @user = Factory(:user)
    @event = Factory(:event, :user => @user )
    login_user @user
  end
scenario "Show Event" do
    click_link "Show Details"
    page.current_path.should == event_path(@event)
    page.should have_content(@event.name)
```

[15] http://coulda.tiggerpalace.com/
[16] https://github.com/grosser/parallel_tests

```
    page.should have_content(@event.description)
  end
end
```

Add the "Show Details" link to **app/views/events/index.haml.html** (Thanks Alert Reader rsludge!) Here is the new `ul` and loop for events.

```
%ul#events
  - for event in @events
    %li{:class => @event == event ? :selected : nil}
      %span.del_form
        =button_to "X", event, :confirm => "Are you sure?", :method => :delete
      %span.event_name
        = link_to event.name, edit_event_path(event)
      %span.event_details
        = link_to "Show Details", event_path(event)
      %span.event_description= event.description
      %div.clear
```

If you've been keeping up, then you know what is coming. The spec fails, because there is no `events_controller#show` method:

```
def show
    @event = current_user.events.find(params[:id])
end
```

Next, it'll ask for the view template (**app/views/events/show.html.haml**):

```
%h2= @event.name
.description= @event.description
```

Yes, I know it isn't much, but it won't be until we add Occasions (next chapter, I promise!). Creating this view makes the spec pass, and we had a new feature.

Time Flies

At the end of this spec, we had just about reached an hour. I mentioned adding Occasions, and Evan (understandably) said we were at a good place to stop. I agreed,

and he sent me the code. You can see all the changes,[17] which actually include some minor changes from previous comments (thanks Alert Readers!)

Go and Pair

All in all, I really enjoyed the pairing session with Evan. I learned a ton in an hour, and Loccasions is better for it. Evan is passionate about helping the Ruby community, and he has found a manner (the `tmux`/Skype crazy-awesome) and a message (RubyPair[18]). I encourage you to sign up at RubyPair, find another programmer, and do some work. Evan is always looking for pairers to work on RubyPair, so sign up on his Office Hours[19] and help out. Finally, I can't adequately express my appreciation to Evan for allowing me to document this experience and for being gracious and understanding. Right, now pair up!

[17] https://github.com/ruprict/loccasions/commit/515f6aa255e9a0d3d838d5da8207269a387c76fd
[18] http://rubypair.com/
[19] http://goo.gl/0Txzn

Chapter 10

Hiring a Foreman, Inheriting Resources, and Occasions

In this chapter, I want to finish the Occasions MVC sequence. First, though, let's make firing up the development environment a bit easier. Maybe that will kick-start our productivity …

Hiring a Foreman

Every time I want to hack on Loccasions, I have to fire up guard, a web server (`rails s`, for now), and mongodb along with my vim session. Without fail, I forget to fire up mongodb, so guard blows up all over the place. It's an annoying time-waster and also puts me in a bad mindset at the start of my hack session. I would like to clean this up a bit, so I am bringing in Foreman.[1] Foreman is a "manager for Procfile-based applications", which Google will tell you means you can create a *Procfile* (we'll put ours in the root of the app) and list out the processes we want Foreman to start up.

[1] http://rubydoc.info/gems/foreman/0.24.0/frames

That's sounds positively smashing to me, so I add `gem foreman, "~> 0.24.0"` to the `:development` and `:test` groups in my Gemfile, quick `bundle install` and foreman is officially on the payroll. I have three processes I want to run in development: `mongod`, `guard`, and `rails s`, so my *Procfile* looks like:

```
web: rails s
test: guard
db: mongod --dbpath=/Users/ggoodrich/db/data
```

Now, I can type `foreman start` in my application directory and Foreman will start these three processes.

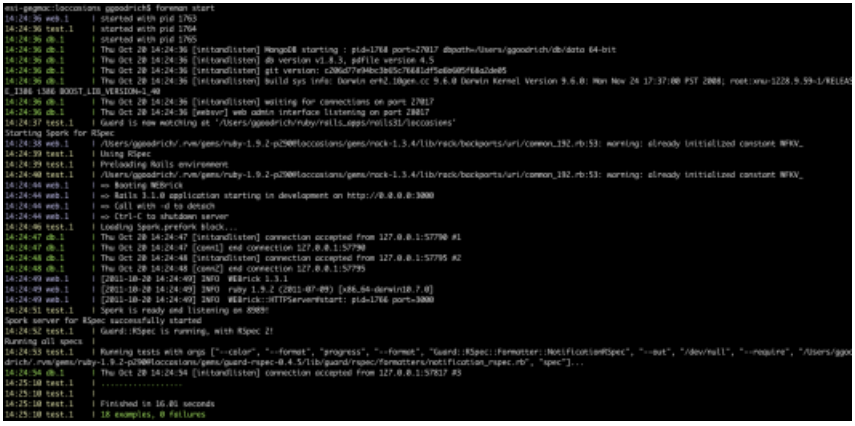

Figure 10.1. No hard hat here

I like to imagine a scruffy, hard-hat wearing dude screaming at the processes ("ALL RIGHT, Database! Get off your lazy shard and prepare for data!") Although, being honest, I think a better name for Formean would have been Procadile. I can already see the logo…maybe I need to get a non-programming hobby…

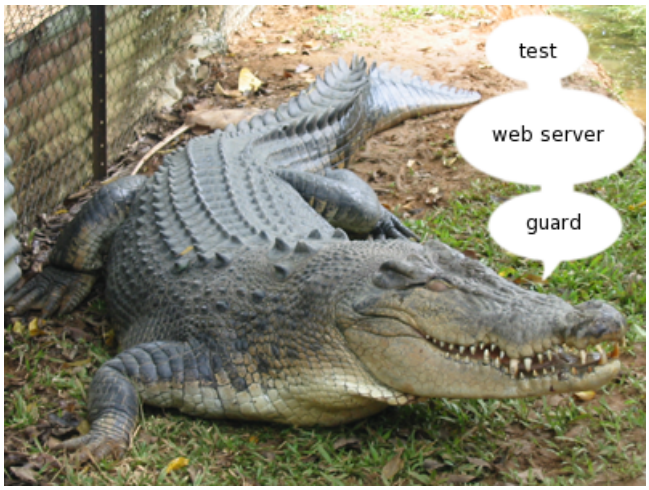

Figure 10.2. Okay, maybe the logo would be better than this ...

Occasions

We're finally to a point where we can design how we'll add Occasions. Occasions, as you may remember, belong to an Event. An Occasion is an individual occurrence of that Event. So, if your Event is "Selling Girl Scout Cookies", then an Occasion for that event might be "February 2nd, 2010" with the lat/long of 35.223/-85.443 (My Neighbor's House), and a note of "2 boxes of Samoas". Another Occasion for that Event could, then, have a date of February 10th, 2010, with the lat/long of (lat/long for my kid's school), and a note that says "Mrs. Whatsherface bought 1 box of Thin Mints."

Let's write some unit tests around that idea. Put this in **spec/models/occasion_spec.rb**:

```
require 'spec_helper'
describe 'Occasion' do
  before do
    @event = Factory.build(:event)
    @occasion = @event.occasions.build
  end
  it "should belong to an event" do
    @occasion.event.should_not be_nil
  end
  it "should have a time and date of occurrence" do
    dt = Time.now
```

```
    @occasion.occurred_at = dt
    @occasion.occurred_at.to_s.should == dt.to_s
  end
  it "should have a latitude and longitude" do
    @occasion.latitude = -85.000
    @occasion.longitude = 35.3232
    @occasion.latitude.should == -85.000
    @occasion.longitude.should == 35.3232
  end
  it "should have a note" do
    @occasion.note = "This thang went down"
    @occasion.note.should == "This thang went down"
  end
end
```

These tests fail, because we haven't created an Occasion model and Event doesn't have a `occasions` method. A quick `rails g model Occasion occurred_at:datetime latitude:float longitude:float note:text -s` will take care of that. (Note: the `-s` skips existing files, which is our spec file that we already created). We have to modify the generated model file to tell it that it lives in Events. Here is our **app/models/occasion.rb** file (I've gone ahead and added validations and accessors):

```
class Occasion
  include Mongoid::Document
  field :occurred_at, :type => Time
  field :latitude, :type => Float
  field :longitude, :type => Float
  field :note, :type => String
  embedded_in :event, :inverse_of => :occasions
  validates :occurred_at, :latitude, :longitude, :presence => true
  attr_accessible :occurred_at, :latitude, :longitude, :note
end
```

Also, open up **models/event.rb** and add `embeds_many :occasions` below the `embedded_in :user` line. I realized, looking at this file again, that I had neglected to define which attributes on Event should be accessible. This is bad mojo, so I added `attr_accessible :name, :description` to the Event model.

Changing Our Spork Configuration

In the midst of writing the Occasion model spec, I added a new factory to create an Occasion in **spec/factories.rb**:

