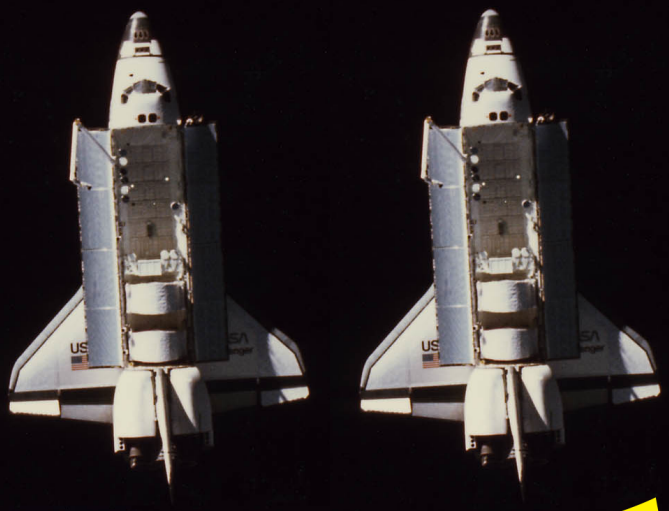

MASTER SPACE AND TIME WITH JAVASCRIPT

BOOK 2: OBJECTS IN JAVASCRIPT

Noel Rappin

a noelrappin.com publication

Master Space and Time With JavaScript

Book 2: Objects in JavaScript

By Noel Rappin
http://www.noelrappin.com

Release 004 October, 2012

The original image used as the basis of the cover is described at http://commons.wikimedia.org/wiki/File:Challenger1983.jpg as a scene of the Space Shuttle Challenger taken during STS-6 in 1983 with a 70mm camera onboard the shuttle pallet satellite. According to NASA policy, the photo is in the public domain.

Contents

Chapter 1

Welcome to *Master Space and Time With JavaScript*

Thanks for purchasing (hopefully) or otherwise acquiring (I won't tell, but it'd be nice if you paid...) *Master Space and Time With JavaScript*.

Here are a few things I'd like for you to know:

Section 1.1

What have I purchased?

Master Space and Time With JavaScript is a book in four parts. All four parts are available at http://www.noelrappin.com/mstwjs.

The first part was *Part 1: The Basics*, which is available for free. It contains an introduction to Jasmine testing and jQuery, plus a look at JavaScript's object model.

You are reading *Book 2: Objects in JavaScript*. It contains more complete examples of using and testing objects in JavaScript, including communication with a remote server and JSON.

Book 3: Backbone, is currently available in beta for $7. The beta currently consists of about half of the finished book. It continues building the website using Backbone.js to create single-page interfaces for some more complex user interaction.

Book 4: Ember, will probably be available by the end of 2012, also for $7. It will build out completely different parts of the website using Ember.js to create complex client-side interactions.

You may pre-purchase all four parts of the book for $15, a $6 savings over buying all three parts separately. This deal may not be available indefinitely.

This book has no digital rights management or copy protection. As far as I am concerned, you have much the same rights with this file as you would with a physical book. I would appreciate if you would support this book by keeping the files away from public file-sharing servers. If you do wish to distribute this book throughout your organization, it would be great if you would purchase additional copies or contact me at `noel@noelrappin.com` to work out a way for you to purchase a site license.

Section 1.2

Who Are You? Who? Who?

Inevitably, when writing a book like this, I need to make some assumptions about you. In addition to being smart, and obviously possessing great taste in self-published technical books, you already know some things, and you probably are hoping to learn some other things from this book.

In some ways, intermediate level books are the hardest to write. In a beginner book, you can assume the reader knows nothing, in an advanced hyper-specific book, you don't really care what the reader knows, they've probably self-selected just by needing the book. In an intermediate book, though, you are potentially dealing with a wider range of reader knowledge.

Here's a rundown of what I think you know:

JavaScript: I'm assuming that you have a basic familiarity with what JavaScript looks like. In other words, we're not going to explain what an `if` statement is or what a string is. Ideally, you're like I was a several months before I started this project – you've dealt with JavaScript when you had to, then had your mind blown by what somebody who really knew what they were doing could do. You specifically do not need any knowledge of JavaScript's object or prototype model, we'll talk about that.

Server Stuff: Since this is a JavaScript book, the overwhelming majority of the topics are on the client side and have nothing to do with any specific server-side tool. That said, there is a sample application that we'll be working on, and that application happens to use Ruby on Rails. You don't actually need to know anything about Rails to run the JavaScript examples, though if you are a Rails programmer, there will be one or two extras. It will be helpful if you are good enough at a command prompt to install the sample application per the instructions later in this preface.

CoffeeScript: I'm not assuming any knowledge of CoffeeScript. If you happen to have some, and want to follow along with the examples using CoffeeScript, have at it, I'll provide a separate CoffeeScript version of the code repository. NOTE: This isn't ready yet, don't go looking for it.

Testing Tools: I'm not assuming any knowledge of testing tools or of any test-first testing process. We'll cover all of that.

jQuery: I'm not assuming any prior jQuery knowledge.

Backbone.js: I'm not assuming any prior Backbone.js knowledge.

Ember.js: I'm not assuming any prior Ember.js knowlege.

Section 1.3

What to Expect When You Are Reading

On the flip side, it's fair of you to have certain expectations and assumptions about me and about this book. Here are a couple of things to keep in mind:

- I firmly and passionately believe in the effectiveness of Behavior-Driven Development as a way of writing great code, especially in a dynamic language like JavaScript. As a result, we're going to write tests for as much of the functionality in this book as is possible, and we're going to write the tests first, before we write the code. If you are completely unfamiliar with testing, this may be a challenge in the early going, as we're juggling Jasmine and jQuery. Don't worry, you can do it, and the rewards will be high.
- This book is focused on the current versions of the languages and libraries available. As I write this, that means ECMA Script 5 for JavaScript, jQuery 1.7.2, Jasmine 1.2.0, Backbone.js 0.9.2, Ember.js 0.9.8.1 and Rails 3.2.x. Keeping up with current versions is hard enough, without worrying about the interactions among multiple versions.

Section 1.4

But is it finished?

This book is incomplete. You will be notified from time to time that a new version of the book is available.

Here's a partial list of things that still need to be done in Books 1, 2, and 3:

- CoffeeScript versions of all the sample code in the book will be made available.
- Formatting for Kindle and ePub versions is still a little wonky in spots. I'm working on it.
- The directions for setting up the sample application probably need to be improved.
- Book 3 is only half complete – there's another extended example that's coming.
- Something only you know – if you think there's something missing in the book let me know via any of the mechanisms listed below.

Section 1.5

What if I want to talk about this book?

Please do! The only way this book will be distributed widely is if people who find something useful in it tell their friends and colleagues.

You can reach me with email comments about the book at `noel@noelrappin.com`. Or you can reach me on twitter as `@noelrap`. If you want to talk about the book on Twitter, it'd be great if you use the hashtag `#mstwjs`, which gives me a good chance of seeing your comment.

This book has mistakes, I just don't know what they are yet. If you happen to find an error in the book that needs correcting, please use the email address `errata@noelrappin.com` to let me know.

There's also a discussion forum for the book at `http://www.noelrappin.com/mstwjs-forum/`. You do need to sign up in order to post, which you can do at `http://www.noelrappin.com/register-for-book-forum/`

Section 1.6

Following Along

The source code for this book is at https://github.com/noelrappin/mstwjs_code. The server side part of the code of this is a Rails application. You won't need to understand any of the Rails code to work through the examples in the book, but you will need to make the application run. Also, a basic knowledge of the Git source control application will help you view the source code.

I've tried to make this simple. The external prerequisites to run the code are Ruby 1.9 and MySQL. RailsInstaller http://www.railsinstaller.org is an easy way to get the Ruby prerequisites for this application installed if you do not already have them. MySQL can be installed via your system's package manager or from http://www.mysql.com.

Once you have the prerequisites installed and the repository copied, you can set everything up for the system by going to the new directory and entering the command `rake mstwjs:setup`. This command will install bundler, load all the Ruby Gems needed for the application, and set up databases. Then you can run the server with the command `rails server`, and the application should be visible at `http://localhost:3000`. Please contact noel@noelrappin.com if you have configuration issues with the setup, and we'll try to work through them.

The git repository for this application has separate branches for each section of the book with source code. Code samples that come from the repository are captioned with the filename and branch they were retrieved from. In order to view the branch, you need to run `git checkout -b <BRANCH> origin/<BRANCH>` from the command line.

Okay, let's get on with it.

Chapter 2

You AutoComplete Me

Section 2.1

Our story so far

In Book 1, you were contracted by the mysterious Dr. What to add JavaScript to a web site called *Time Travel Adventures*, which purports to be a travel agency for time travel vacations. In response to a request from your new client, you created the world's most elaborate show/hide toggle, and learned about Jasmine, jQuery, and JavaScript objects along the way. Now it's time to get a little more elaborate. The code can be found at https://github.com/noelrappin/mstwjs_code.

You've finished the fully modular version of the toggle switch and now you are looking for a real challenge. Luckily, one finds you.

> Dear Mr. and Mrs. Programmer and All the Ships At Sea,
>
> I'd like you to take a look at the user preferences page, and give me a nice way for the user to see what preferences they've chosen. I want something like a combo box, but that autocompletes, and also shows a separate list of the user's current choices. I know you can do it because I've already seen it in the future. So get going.
>
> Your Obedient Humble Servent,
>
> Dr. What

Rather than have the UI for this be a long series of checkboxes or multi-select lists, we'll build a widget where the user types profile information into a text field. We'll use JavaScript to maintain a running list of all the items the user has entered and give a place in the form to remove items. We'll build the management features in JavaScript, and eventually, we'll use the jQuery UI autocomplete widget to make the text entry easier.

Along the way, we'll take a look at how jQuery manages events. We'll use what we learned about JavaScript objects and functions to help us package our JavaScript widget as a standalone piece of code that is easy to insert into an existing form.

Section 2.2

Starting our widget

Let's take a couple of seconds up front to figure out enough of how we want this tool to behave so that we can start writing Jasmine specs. To be clear, we don't want to specify the JavaScript implementation in any detail at this point, but we do need to understand how it might express itself on the HTML page so that we have some lever for describing its behavior. The way I see it, this autocomplete multi-selector has the following elements:[1]

- It is instantiated inside a form and manages only the values of the selected items to be submitted with the form. In other words, no Ajax, at least not yet.
- The main user interface segments are a jQuery UI autocomplete field and a list of the selected items. There's a hidden field that maintains the current list of items for form submission.
- An add button adds the item to the selected list.
- Each selected item has a button to delete it from the list.