```
factory :occasion do
  latitude 35.1234
  longitude -80.1234
  occurred_at DateTime.now
  note "Test Occasion"
  event
end
```

With my new factory, I changed the `before` block in the occasion spec to use it. This resulted in my specs blowing up all over the place with errors like:

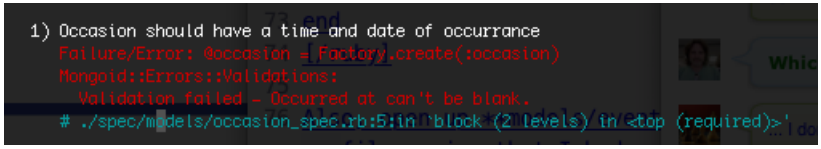

Figure 10.3. Donde esta mi factory?

So, my new-fangled Spork/Guard super fantastic environment wasn't reloading the factories. I frantically turned to Google and asked "WHAT NOW??!?" Google calmly replied, "Put this in the Spork.each_run block in your **spec/spec_helper.rb** file, my man.":

```
# Reload our factories
FactoryGirl.factories.clear
Dir[Rails.root.join("spec/factories.rb")].each{|f| load f}
```

Guard knows to reload the RSpec environment when you mess with **spec_helper.rb**, so my tests were happy again. While we are in there, let's add something to reload the routes too:

```
# Reload routes
Loccasions::Application.reload_routes!
```

Now that we have a model, we need a way to create them.

You Say ~~Potatoe~~ "Hurry up", and I Say ~~Potahtoe~~ "Occasions Controller"

At this point, we should all be Olympic Gold Medalists at creating the vanilla Rails Controller for a resource. In this case, our resources are Occasions. Go ahead and try to get a working (and spec'd) controller for Occasions up and running. You can check what I did with this gist[2] and see how it came out.

Inherited Resources

WHOA! What's up with THAT gist? That doesn't look like what we did for the events controller. You're right, it doesn't look like that. I tricked you. Jose Valim of Plataformatec (and Crafting Rails Applications[3]) fame created the inherited_resources[4] gem to address the fact that 95% of all RESTful controllers in Rails do the same stuff. Using Jose's gem, we can have our OccasionsController inherit from `InheritedResources::Base` and we get the 7 ~~Deadly~~common controller actions for free. I *heart* this community. (BTW, now we be a good time to add `gem "inherited_resources", "~> 1.3.0"` to your Gemfile and `bundle install` that baby.)

In this case, it's not *totally* free, though, as we have to do some configuration to handle our "special" circumstances. These circumstances relate mostly to our using MongoDB and the fact that Occasions are embedded within a document hierarchy (User ==> Events ==> Occasions). If you try to do something like `Occasion.where(:event_id => @event.id)` or whatever, you get the following error that scares the hell out of you the first time you see it:

```
Mongoid::Errors::InvalidCollection: Access to the collection for
Occasion is not allowed since it is an embedded document, please
access a collection from the root document.
```

Once you calm down, you realize that this makes total sense. Because we are using a document database, occasions are *embedded within* events and events are *embed-*

[2] https://gist.github.com/1352047
[3] http://pragprog.com/book/jvrails/crafting-rails-applications
[4] https://github.com/josevalim/inherited_resources

ded within users. So, rather than use the regular ActiveModel class methods to access the collections, you have to walk down the document hierarchy. We need a user (`current_user`, which we are already using to scope events), and an event. Where do we get the event?

The route parameters have a `:event_id` entry so, if we were doing this ourselves, we'd grab that and query the `current_user.events` collection. This is a pretty common scenario, and the inherited resources gem is crazy smart about common scenarios. Let's take a look at this configuration in the **app/controllers/occasionscontroller.rb** file:

```
belongs_to :event
actions :all, :except => [:show, :index]
def begin_of_association_chain
  current_user
end
```

But wait! There's more!! You see that `action` method call up there? That tells inherited_resources which actions we want (or don't want, in this case) for our controller. Occasions will only ever been seen through an Event, so there is no point in creating the `show` and `index` actions (we will change our mind when we get to the Loccasions API) right now. The truly perceptive among you are now asking "But, what about redirects?", which is a great question. A common idiom for Rails RESTful controllers is to redirect to the index or show page after resource creation. Again, we aren't going to do that here, we want to go to the `events#show` action. The inherited_resources gem has a feature called "Smart redirects" that (from their GitHub page[5]:)

> Redirects in create and update actions calculates in following order resourceurl, collectionurl, parenturl (which we are going to see later), rooturl. Redirect in destroy action calculate in following order collectionurl, parenturl, root_url.

In other words, it figures out what we want. I squealed like a little girl when I found that feature. (To be fair, though, I squeal a lot.)

[5] http://github.com/josevalim/inherited_resources

Pretty straightforward, and we've reduced the amount of code we need to write. Occasions can be added to an event. I've written the **spec/acceptance/add_occasions_spec.rb** and **delete_occasions_spec.rb**. I am not currently going to worry about update, because I am having a problem seeing the use case. I am sure we'll be back to update later, but right now I want to get to the map.

Update: Alert Reader Nicholas Henry points out in the comments that you need to:

- Amend **events/show.html.haml** with the Occasion form GitHub[6]
- Add **occasions/_occasion.html.haml** GitHub[7]
- Add the route for occasions GitHub[8]

Loccasions.map do { |its| about.time()}

Well, almost … the map will be covered in the next chapter.

[6] https://github.com/ruprict/loccasions/blob/master/app/views/events/show.html.haml
[7] https://github.com/ruprict/loccasions/blob/master/app/views/occasions/_occasion.html.haml
[8] https://github.com/ruprict/loccasions/blob/master/config/routes.rb

Chapter 11

Going Client-side with Leaflet, Backbone, and Jasmine

We've finally arrived at the moment of the map. For the last few chapters, I've promised things like "in the next chapter we will deal with the map" and "I will lower taxes," and I haven't delivered. Here, I'll fulfill at least one of those promises.

Adding the map to this application is almost completely a client-side proposition. As such, this chapter (and one or two following it) will be a metric ton of JavaScript and a thimble's worth of Ruby.

Libraries, Frameworks, and Maps, OH MY!

As I mentioned when setting up this application in Chapter 4, I use Backbone[1] as the framework for the JavaScript and Jasmine[2] as my client-side testing framework. My justifications for this choice is that I like both frameworks… a lot.

[1] https://github.com/documentcloud/backbone
[2] http://pivotal.github.com/jasmine/

I do not, however, use a gem to generate Backbone files or magically hook up Backbone to the server. The plan is to write the Backbone classes from scratch, adding in the config to get them talking to the server as needed. I have absolutetly nothing against using a gem for Backbone and rails, other than there seems to be confusion about which one does what (If you go to the rails-backbone[3] repo, it tells you to use `gem "backbone-rails"`, but if you go to the backbone-rails[4] repo, it tells you to use `gem "rails-backbone"`. I was in an infinite loop looking at both of them, saved only my wife cutting the power to my office.

Also, I won't cover the basics of Backbone, as that has[5] been[6] done.[7] If you don't understand Backbone, spend some time learning the basics, which should more than prepare you for this article.

For Jasmine, I use the jasmine gem[8] because it's backed by Pivotal and they rock. However, for Loccasions I'll employ a specific branch[9] of the gem, which is capable of running the specs and JavaScript source files through the asset pipeline. We are, after all, using Rails 3.1.

Web-based maps are all JavaScript, all the time these days. Choosing the right js-map framework was a bit daunting, especially since I've spent much of my career using one (ESRI's ArcGIS Server, if you're interested. You're not.)

- Google Maps is out, due to the possibility of Google pulling a Crazy Ivan[10] about licensing.
- Yahoo is out, because frankly, because it's dead.
- I briefly looked at OpenLayers[11] and may yet use it, since it's true open source, which I like.

[3] https://github.com/ivanvanderbyl/rails-backbone
[4] https://github.com/ivanvanderbyl/rails-backbone
[5] http://liquidmedia.ca/blog/2011/01/Backbone-js-part-1/
[6] http://thomasdavis.github.com/tutorial/Backbone-introduction.html
[7] http://backbonetutorials.com/
[8] https://github.com/pivotal/jasmine-gem
[9] https://github.com/pivotal/jasmine-gem/tree/1.2.rc1
[10] http://googlegeodevelopers.blogspot.com/2011/10/introduction-of-usage-limits-to-maps.html
[11] http://openlayers.org/

- I also tried to use Mapstraction,[12] which abstracts away the provider and let's you change providers on the fly. However, when I tried using OpenLayers with Mapstraction, I couldn't figure out how to change the theme, so I moved on.
- Finally, I settled on Leaflet,[13] mainly because it looks great and I like the api so far.

The last bit is a bleeding-and-we're-taking-blood-thinners addition to the application. I've used Backbone a few times and you usually need templates to present your model and collection data in the HTML. I really didn't want to bring in another templating language (in addition to haml) if I could avoid it, so I found haml-coffee.[14] This gem allows you to write your templates using HAML, and then makes them available in your JavaScript on the client. We'll run through at least one scenario soon that shows this clearly.

Setup

Whew! Now, let's get the application setup with all our new client-side yummies. First, download all the code we'll need:

- Leaflet[15] put this in **vendor/assets/javascripts/leaflet/leaflet.js**. Leaflet also has a CSS stylesheet and images. Put these (css files and images directory) in **vendor/assets/stylesheets/leaflet**
- Backbone[16] put this in **vendor/assets/javascripts/Backbone/Backbone-min.js**
- Underscore[17] put this in **vendor/assets/javascripts/Backbone/underscore-min.js**

I also created a **vendor/javsascripts/vendor.js** file that will load these files. The asset pipeline will do this automatically, but, um, I am a control freak:

[12] http://mapstraction.com/
[13] http://leaflet.cloudmade.com/
[14] https://github.com/9elements/haml-coffee
[15] http://leaflet.cloudmade.com/download.html
[16] http://documentcloud.github.com/backbone/backbone-min.js
[17] http://documentcloud.github.com/underscore/underscore-min.js