We're going to build this widget as a self-contained entity with as little interaction with the outside world as possible. In particular, we're going to treat the DOM tree as just another external dependency, and try to make the interaction between the page and our autocomplete widget as constrained as we can. By doing so, we minimize the amount of structure that needs to be pre-declared in the page, reduce the possibility of duplicating logic between the page and our JavaScript, and make our widget more robust against changes in the page. As an added bonus, having minimal dependencies will make our code easier to test.

We're going to assume the use of a single element with a known DOM ID specified when we create the widget to indicate where we should put our additional fields. We'll create the internal form fields by creating a new autocomplete widget at jQuery starting time. We'll need

1. I'm indebted to my colleague Dave Giunta for improving the quality of the JavaScript and CSS from my original example

a field name for the hidden field and autocomplete field. The widget will also need an initial set of selected values, plus a mechanism for determining the universe of potential values to autocomplete.

All our autocomplete tests will share a common setup. This gives us a chance to design the API we want for creating this widget under the guise of writing our tests.

Filename: spec/javascripts/autocompleteSpec.js (Branch: section_6_1)

```
describe("Autocomplete widget", function() {
  beforeEach(function() {
    affix("form div#autodiv");
    var autocompleteData = {1: "Alpha", 2: "Beta", 3: "Gamma", 4: "Delta"};
    initializeAutocompleteSelector({
        parentSelector: "#autodiv",
        field: "[user][activity_ids]",
        initialValue: "1,3",
        dataUniverse: autocompleteData
    });
  });

// Our tests will go here

});
```

Sample 2-2-1: Setup for the autocomplete widget tests

The setup is doing two things. First off, we're using the `affix` method from Justin Searls' `jasmine-fixture` library (https://github.com/searls/jasmine-fixture). The `affix` method is like jQuery in reverse, we give it a selector, and it creates the elements, in the correct relationship and inserts them to the test runner DOM tree. The `jasmine-fixture` library also takes care of cleaning up the DOM between tests. It's a clean way to set up simple fixture data, and since we want our interaction with the actual DOM to be simple, that's perfect for us. We're creating a `form` tag that has a `div` with a DOM id of `autodiv`, and that's all we need.

With the fixture in place, we'll actually call our setup method `initAutocompleteSelector` with some options. We give it a selector for the parent element to insert the autocomplete widget inside, and a field name to use for the hidden field (note that we're using the Rails naming convention where square brackets get converted to hashes on the server). We're also setting an initial set of ID values, and a universe of data to be auto completed. We're saying that there

are four key/value pairs for ID and value to be selected, and that the selector will be initialized to have two of those values already selected.

The autocomplete widget will set up its own form fields inside that parent `div`, then define the behavior that relates those fields together. Our first set of tests covers just the creation of the hidden field, which will hold the selected ids to be sent with the form submission.

Filename: spec/javascripts/autocompleteSpec.js (Branch: section_6_1)

```
describe("sets up expected hidden element", function() {
  beforeEach(function() {
    this.hiddenField = $("#autodiv #user_activity_ids");
  });

  it("gives the hidden field the correct type", function() {
    expect(this.hiddenField).toHaveAttr("type", "hidden");
  });

  it("gives the hidden field the correct name", function() {
    expect(this.hiddenField).toHaveAttr("name", "[user][activity_ids]");
  });

  it("gives the hidden field the correct value", function() {
    expect(this.hiddenField).toHaveAttr("value", "1,3");
  });
});
```

Sample 2-2-2: Hidden field creation specs

We're searching for a hidden field based on the DOM id of the `div` tag and the new hidden field based on the data that we send to the initializer. We're then verifying that the new field is, in fact, hidden, and has the name and value that we want.

The next step of the process is the simplest thing that could possibly work, and this is pretty simple. I'm not in any way suggesting that this will be the final implementation, but it does pass the test, and we'll clean up the code as we bring in the other features.

Filename: app/assets/javascripts/autocompleteSelector.js (Branch: section_6_1)

```
var initializeAutocompleteSelector = function(options) {
  var $parent = $(options.parentSelector);
  var field = options.field;
  var id = field.replace("][", "_").replace("[", "").replace("]", "");
  var input = $("<input type='hidden'/>")
      .attr({"id": id, "name": field})
      .val(options.initialValue);
  $parent.append(input);
};
```

Sample 2-2-3: Hidden field creation code

The first thing we do is convert the form name from the incoming value, `[user][activity_ids]`, to a DOM id by using some string manipulation. The resulting id is `user_activity_ids`. Note that the brackets in the form name is specifically a Ruby on Rails naming convention, and that other server frameworks might want different form names, and therefore need different ways of generating DOM ids.

Eventually, we use the third variant of the `jQuery` or `$` function. If you call `$` with a string that looks like it contains an HTML tag, then jQuery builds a jQuery object with one element, namely that HTML tag. In this case, we're building the static parts of the hidden field tag as a piece of HTML. Once we have the jQuery object, we use jQuery DOM manipulation to add the `id`, `name`, and `value` attributes. It's far cleaner to build the dynamic parts using jQuery then to try to build the string up in plain JavaScript. The earlier lines are just getting the data in place, extracting the field name from the `div` element's `data` field, and using some string manipulation to convert it to a DOM id. In this case, we're passing all the attributes in one call to `attr`, although we could do it as two different lines. The `type` attribute needs to be part of the actual tag, however, because jQuery does not allow you to change the type of an input or button tag programmatically, because certain browsers (*cough* IE *cough*) don't allow it.

In the actual browser, you'd use this widget by initializing it inside a jQuery function, in much the same way it is initialized in the Jasmine test:

```
var activityData = {1: "Hiking", 2: "Changing History",
    3: "Baseball", 4: "Programming"};
$(function() {
  initializeAutocompleteSelectors({
      parentSelector: "#autodiv",
      field: "[user][activity_ids]",
```

```
      dataUniverse: activityData,
      initialValue: ""
  });
});
```

Sample 2-2-4:

Even when you are building code via TDD, it's a good idea to sanity check in the browser that your test passing code does what you expect in that environment.

Section 2.3

Adding The Input Object

With the specs passing, the next step is the text element and add button which allows the user to select an item. Here are the specs.

Filename: spec/javascripts/autocompleteSpec.js (Branch: section_6_2)

```
describe("sets up expected input element", function() {
  beforeEach(function() {
    this.inputElement = $("#autodiv #user_activity_ids_autocomplete");
  });

  it("gives the input element the proper type", function() {
    expect(this.inputElement).toHaveAttr("type", "text");
  });

  it("gives the input element the proper value", function() {
    expect(this.inputElement).toHaveAttr("value", "");
  });

  it("gives the input element an add button", function() {
    var addbutton = $("#autodiv a#user_activity_ids_add_button");
    expect(addbutton).toHaveClass("selector_add_button");
  });
});
```

Sample 2-3-1: Text field creation specs

These specs are similar to the first set. For the moment, we're handwaving over whatever will make the text input behave like an autocomplete, but we'll get to that eventually.

Here's some code that makes the test pass. Note that we've kept the hidden field code in its own method and written another method to manage text input. We've renamed the method for the hidden text, and made a new `initializeAutocompleteSelector` method.

Filename: app/assets/javascripts/autocompleteSelector.js (Branch: section_6_2)

```
var initializeTextInput = function(options) {
  var $parent = $(options.parentSelector);
  var field = options.field;
  var id = field.replace("][", "_").replace("[", "").replace("]", "");
  var input = $("<input type='text'/>")
      .attr({"id": id + "_autocomplete",
          "name": field + "[autocomplete]"});
  $parent.append(input);
  var add_button = $("<a href='#'>")
      .attr("id", id + "_add_button")
      .html("Add")
      .addClass('selector_add_button');
  $parent.append(add_button);
}

var initializeAutocompleteSelector = function(options) {
  initializeHiddenField(options);
  initializeTextInput(options);
}
```

Sample 2-3-2: JavaScript for the text

Section 2.4

Making the Widget an Object

The guts of the text input method are very similar to the hidden field method. Suspiciously similar. Almost as though somebody was deliberately not cleaning up the code to make a point. Anyway, there's a lot of duplication, which indicates that more refactoring is in order.

I'm going to skip all the intermediate refactoring steps and present you with a modularized version of the autocomplete widget.

Filename: app/assets/javascripts/autocompleteSelector.js (Branch: section_6_3)

```
var AutocompleteSelector = function() {

  var Constructor = function(options) {
    this.domParent = options.parentSelector;
    this.options = options;
    this.field = options.field;
    this.universe = options.dataUniverse;
    $(this.domParent).append(this.hiddenField())
        .append(this.textInput())
        .append(this.addButton());
  }

  Constructor.prototype = {
    determineId: function(suffix) {
      var id = this.field.replace("][", "_").replace("[", "").replace("]", "");
      if(suffix) {
        id = id + "_" + suffix;
      }
      return id;
    },

    initialValue: function() {
      return this.options.initialValue;
    },

    hiddenField: function() {
      return $("<input type='hidden'/>")
          .attr("id", this.determineId())
          .attr("name", this.field)
          .val(this.initialValue());
    },

    textInput: function() {
      return $("<input type='text'/>")
```

```
          .attr("id", this.determineId("autocomplete"))
          .attr("name", this.field + "[autocomplete]");
    },

    addButton: function() {
      return $("<a href='#'>")
          .attr("id", this.determineId("add_button"))
          .html("Add")
          .addClass('selector_add_button');
    }
  };

  return Constructor;
}();

var initializeAutocompleteSelector = function(options) {
  new AutocompleteSelector(options);
}
```

Sample 2-4-1: Now wigitized

Okay, that's a lot of code. As I hope is clear, we've sort of smashed together the module pattern with the autocomplete functionality. Here are a couple of notes on what's going on there:

- The `initializeAutocompleteSelectors` function now just defers to the creator of our module pattern object. The argument to the constructor is the set of options we pass in to control the starting state of the widget.
- Inside the constructor for the class, we're setting a couple of properties of the new object, and then appending our hidden field, text input, and add button.
- Since we've pulled a couple of common values like `determineId` and `initialValue` into their own helper methods, we can limit the methods like `hiddenField` to just the creation of the DOM element in question. We've also changed from methods that have a side effect of appending the DOM element, to methods that return a value and allow the appending to take place in the constructor. Keeping side effects out of methods will make the code more manageable over the long haul.