```
// vendor/assets/javascripts/vendor.js
//= require leaflet/leaflet
//= require Backbone/underscore-min
//= require Backbone/Backbone-min
```

Similarly, I created a **vendor/stylesheets/vendor.css** file for the Leaflet CSS:

```
/*
 * This is the vendor.css in our vendor/assets/stylesheets dir
 *=require leaflet/leaflet
 *
*/
```

Finally, add a couple of gems to our Gemfile:

```
group :assets do
  ...
  gem 'haml-coffee'
end
group :test, :development do
  ...
  gem 'jasmine', :git => 'git://github.com/pivotal/jasmine-gem.git',
                 :branch => '1.2.rc1'
end
```

The aforementioned (I luuuv using that word) `haml-coffee` and `jasmine` (remember, we're using a branch) gems. Oh, and I am sure you remembered to `bundle install`, right?

Now, it's time to set up jasmine, so run `rails g jasmine:install` to set up Jasmine support in the application. Jasmine comes with its own server, which is launched by typing `rake jasmine`. Being super-savvy users of Foreman (remember Chapter 10) we'll add it to our Procfile:

```
web: rails s
db: mongod --dbpath=/Users/ggoodrich/db/data
test: guard
jasmine: bundle exec rake jasmine
```

Next time you `foreman start`, Jasmine will be running. The Jasmine server runs at http://localhost:8888, but it won't be very interesting until we get some specs added.

Spec files are added to Jasmine in the **spec/javascript/support/jasmine.yml** file. Because we are using a branch that supports the asset pipeline, our **jasmine.yml** file is a bit different than the one used in the current release. Here is a gist of the one I've set up for Loccasions.[18] One change I made was to pull in our CoffeeScript files (this is what the branch gives us) and manually add our vendor JavaScripts (jQuery, Underscore, Backbone, and Leaflet). The other change is the last line of the file, identifying `app/assets` as a path to be served by the asset pipeline.

Lastly, I want to leverage Sinon[19] for mocking and stubbing when needed in my JavaScript tests. There is a nice plugin to Jasmine for Sinon[20] here. Download both those files into our `spec/javascripts/helpers/` directory. You'll need to add the **spec/javascripts/helpers/jquery.js** file (you can copy it from the jQuery site[21] or from the jquery-rails gem) because Jasmine won't load jQuery yet (they're working on it…).

I would suggest reading through this series about Jasmine and Sinon[22] to become familiar with how this all fits together. You'll no doubt notice its influence on Loccasions.

Whew… easy as, um, really hard pie.

[18] https://gist.github.com/461e8d238a38562e92d2#file_jasmine.yml
[19] http://sinonjs.org/
[20] https://github.com/froots/jasmine-sinon
[21] http://code.jquery.com/jquery-1.7.1.min.js
[22] http://tinnedfruit.com/2011/03/03/testing-backbone-apps-with-jasmine-sinon.html

Client-side Directory Structures, and the Women Who Love Them

For every two Backbone introductory articles, there is one on structuring[23] your[24] Backbone[25] code. The structure of the files is a development-only concern, since the asset pipeline will smush all the JavaScript into one **application.js** file. After reading through some of the articles, I've come up with this:

```
|~app/
| |~assets/
| | |+images/
| | |~javascripts/
| | | |~collections/
| | | | `-eventsCollection.js.coffee
| | | |~lib/
| | | | `-leafletMapProvider.js.coffee
| | | |~models/
| | | | `-event.js.coffee
| | | |~templates/
| | | | `~events/
| | | | |-line_item.js.haml-coffee
| | | |~views/
| | | | |-eventEditView.js.coffee
| | | | |-eventListView.js.coffee
| | | | |-eventView.js.coffee
| | | | `-mapView.js.coffee
| | | |-app.js.coffee
| | | |-application.js
| | | |-events.js.coffee
| | | |-home.js.coffee
| | | |-occasions.js.coffee
| | | |-router.js.coffee
| | | `-tabs.js
```

Figure 11.1. Our, um, Backbone

In a nutshell, the collections, models, views, and templates all get their own directory. The lib directory is for code that is not part of the Backbone structure.

[23] http://backbonetutorials.com/organizing-Backbone-using-modules/
[24] http://weblog.bocoup.com/organizing-your-Backbone-js-application-with-modules
[25] http://jguimont.com/post/11890935147/structuring-and-organizing-a-Backbone-js-app-with

As I mentioned, the asset pipeline in Rails 3.1 will load all this stuff for you, but I want to control the order of how things are loaded. As such, we need to make a change to your **app/assets/javascript/application.js** file, like so:

```
//= require jquery
//= require jquery_ujs
//= require vendor
//= require ./app
//= require ./router
//= require_tree ./lib
//= require_tree ./models
//= require_tree ./collections
//= require_tree ./views
//= require_tree ./templates
```

Setup Complete, Now What?

With all of our client-side whatsits in place, we can look at the design of our Events page from the perspective of Backbone. We'll now flush out the Events#index page, as shown in the following screenshot.

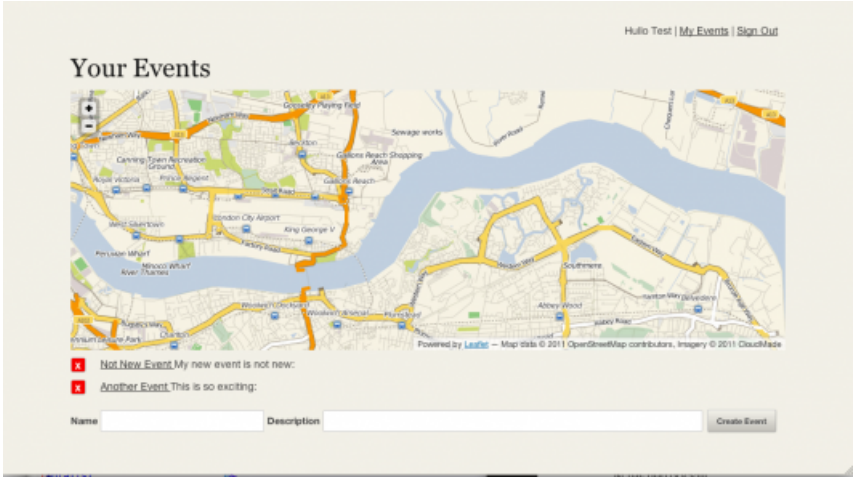

Figure 11.2. What You See

One way to approach this page with Backbone is to slice the page into views, like so:

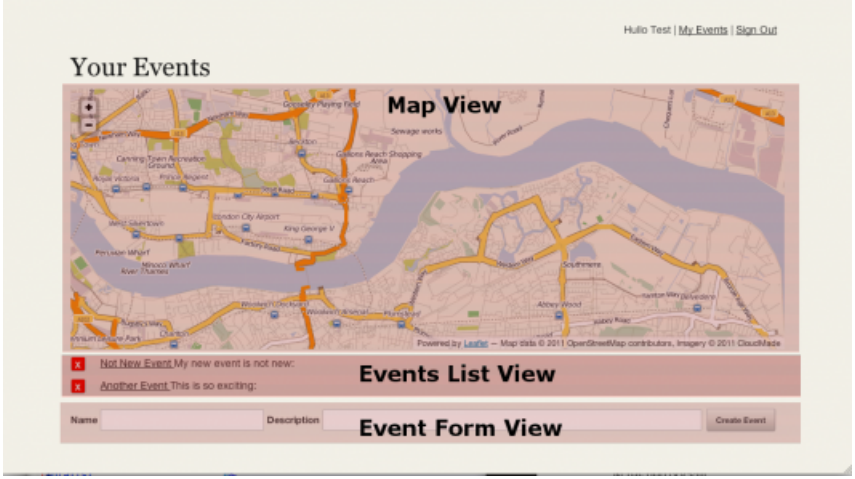

Figure 11.3. What Backbone Sees

This is the approach taken, and should be enough for us to start writing tests.

Gentleman, Right Now on Stage 3, Put Your Hands Together for JAAASSSMMIIIIINE

Let's start with the map view. Here is our first Jasmine test:

```
// spec/javascripts/views/mapView_spec.js
describe("MapView", function() {
  describe("initialize", function() {
    beforeEach(function() {
      loadFixtures("map.html");
    });
    it("should use the #map element by default",
      sinon.test(function() {}));
    it("should create a map", sinon.test(function() {
    }));
  });
});
```

You can see that Jasmine looks a lot like RSpec syntax. There are "describe" blocks (which can be nested) and "it" blocks to test specific behavior. Also, our `beforeEach` allows us to load a fixture file. In this case, our fixture file is adding a `div#map` to the page (really, it's just `<div id='map'></div>`), which will hold our map for the

tests. Client-side testing is often dependent on the structure of the markup, so being able to load fixture files (which Jasmine will unload after the test) is really nice.

Our map views tests are simple. First, make sure that the view uses the `#map` DOM element. Second, make sure it creates a "map."

As quick side note, you may have noticed the `it` functions are wrapped with `sinon.test()`. This creates a "sandbox" for the test. If the MapView has any dependencies (and it does), we'll be stubbing/mocking them as needed. The sandbox makes it easy to restore any stubbed or mocked objects when a test completes.

What is a map, though? I mentioned before that we will use Leaflet, but I am not sure that we won't change map providers. As such, we should hide the map behind an abstraction in the view. Also, I don't really need to bootstrap an actual Leaflet map for my tests. Therefore, I have created the concept of a "MapProvider," which I pass to the view on initialize. We can use Sinon to mock/stub the provider, keeping out tests light.

The MapProvider interface/protocol is very simple right now:

```
App.MapProviders.Leaflet = ->
  # Create new map
  createMap: (elementId) ->
  addBaseMap: ()->
  addLayerToMap: (layer) ->
  setViewForMap: (options) ->
```

Just four functions, named to make their purpose very obvious. The implementation of the Leaflet provider[26] is here if you're interested.

Let's complete our 2 map view tests:

```
 it("should use the #map element by default", sinon.test(function() {
  // Arrange
  var mp = new App.MapProviders.Leaflet();
  var mapSpy = this.stub(mp, "createMap");
  var setViewSpy = this.stub(mp,"setViewForMap");
  //Act
```

[26] https://gist.github.com/461e8d238a38562e92d2#file_app.map_providers.leaflet.js.coffee

```
  var view = new App.MapView({
    mapProvider: mp
  });
  //Assert
  expect(view.el.id).toEqual("map");
  mapSpy.restore();
  setViewSpy.restore();
  </code></pre>
}));
it("should create a map", sinon.test(function() {
  //Arrange
  var mp = new App.MapProviders.Leaflet();
  var mapProviderMock = this.mock(mp);
  mapProviderMock.expects("createMap").withArgs("map").once();
  mapProviderMock.expects("setViewForMap").once();
  //Act
  var view = new App.MapView({
    mapProvider: mp
  });
  //Assert
  mapProviderMock.verify();
}));
```

These two test show different types of testing. The first test ensures that the MapView does the right thing, provided it's dependency does the right thing. We don't really care what the MapProvider does for this test, because it's not pertinent to the outcome of the test. So, we stub those methods out, which stops the calls from getting to the Leaflet API while making sure they don't throw an error.

The second test is an example of "Expectation Based Unit Testing." Here, we DO care how the MapView interacts with its dependency. We *expect* it to call certain methods, and we ask our mock object to verify that those methods were, indeed, called.

Reloading our Jasmine test page (http://localhost:8888, remember?) we see:

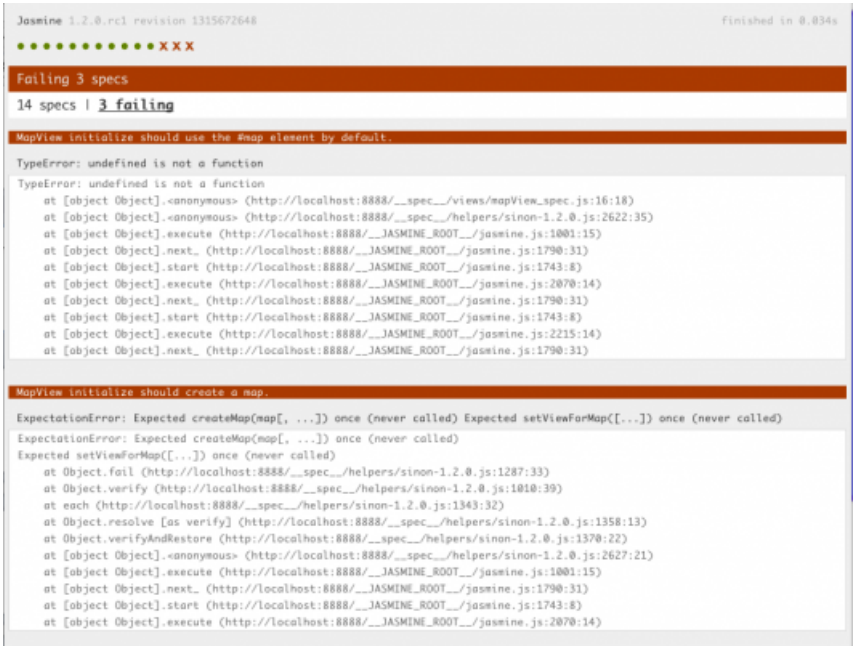

Figure 11.4. Fail

We can fix that.

I'm the Map[View]!

Any Dora fans out there? No? *coughs* Right, me neither. Here's the MapView implementation:

```
###
app/assets/javascripts/views/mapView.js
###
class App.MapView extends Backbone.View
  el: "div#map",
  initialize: ->
    @mapProvider = this.options.mapProvider
    @initialCenter = this.options.initialCenter ||
                     { latitude: 51.505, longitude: 0.09 }
    @render()
  setInitialView: ->
    @mapProvider.setViewForMap
      latitude: @initialCenter.latitude,
```

```
      longitude: @initialCenter.longitude
      zoomLevel: 13
  render: ->
    @mapProvider.createMap(@el.id)
    @setInitialView()
```

Once you add this file, reloading the Jasmine test page looks like:

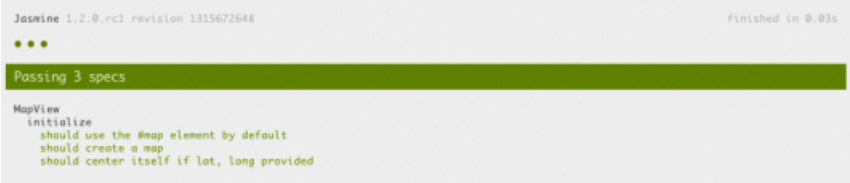

Figure 11.5. Pass

Do You Know the Way to Map, Jose?

Okay, so, we have a MapView. How do we get our page to use it? I mean, I don't see a map when I go to my '/events' page in Loccasions. This is where Backbone's "Router" (or the Artist-Formally-Known-as-Controller) comes into play. In this case, we want to route the "root route" or "/" to a place where it knows to create our MapView. I've written a couple of specs to fulfill this requirement:

```
// spec/javascripts/appRouter_spec.js
describe("AppRouter", function() {
  describe("index", function() {
    beforeEach(function() {
      loadFixtures("map.html");
      this.mapViewStub = sinon.spy(App.MapView.prototype,
                                   "initialize");
      window.bootstrapEvents = [];
    });
    afterEach(function(){
      this.mapViewStub.restore();
    });
    it("should create a map view", function() {
      this.router = new App.Router();
      this.router.index();
      expect(this.mapViewStub).toHaveBeenCalled();
    });
```

```
  });
  describe("/", function () {
    it("should respond to empty hash with index", function() {
      this.router = new App.Router();
      this.routeSpy = sinon.spy();
      try {
        Backbone.history.start({silent:true, pushState:true});
      } catch(e) {}
      this.router.navigate("elsewhere");
      this.router.bind("route:index", this.routeSpy);
      this.router.navigate("", true);
      expect(this.routeSpy).toHaveBeenCalledOnce();
      expect(this.routeSpy).toHaveBeenCalledWith();
    });
  });
});
```

In this case, we use a third Sinon stubby/mocky thingy called a "spy". A spy is like a mock, but you can't set a return value on it. A spy exists just to record if it was called and what arguments were used to call it. That works well for our first test, where we are making sure the initialize method on our MapView is called. The second test uses a spy to make sure that, when the user goes to the "root route", the AppRouter#index function is called.

Reloading our Jasmine page gives us the appropriate failures, so let's write our AppRouter class:

```
###
app/assets/javascripts/router.js.coffee<
###
class App.Router extends Backbone.Router
  routes:
    "" : "index"
  index: ->
    if $('#map').length > 0
      @mapView = new App.MapView(
        mapProvider: new App.MapProviders.Leaflet()
      )
```

Done. The specs pass.

Start Me Up

The last bit we need to put in place is to bootstrap our Backbone application code. In other words, we need to tell Backbone to get the Router in place, create all the views, and whatever else it needs to do. I created a little CoffeeScript file for this:

```
###
app/assets/javascripts/app.js.coffee
###
window.App =
  start: ->
    new App.Router()
    Backbone.history.start(
      pushState: true, root: "/events"
    )
$(App.start)
```

This file defines my application namespace (`App`) and creates our `start` function. The `start` function simply creates our router and tells Backbone to start handling the routing. If you want to understand exactly what `Backbone.history.start` is doing, look here.[27]

Update

Mr. Henry (from the comments below) pointed out the following:

- Add the `div #map` to your **events/index.html.haml**
- Add `#map { height: 350px }` to the **app/assets/stylesheets/events.css.scss**

Here's the page with the map:

[27] http://documentcloud.github.com/backbone/#History-start

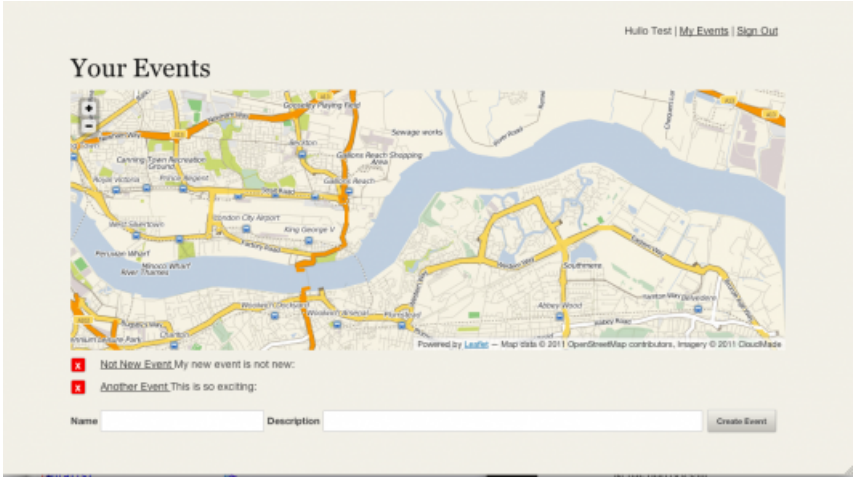

Figure 11.6. What You See

We still have a lot of work to do on this page. However, it's a good stopping point, and I want to play with the map for a bit.

My Blogger Went All Over the Place and All I Got Was This Lousy Map

I know, that was a ton of work just to get a map. However, we also now have:

- A structure for our client-side code.
- A way to drive out the client-side code with tests (TDD/BDD/ETC)
- A MAP!

We'll finish out the Events index page in the next chapter, and then highlight how the Occasions page is different. After that, v0.0.1 of Loccasions should just about be done.

Chapter 12

Getting to Occasions

Last time we completed the client–side items needed to display the Events on the User Events page. Our focus now turns to adding and removing events asynchronously using Backbone.

In our screenshot of the Events page, the only part of the view we have not implemented is the EventFormView. Obviously, this view will be responsible for displaying a form to the user that allows the creation of a new Event. The form is simple:

Figure 12.1. The Event Form

The first thing I want to change is the name. Putting "Form" in the view name seems brittle to me, so let's change it to what we are doing, which is creating an event: CreateEventView.

The CreateEventView should:

- Contain a form.

- Create an Event.

I think it's worth pointing out that we are not testing the event hook-up (meaning, how the form submission event is handled), because that would be testing the Backbone framework. Our tests look like:

```
describe("CreateEventView", function() {
  beforeEach(function() {
    loadFixtures("eventForm.html");
    this.view = new App.CreateEventView();
    this.form = $(this.view.el).find("form")[0];
  });
  it("should provide a form", function() {
    expect($(this.view.el).find("form").length).toEqual(1);
  });
  describe("creating an event", function() {
    beforeEach(function() {
      window.eventCollection = new Backbone.Collection();
      this.createStub = sinon.stub(window.eventCollection, "create");
      $(this.form).find("#event_name").val("Test Event");
      $(this.form).find("#event_description")
                  .val("Test Event Description");
      this.view.createEvent();
    });
    it("should call create on the EventCollection", function() {
      expect(this.createStub).toHaveBeenCalled();
    });
  });
});
```

Running these tests, the jasmine specs go red, as they should. Looking at the "creating an event" spec, we are putting some data into the form, then making sure the `window.eventCollection.create()` is called. Backbone offers the `create` method on collections as a convenience to both create the object and add it to the collection. Nice.

These tests flushed out an issue in dealing with the form. In order to create a `App.TestEvent`, we have to parse the attribute values out of the form. There are many utilities out there that will serialize a form to JSON, but I think we'll handle this ourselves for now.

Unfortunately, we can't just use something as simple as jQuery's `form.serializeArray()` method, because there are values in the form that are not attributes on an Event. An example is the "authenticity_token" used by Rails to help find off CSRF attacks. We don't want that on our Event. I am going to write a utility method to do what I think we need. Test ahoy:

```
describe("parsing form attributes", function() {
  it("should have the correct attribute values", function() {
    $(this.form).find("#event_name").val("Test Event");
    $(this.form).find("#event_description")
                .val("Test Event Description");
    var attributes = this.view.parseFormAttributes().event;
    expect(attributes.name).toEqual("Test Event");
    expect(attributes.description).toEqual("Test Event Description");
  });
});
```

Implementation:

```
parseFormAttributes: ->
  _.inject(
    @form.serializeArray(),
    (memo, pair) ->
      key = pair.name
      return memo unless /^event/.test(key)
      val = pair.value
      if key.indexOf('[') > 0
        parentKey = key.substr(0, key.indexOf('['))
        childKey = key.split('[')[1].split(']')[0]
        if typeof memo[parentKey] == "undefined"
          memo[parentKey] = {}
        memo[parentKey][childKey] = val
      else
        memo[key] = val
      return memo
  ,{})
```

With `parseFormAttributes()` in place, we can finish the `createEvent()`. Here is the entire CreateEventView:

```
App or= {}
App.CreateEventView = Backbone.View.extend(
  el: "#edit_event"
  initialize: ->
    @form = $(this.el).find("form")
  events:
    "submit form" : "handleFormSubmission"
  handleFormSubmission: (e) ->
    e.stopPropagation()
    @createEvent()
    false
  createEvent: ()->
    evento = new App.Event(@parseFormAttributes().event)
    has_id = @form.attr("action").match(/\/events\/(\w*)/)
    if has_id
      evento.id = has_id[1]
      evento.save()
    else
      eventCollection.create(evento)
  parseFormAttributes: ->
    _.inject(
      @form.serializeArray(),
      (memo, pair) ->
        key = pair.name
        return memo unless /^event/.test(key)
        val = pair.value
        if key.indexOf('[') > 0
          parentKey = key.substr(0, key.indexOf('['))
          childKey = key.split('[')[1].split(']')[0]
          if typeof memo[parentKey] == "undefined"
            memo[parentKey] = {}
          memo[parentKey][childKey] = val
        else
          memo[key] = val
        return memo
    ,{})
)
```

Lastly, we have to tell our router to create this view along with the other views:

```
// spec/javscripts/router_spec.js
describe("index", function() {
    beforeEach(function() {
```

```
      ...
      this.createViewSpy = sinon.stub(App, "CreateEventView")
                                .returns(this.mockView);
      this.router.index();
    });
    afterEach(function() {
      ...
      App.CreateEventView.restore();
    });
    ...
    it("should create the CreateEventView", function() {
      expect(this.createViewSpy).toHaveBeenCalled();
    });
  });
});
```

Remembering from the previous chapter, we need to add a method in the `index` method of the router, like so:

```
# app/assets/javascripts/router.js.coffee
index: ->
  @eventListView = new App.EventListView(
    {collection: window.eventCollection or=
➥ new App.EventsCollection()})
  @eventListView.render()
  @createEventView = new App.CreateEventView()
  if $('#map').length > 0
    @mapView = new App.MapView(App.MapProviders.Leaflet)
    @mapView.render()
```

UPDATE: An alert reader (see comments) found some omissions in the article that made it harder to complete… sorry! I really appreciate people finding stuff like this.

You need to make sure your EventsController#create method looks like:

```
def create
  event = current_user.events.build(params[:event])
  event.save!
  respond_with(event) do |format|
    format.html { redirect_to events_path }
  end
end
```

Also, make sure this is at the top of the EventsController class definition:

```
respond_to :html, :json
```

If you don't, the wrong HTTP status is returned and Backbone won't update the view. (Thanks Nicholas!)

If you go to http://localhost:3000 and click through the "My Events" page, you should be able to add Events, and watch them show up in our list.

Deleting Events

The Yin to our adding Events Yang (that sounds kinda dirty…) is deleting events. I am torn on whether or not to include delete functionality on the events#index page as a part of the list. While I can see use cases of wanting to delete, I can also see making them click-thru to the event page to delete the event as a more explicit you-better-know-what-the-hell-you-are-doing UI flow. Let's assume our users are not too click-happy and are grown up enough to handle deleting the events from the list.

One Event at a Time

Awhile back, I decided that the events#show page was going to be a separate page from the events#index page, rather than trying to do a Single Page Application[1] approach. That decision led to the question on how to execute the correct JavaScript on each page. In the event#index case, we have an EventsCollection and views around listing and creating Events. For the events#show page, we'll be focused on a list of Occasions and views around manipulating the Occasions for the current Event.

A bit of searching led me to this post by Jason Garber[2] that expands upon an approach (by the incomparable Paul Irish) to this problem. You should read the post for full details, but the crux of the approach is to create a utility class that calls a load method based on some data-* attributes written on the body element. Following

[1] http://en.wikipedia.org/wiki/Single-page_application
[2] http://www.viget.com/inspire/extending-paul-irishs-comprehensive-dom-ready-execution/

that post's lead, we change our body element in the **app/views/layout/application.haml.html** as follows:

```
%body{:"data-controller" => controller_name,
      :"data-action" => action_name }
```

With that in place, I changed the **app/assets/javascripts/app.js.coffee** to include our new `util` class:

```
window.App =
  common:
    init: ->
  events:
    init: ->
    index: ->
      window.eventCollection = new App.EventsCollection(bootstrapEvents)
      new App.EventsShowRouter()
      Backbone.history.start
        root: "/events"
    show:
      new App.EventRouter()
      ev_id =  location.href.match(/\/events\/(.*)/)[1]
      Backbone.history.start
        root: "/events/"+ev_id
UTIL =
  exec: ( controller, action )->
    ns =  App
    action or= "init"
    if ( controller != "" &&
      ns[controller] &&
      typeof ns[controller][action] == "function" )
      ns[controller][action]()
  init: ->
    body = document.body
    controller = body.getAttribute( "data-controller" )
    action = body.getAttribute( "data-action" )
    UTIL.exec( "common" )
    UTIL.exec( controller )
    UTIL.exec( controller, action )
$(UTIL.init)
```

As a part of this change, I renamed `App.Router` to `App.EventsRouter`, created a `app/assets/javascripts/routers` folder and copied the newly renamed **eventsRouter.js.coffee** into that directory. I also had to rename the spec to **eventsRouter_spec.js** and modify both files, changing `App.Router` to `App.EventsRouter`. After each change, I reran my Jasmine suite and fixed things until the suite passed. I love having tests!

The last accommodation for this change was to change **app/assets/applications/js**, removing:

```
//= require router
```

```
//= require_tree ./routers
```

… and removing the `$(App.start)` call from the bottom of that file.

All the specs should pass, and the existing functionality should work again. Now we can focus on a single Event and its Occasions.

Finally, an Occasion for Occasions

We can officially call this the "downhill slope." Once we can add occasions and see them on a map, we are very close to done.

The approach to this page will be very similar to the Events page. As such, I think it's a fine opportunity for you, the Loccasions reader, to attempt to create the event#show page on your own. At a minimum, the page should be able to:

- Add Occasions
- Delete Occasions
- List out all the Occasions for the Event

As a starting point, here is what mine looks like:

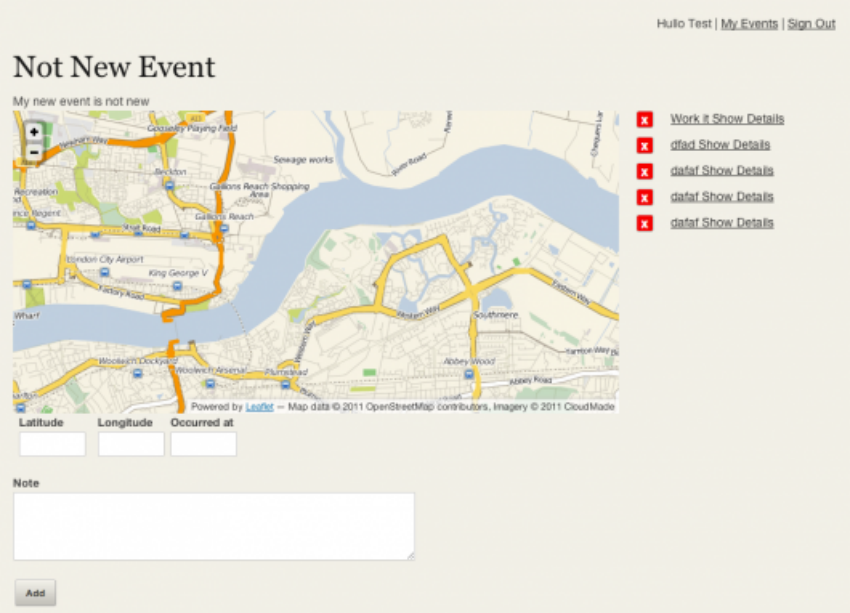

Figure 12.2. Shoot for this, but make it "yours"

Again, we have three Backbone view areas: the map, the list of Occasions, and the form to create a new Occasion. I put the list off to the right of the map in this view, just to be different. For extra credit, you can layout your page differently too.

For a couple of more clues, I created an `App.EventRouter` for the event show page (which you see mentioned in our UTIL code above). After that, it was almost a matter of copying the Event specs, changing them to handle Occasions, and then making those specs pass. If you get stuck, go to the Git repository[3] and see where I ended up.

The next chapter will cover interacting with the map, where we'll take the Occasion form and integrate it with some map functionality. The last chapter will be a retrospective of what could have been done better (wow… that could be a LOOOOOONG one) and what could still be done with Loccasions.

[3] http://github.com/ruprict/loccasions

Chapter 13

Bubbly Map Events

We'll now focus on reacting to map events; in this case, the user interacting with the map to add Occasions. Specifically, I want the user to be able to add an Occasion by clicking on the map. I'll use a cool map bubble to present the attributes form for the Occasion, and we'll post the new Occasion back to the server once the user submits the form. Easy, right?

Responding to Map Clicks

The first task is to respond to the user clicking on the map. All these new-fangled JavaScript map frameworks make this a piece of cake.

Going way back to one of the first articles, I have abstracted the Leaflet map framework into a provider object. In order to allow the code to set a handler for the map click event, I am going to add a `addClickHandler` event to that provider object. Since all it does is delegate the call to Leaflet, I am not going to write a test for it:

```
// in app/assets/lib/leafletMapProvider.js.coffee
...
addClickHandler: (map, callback) ->
  map.on('click', callback)
```

This method will be called in the render method of our map view:

```
// the test
// in spec/javascripts/views/mapView_spec.js
describe("render", function() {
    beforeEach(function() {
      this.mock = this.mapProviderSpy
    });
    it("should add a click handler to the map", function() {
      this.view.render();
      expect(this.mapProviderSpy.addClickHandler)
            .toHaveBeenCalled();
    });
  });
```

```
// in app/assets/javascripts/views/map_view.js.coffee
...
render: ->
    @map = new @mapFactory.Map(@el.id)
    @mapFactory.addClickHandler(@map,@newOccasion.bind(@))
    @addBaseLayer()
    @setInitialView()
```

Our render method is getting pretty busy, but I can live with it. You can see that I am passing in a newOccasion method as the handler. This is the function that will receive our map event and, by my estimation, it needs to do two things:

1. Put a marker on the map where the click occurred.

2. Show the new Occasion form.

Here are our tests:

```
// in spec/javascripts/views/map_view.js.coffee
describe("When a new Occasion is requested", function() {
    beforeEach(function() {
```

```
      var e = {
        latlng: {lat: 100.00, lng: 100.00}
      };
      this.view.newOccasion(e);
    });
    it("should have a form", function() {
      expect($("#new_occasion_form").length).toEqual(1);
    });
    it("should add a marker to the map", function () {
      expect(this.mapProviderSpy.addNewOccasionMarker)
        .toHaveBeenCalled();
    });
});
```

And the code:

```
// in app/assets/javascripts/views/mapView.js.coffee
newOccasion: (e) ->
    @mapFactory.addNewOccasionMarker(@map, e)
```

I neglected to show you the implementation of the `addNewOccasionMarker` on the Leaflet map provider, so here it is:

```
// in app/assets/lib/leafletMapProvider.js.coffee
...
addNewOccasionMarker: (map, e) ->
    ll = new L.LatLng(e.latlng.lat,e.latlng.lng)
    marker = new L.Marker(ll)
    map.addLayer(marker)
```

At this point, if you go to a specific event page, you should be able to click on the map and markers will show up where you click. Kinda fun, isn't it?

The second part of adding a new Occasion is to show the form. We already have a form to create Occasions on the page, but we don't want it there. We want the form in our fancy map bubble. To make our map bubble dreams come true, I did the following:

Change the Event Show View

The event#show HAML template currently has:

```
%div.clear
%div#edit_occasion{ :style => "display:none"}
  = form_for [@event,
              current_user.events.find(@event.id).occasions.build()]
➥do |f|
    %div.coordinate_input
      = f.label :latitude
      = f.text_field :latitude
    %div.coordinate_input
      = f.label :longitude
      = f.text_field :longitude
    %div.date_field
      = f.label :occurred_at
      = f.text_field :occurred_at
    %div.note_field
      = f.label :note
      = f.text_area :note
    = f.submit "Add"
```

I changed the form from having an ID of "edit_occasion" to a class of "edit_occasion". In other words, change the "#" to a ".".

Remove the CreateOccasionView Call from EventRouter

I was new-ing up an the CreateOccasionView inside our EventRouter. I don't want to do that anymore, so take that call out:

```
App.EventRouter = Backbone.Router.extend
  routes:
    ""       : "index"
  index: ->
    @occasionListView = new App.OccasionListView
      collection: window.occasionCollection or= new
➥App.OccasionsCollection()
    @occasionListView.render()
    @createOccasionView = new App.CreateOccasionView() # REMOVE
```

```
    if $('#map').length > 0
      @mapView = new App.MapView(App.MapProviders.Leaflet)
      @mapView.render()
```

Create a CreateOccasionView When the Map is Clicked

Since I want the view to show on map click, we can put a call to show that view in the same place we create the marker:

```
// in app/assets/javascripts/view/mapView.js
newOccasion: (e) ->
    view = new App.CreateOccasionView()
    view.render()
    @mapFactory.addNewOccasionMarker(@map, e,view.el )
```

I pass the event and the view render into the map provider, because I don't want my mapView to know anything about the event contents. The `addNewOcccasionMarker` function will deal with getting the coordinates and populated the form inputs. This is, admittedly, a bit messy, but we're on a fake deadline here.

Because we are showing the form every time the user clicks, I am going to clone the original form and use it as a template for each CreateOccasionView:

```
//in app/assets/javascripts/lib/leafletMapProvider.js.coffee
...
addNewOccasionMarker: (map, e, content) ->
    ll = new L.LatLng(e.latlng.lat,e.latlng.lng)
    marker = new L.Marker(ll)
    marker.bindPopup(content)
    map.addLayer(marker)
    marker.openPopup()
    $("#occasion_latitude").val(e.latlng.lat)
    $("#occasion_longitude").val(e.latlng.lng)
    $("#occasion_occurred_at").val(new Date())
```

This (overly) busy function is creating our marker and putting the latitude, longitude, and occurred date into the form. I should probably hide those form inputs from the user, eh?

```
%div.edit_occasion{ :style => "display:none"}
  = form_for [@event, current_user.events.find(@event.id)
➥.occasions.build()] do |f|
    %div.coordinate</em>input
      = f.hidden_field :latitude
    %div.coordinate_input
      = f.hidden_field :longitude
    %div.date_field
      = f.hidden_field :occurred_at
    %div.note_field
      = f.label :note
      = f.text_area :note
    = f.submit "Add"
```

I went ahead and removed the labels, too. This is what our cool new form looks like:

Figure 13.1. 0000... Pretty map form...

The even cooler thing is, it works. Type in a note and hit 'Add' and blammo! you have a new Occasion show up on the map and in the list next to the map. Backbone is just cool.

More Housekeeping

I am sure you have spent the last 45 minutes creating Occasions like a maniac. I can't blame you, really, it's pretty danged exciting. I bet you said, at least once, "I wish the form would go away when I create the Occasion." Well, strap in, sporto, today is your lucky day.

The question of making the form disappear lead me to the realization that part of my design was, um, how do I put this mildy…, dog vomit. First of all, the MapProvider is an okay idea, but it should have returned a Map object with all the methods I need. The current approach of calling methods on the MapProvider and passing in the map is, as I said, vomitus caninus.

If I ever refactor this app, I will likely start there. As it stands, I need to get this working using the current design so I can finish this article.

Back to making the form disappear. It's easy enough to do, and it has given me the opportunity to show a cool Backbone feature: custom events. As you might've guessed, custom events allow you to trigger your very own named events and then bind to them as needed. I am going to use this to indicate to the MapView that an Occasion has been created.

The CreateOccasionView is in charge of creating the new Occasion (duh) so I am going to raise a custom event from that view called "map:occasionAdded.":

```
// in app/assets/javascripts/views/createOccasionView.js.coffee
...
createOccasion: ()->
    occasion = new App.Occasion(UTIL.parseFormAttributes(@form,
➥ "occasion").occasion)
    has_id = @form.attr("action").match(/\/occasions\/(\w*)/)
    if has_id
      occasion.id = has_id[1]
      occasion.save()
    else
      occasionCollection.create(occasion)
      $(this.el).trigger('map:occasionAdded', occasion)
```

All it takes is one line and we're cooking with custom events. I'll bind to this event in the MapView and tell the MapProvider (ugh) to hide the popup:

```
//in app/assets/javascripts/mapView.js.coffee
App.MapView =  Backbone.View.extend
  el: "div#map"
  events:
    "map:occasionAdded" : "handleSubmit"
...
  handleSubmit: ->
    @mapFactory.hidePopup(@map)
```

And the method called in the map provider:

```
// in app/assets/javascripts/lib/leafletMapProvider.js.coffee
...
 hidePopup: (map)->
    map.closePopup()
```

So, that's great, the popup is gone, baby, gone. However, we have a remaining issue. If you click on the marker you just created, it shows the form. I don't want that, I want it to show the note like the other markers, like so:

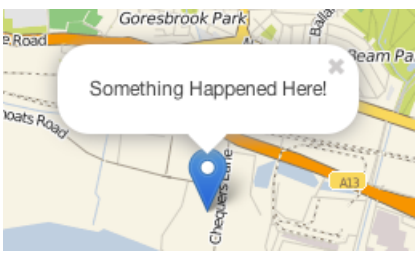

Figure 13.2. Things Happen

This issue led me to another faux paus where I am not binding the map markers to the Occasion collection. That's just silly, because that kind of binding is the WHOLE REASON to use something like Backbone. It was easy enough to fix, because Backbone is the bees knees (which, I think, is a good thing.)

In a nutshell, I bound the 'add' and 'all' events of the occasionCollection to methods on the MapView. These methods then add the new Occasion or regenerate all the markers, as needed. Here they are:

```
App.MapView =  Backbone.View.extend
...
  initialize: (mapProvider) ->
    @mapFactory = mapProvider
    @collection = window.occasionCollection
    @collection.bind("add", @addOccasion, this)
    @collection.bind("all", @drawOccasions, this)
  drawOccasions: () ->
    self = this
    @mapFactory.removeAllMarkers(@map)
    window.occasionCollection.each((occ)->
      self.mapFactory.addOccasion(self.map,occ)
    )
  addOccasion: (occ) ->
    @mapFactory.addOccasion(@map, occ)
...
```

There, now when an Occasion is born (they're so cute when they're new…) the map will add a marker. The 'all' method covers an Occasion being deleted.

The more astute among you realize that, now, adding an Occasion leaves two markers in the new spot. So, along with hiding the popup after creating an occasion, we need to delete the map marker that we used to show the form. Again, not too bad:

```
App.MapProviders.Leaflet  =
...
  addOccasion: (map,occ) ->
    if not @layerGroup? ❶
      @layerGroup = new L.LayerGroup()
      map.addLayer(@layerGroup)
    ll = new L.LatLng(
      parseFloat(occ.get("latitude")),
      parseFloat(occ.get("longitude"))
    )
    marker = new L.Marker(ll)
    marker.bindPopup(occ.get("note"))
    @layerGroup.addLayer(marker)
  addNewOccasionMarker: (map, e, content) -> ❷
    ll = new L.LatLng(e.latlng.lat,e.latlng.lng)
    @marker = new L.Marker(ll)
    @marker.bindPopup(content)
```