Section 2.5

Adding Display of values

There's one more piece of functionality, which is to add a display of the already selected values. Since we basically know the drill by now, we can go through the test quickly:

Filename: spec/javascripts/autocompleteSpec.js (Branch: section_6_4)

```
describe("sets up a list of known values", function() {
  beforeEach(function() {
    this.ul = $("#autodiv #user_activity_ids_list");
  });

  it("sets up expected elements", function() {
    expect(this.ul.find("#user_activity_ids_element_1")).toHaveText("Alpha Delete");
    expect(this.ul.find("#user_activity_ids_element_1 .delete-button")).toExist();
    expect(this.ul.find("#user_activity_ids_element_3")).toHaveText("Gamma Delete");
    expect(this.ul.find("#user_activity_ids_element_3 .delete-button")).toExist();
  });

  it("does not set up missing elements", function() {
    expect(this.ul.find("#user_activity_ids_element_2")).not.toExist();
  });
});
```

Sample 2-5-1: Display of value specs

We are verifying that each element in the initial value data attribute of the HTML fixture results in a list element with its very own delete button. We're also verifying that the global value that is not in the initial value attribute doesn't get a list element. Negative tests like this can be tricky – it's easy to get a false positive if you have a typo – but testing both sides of existence/non-existence logic makes the test more robust.

Making the test pass requires some changes to the constructor and a new method. Constructor first:

Filename: app/assets/javascripts/autocompleteSelector.js (Branch: section_6_4)

```
var Constructor = function(options) {
  this.domParent = options.parentSelector;
  this.options = options;
  this.field = options.field;
  this.universe = options.dataUniverse;
  $(this.domParent).append(this.hiddenField())
      .append(this.textInput())
      .append(this.addButton())
      .append(this.valueList());
};
```

Sample 2-5-2: I've added one method call to the constructor

The constructor wraps up by calling a new method to create the list of values.

Filename: app/assets/javascripts/autocompleteSelector.js (Branch: section_6_4)

```
  valueList: function() {
    var $ul = $("<ul>").attr("id", this.determineId("list"));
    var that = this;
    $.each(this.initialValue().split(","), function(index, value) {
      var $li = $("<li>").attr("id", that.determineId("element_" + value))
          .text(that.universe[value]);
      var $a = $("<a href='#'>").addClass("delete-button")
          .attr("id", that.determineId("delete_" + value))
          .text(" Delete");
      $li.append($a);
      $ul.append($li);
    });
    return $ul;
  }
```

Sample 2-5-3: The method that creates the list of values

The part of this method that we haven't seen is the `$.each` call. The `$.each` is one of many pure utility methods defined as properties of the `$` object. The `$.each` method takes a list argument and a function argument and applies the function to each element of the list in turn. In this case, creating a list element and delete button for each value already in the selected list.

You probably noticed the weirdness of `that = this`, followed by the references like `that.determineId("element_" + value)`. These unusual references are workarounds for the way JavaScript scope behaves. Inside the callback function defined as the second argument to `$.each`, the `this` variable is set to the value of the element going through the function. (In other words, basically the same as the `value` argument to the interior function.)

However, since the `$.each` call is inside a larger object, JavaScript's behavior here shadows what you would normally expect to be `this`, namely the instance itself. The awkward and somewhat ridiculous workaround is to give the instance an alternate name – I've chosen `that`. Anyway, the `that` name is not shadowed inside the `$.each` function, so it's still accessible for accessing the instance variables and methods of the larger instance. CoffeeScript fans will note that you can get the same functionality without the awkward renaming step by defining the function argument of the `each` method using CoffeeScript's fat arrow, `=>`. You can also achieve this feature with the `bind` method found in underscore.js.

At this point, we've added all the HTML we need to have this selector work. The next step is to wire up events to the needed functionality.

Section 2.6

Adding Events to the Widget

We have two events we need to take care of. Pressing the add button adds the value of the text field to the list of items, while clicking the delete links removes the value. In both cases, we also need to manage the hidden field that the server side submits and uses. (Obviously the server side needs to use that hidden field appropriately, but that is outside our scope here.)

Let's start with the add button. Here's a Jasmine test suite for the basic functionality.

Filename: spec/javascripts/autocompleteSpec.js (Branch: section_6_5)

```
describe("adds an element when clicked", function() {

  beforeEach(function() {
    this.inputElement = $("#autodiv #user_activity_ids_autocomplete");
    this.addButton = $("#autodiv a#user_activity_ids_add_button");
    this.inputElement.val('Beta');
    this.addButton.click();
```

```
    this.ul = $("#autodiv #user_activity_ids_list");
  });

  it("increases the size of the list", function() {
    expect($("#autodiv ul li").size()).toEqual(3);
  });

  it("gives the new element the expected text", function() {
    expect(this.ul.find("#user_activity_ids_element_2")).toHaveText("Beta Delete");
  });

  it("gives the new element a delete button", function() {
    expect(this.ul.find("#user_activity_ids_element_2 .delete-button")).toExist();
  });

  it("updates the value of the hidden field", function() {
    var hiddenField = $("#autodiv #user_activity_ids");
    expect(hiddenField).toHaveAttr("value", "1,3,2");
  });

});
```

Sample 2-6-1: Add button event specs

This test is mostly cobbled together from bits and pieces we've already seen. In particular, we're reusing element definitions for the input box, add button, list of items, and hidden field. (This means we might want to refactor the tests in the future so that these common definitions are themselves functions.) We call the `click` method, to trigger a click event on the given element. After the click, we're checking that a third item is added to the list of elements, that the item has the expected text, `Beta`, and that the id associated with that element has been added to the comma-delimited ID list being handled in the hidden field.

To make this code pass, we need to understand how jQuery manages events. The bedrock method for making something happen in response to an event in jQuery is called `on`, and is a method of the jQuery object in the same way that, say, `html` is. Which is to say that the `on` method is called against a jQuery object that matches one or more DOM elements on the page. The simplest version of `on` takes two arguments, an event type and a handler. For instance, we might bind the add button to some code using a call like this one:

```
$("a#user_activity_ids_add_button").on("click", function(event) {
  console.log("The add button " + $(this).attr("id") + " was pressed");
});
```

Sample 2-6-2:

What we have here is a short binding between a selector that represents the add button and a function that will be called when the add button is clicked. The first argument to `on`, in this case `click`, determines which event will be responded to.

Two items to notice in passing: the callback function takes one argument, which is the JavaScript event object that was created for this event. The event object is often not explicitly included in the argument list for the callback function if it is not used. One reason why the event object might not be used is that jQuery has already taken care of the most common reason to use it, which is to get at the DOM element being "event-ed," so to speak. In jQuery, inside an event handler, the `this` variable points to that element, and, as in the example, is often wrapped in the jQuery function, `$(this)`, to allow jQuery methods to be sent to it.

jQuery provides a shortcut function for all the standard JavaScript events, `click`, `change`, `keydown`, `submit`, `select`, and all the rest. These shortcuts can be used in place of `on` where they apply, so the example would often be invoked like this:

```
$("a#user_activity_ids_add_button").click(function() {
  console.log("The add button " + $(this).attr("id") + " was pressed");
});
```

Sample 2-6-3:

That said, I prefer to include the `event` argument explicitly because it gives a clear visual cue that the function is an event handler. And you never know when you are going to need the event object.

Under normal circumstances, jQuery events walk through the entire DOM tree, meaning that every DOM element that contains the point where the event took place, all the way up to the document object itself is checked to see if it has a handler that matches the event. The closest element to the event is invoked first, and within each element, handlers are invoked in the order in which they were declared. After all handlers have been invoked, the "normal" event response takes place, meaning that things like links being followed or selector lists selecting happen after all your application event handlers.

However, if an event handler function returns `false`, that has two different consequences for the event processing. The event propagation up the DOM tree stops, though other handlers for the same element will still be processed, and the default behavior of the element is blocked. A common example is an anchor link that you want to trigger an Ajax call rather than follow the URL to reload the entire page. The `false` return value there prevents the normal link-following behavior.

It's considered a little bit reckless to just return `false` when all you really want to do is block default behavior. Since the stop propagation behavior comes with it, you may inadvertently prevent necessary behavior that is attached to a containing element on the page. Admittedly, having multiple elements respond to the same event is a little rare, especially in simple applications. However, preventing this behavior out of hand can lead to subtle problems later on, since an event handler can prevent other handlers from being invoked without explicitly being aware of their existence.

Luckily, you can get one behavior without the other by using the `event` object directly. The event object has methods defined that are relevant to our interests here, `stopPropagation` and `preventDefault`. Returning `false` effectively bundles both calls, but you can get just one by using the event methods directly.

```
$("a#user_activity_ids_add_button").click(function(event) {
  console.log("The add button " + $(this).attr("id") + " was pressed");
  event.preventDefault();
})
```

Sample 2-6-4:

An issue to be aware of with `stopPropagation` is that the method only stops event handling from being propagated to other DOM elements. It does not stop other handlers on the same element from being invoked. If you want to call a full and complete stop, you use the method `stopImmediatePropagation`. Also, all three of these methods have associated boolean forms, `isDefaultPrevented`, `isPropagationStopped`, and `isImmediatePropagationStopped`, which return `true` if the method in question has been called for that event object.

Section 2.7

Event Handlers and Proxies

Now that we understand something about jQuery events, it's time to write our event handler. Since we're still working inside the `AutocompleteSelector` object, we'll define the event handler as a method of that object, and bind it to the add button in our constructor. Here's our first pass. This code has some problems with it caused by the way jQuery manages events, but we'll correct the flaws in a moment. In the constructor, the last line binds the DOM id for the add button to our handler. Please note that this new line is going to change in a little bit.

Filename: app/assets/javascripts/autocompleteSelector.js (Branch: section_6_5)

```
var Constructor = function(options) {
  this.options = options;
  this.domParent = $(options.parentSelector);
  this.field = options.field;
  this.universe = options.dataUniverse;
  $(this.domParent).append(this.hiddenField())
      .append(this.textInput())
      .append(this.addButton())
      .append(this.valueList());
  $("#" + this.determineId("add_button")).click(this.addEventHandler);
};
```

Sample 2-7-1: Our add event handler in the constructor

Inside the object, we define the handler.

Filename: app/assets/javascripts/autocompleteSelector.js (Branch: section_6_5)

```
  listElement: function(value) {
    var $li = $("<li>").attr("id", this.determineId("element_" + value))
        .text(this.universe[value]);
    var $a = $("<a href='#'>").addClass("delete-button")
        .attr("id", this.determineId("delete_" + value));
    $a.text("Delete");
    $li.append($a);
    $a.before(" ")
```

```
    return $li;
  },

  valueList: function() {
    var $ul = $("<ul>").attr("id", this.determineId("list"));
    var that = this;
    $.each(this.initialValue().split(","), function(index, value) {
      if(value.length > 0) {
        $ul.append(that.listElement(value));
      }
    });
    return $ul;
  },

  idLookup: function(itemName) {
    for(id in this.universe) {
      if(this.universe[id] === itemName) {
        return id;
      }
    }
    return null;
  },

  addEventHandler: function(event) {
    var newItemName = $('#' + this.determineId("autocomplete")).val();
    var newItemId = this.idLookup(newItemName);
    if(!newItemId) {
      return;
    }
    var hiddenField = $('#' + this.determineId());
    hiddenField.val(hiddenField.val() + "," + newItemId);
    var list = $("#" + this.determineId("list"));
    list.append(this.listElement(newItemId));
    var $autocomplete = $('#' + this.determineId("autocomplete"));
    $autocomplete.val("");
    $autocomplete.focus();
    event.preventDefault();
```

```
  },
```