```
    map.addLayer(@marker)
    @marker.openPopup()
    $("#occasion_latitude").val(e.latlng.lat)
    $("#occasion_longitude").val(e.latlng.lng)
    $("#occasion_occurred_at").val(new Date())
  hidePopup: (map)-> ❸
    map.closePopup()
    map.removeLayer(@marker)
  removeAllMarkers: (map) -> ❹
    if @layerGroup?
      @layerGroup.clearLayers()
```

There is a fair amount going on here:

❶ I have added a `layerGroup` property to the provider. A layer group is a Leaflet concept that allows you to group a bunch of layers (or, in this case, markers) together. The LayerGroup object in the Leaflet API has a `clearLayers()` function, and that is what I need when I want to clear out all the markers so I can regenerate them.

❷ In `addNewOccasionMarker()`, I add another property called `marker` and store our "temporary" marker for the form. Now, I can get it back when I want to clear it out.

❸ In `hidePopup()`, I remove the temporary marker after I hide the popup.

❹ `removeAllMarkers()` clears out the layer group, as I previously mentioned.

All in all, it's not terrible, but those last additions really show the design flow in my provider approach. A factory would have been better, and it will be the first refactor.

Basic Occasion Functionality

Loccasions now has all the basic functionality that I envisioned those many months ago. It's not groundbreaking, but it does show some nice technical concepts, and I certainly learned a ton. The last chapter will be a retrospective, where I'll look at where Loccasions could go and how I could have done things better.

I am sure I'll have plenty of content for that one.

Chapter 14

Retrospective

The Loccasions application now allows most of the basic functionality that I wanted to produce. Back in Chapter 4, I set out with the following user stories:

- As an unregistered user, I want to see the home/landing page
- As an administrator, I want to be able to invite users to Loccasions
- As an invited user, I want to be able to create an account
- As a registered user, I want to be able to create Events
- As a registered user, I want to be able to create Occasions
- As a registered user, I want to see Occasions on a map

Figure 14.1. Shuttershock

We did them all, except the second story dealing with inviting users. Since the first round of user stories is complete, I think it's time for a retrospective.

What is a Retrospective?

If you've never been a part of a real retrospective, this won't change that. While there are many definitions of the term in the context of software development, I take "retrospective" to mean a "short or time-boxed meeting where you discuss what went well, what went not so well, what we can do better, and how have our priorities changed." Usually, you have a team made of of product owners, developers, and other stakeholders to help review the latest iteration of your application. In this case, it's just little ol' me and 15 or so articles. In both cases, however, a retrospective is very useful and should be a mandatory part of any development process.

If you are the kind of person that is down with more formal definitions and the like, here's a few to get you going:

- James Shore, from The Art of Agile Development[1]
- Ian Burgess chimes in with a definition here[2]
- Matthew Bussa runs through an example retrospective.[3]

I'd be lying if I said I did more than just randomly pick a few from a quick Google search, but looking through each of these, I saw the common ingredients:

- A cup of How'd We Do? But not too much, it'll ruin the dish.

[1] http://jamesshore.com/Agile-Book/retrospectives.html
[2] http://www.ianburgess.me.uk/en/software-development/agile-retrospective-lessons-learned
[3] http://www.matthewbussa.com/2010/10/agile-retrospective-example.html

- A cup of What Went Wrong?
- A dash of What Went Right?
- A smidgeon of Planning Our Next Iteration.

That's it. If you put in too much of any of these ingredients, it is likely your retrospective will come out burnt, raw, or in some other inedible form. Under no circumstances are any of the following ingredients to be used:

- Blame
- Anger
- Hyperbole ("Since X went right we should use it for everything!")

In the end, the retrospective is about getting better, not assigning blame. We get better by learning from our mistakes and planning ("Plans are worthless, but planning is everything."—Dwight D. Eisenhower) a bit better. Also, if your retrospective is more than 4 hours long, I would seriously question the value of anything beyond that time. Remember, the one time you aren't improving the app is when you're all sitting in a meeting overanalyzing/fighting/etc.

What Went Wrong?

I set out to make this an expose on developing a "real" Rails application, as opposed to the many contrived examples we see on the Web. After my first "iteration", I don't think I fulfilled this goal. First off, it is the rare application that is the brainchild and work of a single developer. Also, I made many decisions on technology, gems, and the like based on shaky rationale. If you remember, I chose MongoDB, basically, because I wanted to play with it. If this were a "real" app, I would have never made a decision like that.

In many cases, I became a bit careless with the code and the examples. This caused some of the articles to be very difficult to follow and me to end up with the dreaded "Well, It Runs on MY Laptop" type of application. It was only through the efforts of some Alert Readers (especially, Nicholas Henry… that guy is a machine) that this issue was mitigated somewhat.

I think I should have selected a deployment platform (which would have been Heroku, because it is how-you-say? SUPERFANTASTIC) early on and continuously deployed the application to it. That may have encouraged some more participation from the community, as well as made sure my technical decisions weren't tromping all over deployment.

In addition, towards the end, I all but forgot about the use cases. In a real application, I'd have constantly been checking the user scenarios to make sure I wasn't deviating from the path. In the end, Loccasions may have turned out to simply be a more complicated and contrived example of a Rails app, rather than the real app that I initially had in my sights.

What Went Right?

With the negative out of the way, I think a lot of things went well with Loccasions. First, and most important, I learned a ton and had a good time writing this book. As developers, when we are enjoying our craft and we care about the problem we are solving, life is good. Also, I think the choice of Backbone for the client-side code was a perfect choice. The more I use Backbone, the more I like it. It's just enough framework and just enough out of the way, resulting in a great coding experience.

I echo that sentiment about Jasmine. I have been hooked on Jasmine since I first found it, and it continues to impress. Again, Jasmine focuses on the bits that I don't want to worry about, so I can just test my application. Furthermore, I think Jasmine encourages good JavaScript design, exactly like a testing framework should.

Lastly on the client, I really like LeafletJS. If you need a map in your web application, I can't recommend Leaflet highly enough. Beautiful maps that are a pleasure to code against make it my current "slippy" map of choice.

On the server, choosing Devise was easy and has proven to be a great choice. Quite often, gems with the popularity of Devise can morph into Sasquatchian mounds of unusable code, but Devise doesn't feel that way to me.

Foreman, is almost an imperative now, in my opinion. Especially if you are deploying to Heroku, Forman just makes life easier.

Finally, I thought the pair programming episode with Evan Light in Chapter 8 was, arguably, the highlight. In my new job, I work with Evan quite a bit, and he is every bit the Ruby brain and all-round good chap that I met writing that article. I wish I had done more of this.

How to Get Better?

The easiest, and likely most effective way, to improve Loccasions is to involve other folks. By adding another developer or two, this application would improve tenfold. One of the mantras I live by is the importance of developer collaboration. My best work has never been done solo.

Another group of people whose involvement would massively improve Loccasions are users. Getting people to actually use the application and give feedback would drive the direction of the app, and of this retrospective. In my many years of developing custom applications, I've watched our industry/community evolve from treating users as necessary evils to appropriately putting them in the driver's seat. I remember reading a blog post (the author of which I cannot find … sorry!), which stated that (paraphrasing) "You can always tell when a developer has designed your application." This is certainly true of Loccasions—it could use a Designer's Touch.

What's the Plan?

Whenever I start the next iteration of Loccasions, there will be no shortage of things to do and improvements to make. However, I need to plan where I would focus next. After careful consideration, I think I would focus on:

- deployment
- UX/Design

These are broad terms, obviously. Drilling down a bit, for deployment I would push the app out to Heroku. As a part of it, I think looking into some kind of Continuous Integration server would also help iron out deployment and development scenarios. Investigating CI Servers is also on the immediate planning list.

For the UX/Design tasks, I think recruiting a couple of users and a bonafide web designer would be in the cards. If I could get some users to run through the applic-

ation, give feedback, and get a designer to give the application a once over, Loccasions would be exponentially better.

Of course, there are specific tasks that go with each of these higher level planning items, but we'll stop here. Loccasions was a lot of fun to write, and to write about. It may not be a seminal Rails work, but it covered a lot of topics and answered a lot of questions. Thanks for reading, those of you that stuck with it. Maybe I'll pick it up again in a few months.

Hey ...

Thanks for buying this book. We really appreciate your support!

We'd like to think that you're now a "Friend of SitePoint," and so would like to invite you to our special "Friends of SitePoint" page.

Here you can SAVE up to 43% on a range of other super-cool SitePoint products.

Save over 40% with this link:

Link: sitepoint.com/friends

Password: friends