Sample 2-7-2: The actual add event handler

What are we doing here? First off, we retrieve the value from the text input, which is what the user has entered. On the next line, we call a function that does a reverse lookup on our `universe` data to retrieve the object id associated with that name. Next up, we find the hidden field and append the new id to its comma-delimited value string. Then we find the `ul` list of items and add a new list item with the new value. (Behind the scenes we've refactored the relevant lines from the `valueList` method into a new method `listElement` that we can call from here.) Finally, we clear the text field, give it the UI focus, and prevent any default behavior from happening. Nice and neat.

Except that it doesn't quite work. Run this in Jasmine and get:

```
TypeError: Object http://js-travel.dev/jasmine# has no method 'determineId'
```

Sample 2-7-3:

Why doesn't Jasmine like `determineId`, when we've already defined it? It's a similar issue to what faced us when we used `$.each` in creating the value list. Namely, jQuery has redefined `this`, and the redefinition is blocking our ability to get at the JavaScript instance whose method we're looking at. Specifically, in an event handler, jQuery sets `this` to be the DOM element being worked on. Although the exact `that = this` workaround we used before isn't an option here, jQuery allows us to do something similar. But we need to do it when the handler is bound. So, going back to the constructor, note the change in the last line:

Filename: app/assets/javascripts/autocompleteSelector.js (Branch: section_6_6)

```
var Constructor = function(options) {
  this.options = options;
  this.domParent = $(options.parentSelector);
  this.field = options.field;
  this.universe = options.dataUniverse;
  $(this.domParent).append(this.hiddenField())
      .append(this.textInput())
      .append(this.addButton())
      .append(this.valueList());
  $("#" + this.determineId("add_button")).click(
      $.proxy(this.addEventHandler, this));
```

```
};
```

Sample 2-7-4: Our add event handler in the constructor

In the constructor, we've wrapped the event handler using the jQuery utility method `$.proxy`. The `$.proxy` method takes two arguments: a function and a JavaScript object. jQuery will then guarantee that when the first method is called, that the context in which it is called – in other words, the object used as the value of `this` – will be the object passed as the second argument. In our case, the second argument is simply `this` in the constructor's context, meaning that it is the instance in question. Meaning that when the event handler is invoked, `this` in the event handler will now point to the instance, rather than the targeted DOM element.

Section 2.8

Responding to change with delegated events

With that change, the test passes. However, a subtle bug is still hanging around. We'll only be able to see the bug, though after we get the delete button working.

The Jasmine test for the delete button is reasonably similar to the test for the add button, but, you know, with deleting.

Filename: spec/javascripts/autocompleteSpec.js (Branch: section_6_7)

```
describe("deletes an element when clicked", function() {
  beforeEach(function() {
    this.$ul = $("#autodiv #user_activity_ids_list");
    deleteButton = this.$ul.find("#user_activity_ids_element_1 .delete-button");
    deleteButton.click();
  });

  it("expects the list size to decrease", function() {
    expect($("#autodiv ul li").size()).toEqual(1);
  });
```

```
  it("updates the hidden field value", function() {
    var hiddenField = $("#autodiv #user_activity_ids");
    expect(hiddenField).toHaveAttr("value", "3");
  });
});
```

Sample 2-8-1: Add button event specs

This should seem vaguely familiar. We grab the list of items, pick one of the delete buttons to click, and validate that the list item is gone and the hidden field is also missing the id.

The delete handler itself is simpler than the add handler because we don't need to build any elements, but it does have a couple of workarounds for JavaScript's relatively impoverished array manipulation features.

Filename: app/assets/javascripts/autocompleteSelector.js (Branch: section_6_7)

```
  deleteEventHandler: function(event) {
    var idToDelete = $(event.target).attr('id').split("_").pop();
    var hiddenField = $('#' + this.determineId());
    var existingIds = hiddenField.val().split(",");
    var indexToRemove = existingIds.indexOf(idToDelete);
    if(indexToRemove != -1) {
      existingIds.splice(indexToRemove, 1);
      hiddenField.val(existingIds.join(","));
    }
    $(event.target).closest("li").remove();
    event.preventDefault();
  },
```

Sample 2-8-2: The delete event handler

Let's break this down. First, we extract the id of the item we're deleting from the DOM id of the event target. Remember, we have to use `event.target` rather than `this` because we'll be using the `$.proxy` method to ensure that `this` is the object instance. The DOM id will be something like `user_activity_ids_delete_1`, so splitting and then popping the last value gives us `1` – as a string, but that's fine for our purposes.

Next comes the slightly arduous process of extracting the id from the hidden field value. JavaScript doesn't have a direct way to remove an object from an array, as in Ruby's `delete` method, so we scramble. (In a real system, I'd use the excellent underscore.js library). In successive lines of code, we use `indexOf` to determine the index of the id we want to delete in the array. If the item is found, meaning that the index is not `-1`, then we use the `splice` method to remove the element at that index. Note that `splice` mutates the array in place, and the return value is the element that is removed, which is why we aren't using the return value of `splice`. Then we join the spliced array back into a comma-delimited string and reset the value of the hidden field. Finally, we remove the list element from the DOM by using the jQuery method `closest` to walk up the DOM tree until the `li` selector is hit, and we call it a day.

In the constructor, we can attach this event handler to all the delete buttons at once by binding the event to all the items with the DOM class `.delete-button`.

Filename: app/assets/javascripts/autocompleteSelector.js (Branch: section_6_7)

```
var Constructor = function(options) {
  this.options = options;
  this.domParent = $(options.parentSelector);
  this.field = options.field;
  this.universe = options.dataUniverse;
  $(this.domParent).append(this.hiddenField())
      .append(this.textInput())
      .append(this.addButton())
      .append(this.valueList());
  $("#" + this.determineId("add_button")).click(
      $.proxy(this.addEventHandler, this));
  $(".delete-button").click(
      $.proxy(this.deleteEventHandler, this));
};
```

Sample 2-8-3: Constructor with add and delete

Which is all well and good, and the tests pass, except there's still one thing. What's going on here?

Filename: spec/javascripts/autocompleteSpec.js (Branch: section_6_7a)

```
describe("adds an element when clicked, then deletes that element", function() {
  beforeEach(function() {
    inputElement = $("#autodiv #user_activity_ids_autocomplete");
    addbutton = $("#autodiv a#user_activity_ids_add_button");
    inputElement.val('Beta');
    addbutton.click();
    ul = $("#autodiv #user_activity_ids_list");
    deleteButton = ul.find("#user_activity_ids_element_2 .delete-button");
    deleteButton.click();
  });

  it("still has the list size the same", function() {
    expect($("#autodiv ul li").size()).toEqual(2);
  });

  it("has removed the deleted element", function() {
    expect(ul.find("#user_activity_ids_element_2")).not.toExist();
  });

  it("has managed the hidden field correctly", function() {
    var hiddenField = $("#autodiv #user_activity_ids");
    expect(hiddenField).toHaveAttr("value", "1,3");
  });
});
```

Sample 2-8-4: This test adds an element then deletes it

This test adds an element, then attempts to click the new delete button and verify that the newly added element can delete itself. This test fails – the new element is created, but not deleted. If you haven't done much jQuery then it may not occur to you to have this test at first. At least, not until you try this in the browser, at which point the bug becomes pretty clear.

The problem is that the delete event handler is not fired when the newly created delete button is clicked. The reason is deceptively simple – the newly created delete button is not bound to the event handler. When you use the jQuery `on` event handler as we have been using it or the helper functions such as `click` that are based on `on`, the binding applies only to DOM elements that match the selector at the time the `on` method is invoked. The newly created

delete button is newly created, and therefore didn't exist when the object constructor was called and invoked the `click` methods.

Of course, there are ways to work around this problem. We could just bruise our way past it by adding some lines to the `addEventHandler` calling the `click` method on the newly created button and attaching it to the same `deleteEventHandler`. Since we've already made the event handler a separate method, it wouldn't be *that* much duplicated code, right?

The long-term problem with manually associating new items to event handlers is that it's extremely error-prone. As the application gets larger and more dynamic, the chance that some error handler slips through the cracks and is forgotten gets very high – by some made-up estimates, the chance is as high as 115%. Besides, there's a better way.

Essentially, the problem is that jQuery can only bind events to DOM elements that actually exist. What we want, though, is to bind an event to a parent element that already exists, and give the parent element a selector for the future child elements that will respond to the event. When the parent element receives the event, it can do a live search at the time of the event for child objects, thereby enabling events to effectively be bound to elements that may not exist at the time the binding is created. In jQuery, this process is called *delegation*, and events created like this are called *delegated* events.

We can create a delegated event by simply adding another argument to the `on` method after the event with a jQuery selector for subordinate elements. For example, we might use `on` to declare delegated events for our add and delete buttons:

Filename: app/assets/javascripts/autocompleteSelector.js (Branch: section_6_7a)

```
var Constructor = function(options) {
  this.options = options;
  this.domParent = $(options.parentSelector);
  this.field = options.field;
  this.universe = options.dataUniverse;
  $(this.domParent).append(this.hiddenField())
      .append(this.textInput())
      .append(this.addButton())
      .append(this.valueList());
  $(this.domParent).on("click", "#" + this.determineId("add_button"),
      $.proxy(this.addEventHandler, this));
  $(this.domParent).on("click", ".delete-button",
```

```
        $.proxy(this.deleteEventHandler, this));
};
```

Sample 2-8-5: The autocomplete constructor with delegated events

These delegated event handlers will appear to bind dynamically. Once the delegated event is declared, then any element inside the parent that matches the selector is bound to the handler, even if the DOM element is added to the page after the selector is declared. However, while the subordinate object can be created later, the parent object that is the instance that `on` is called against must exist at the time the handler is bound. In an emergency, you can use `$(document)` as the receiver of the method. On the plus side, the document is always there, on the down side, binding event against the whole document could have negative performance implications or bad side effects if the selector binds the event to more elements than you intend.

This particular bit of magic is exactly what we need. With the delete handler declared using a delegated event, the new delete button is connected to `deleteEventHandler` and the test passes.

The exact implementation of delegated events has some implications for behavior. You might imagine that delegated events are implemented by testing every newly added DOM element and binding it to events that match. But, as we implied in our initial description of delegated events, that's not what jQuery does. Instead, jQuery binds the event handler to the receiver of the event, and when a delegated event reaches that element, the element performs a jQuery `find` on the selector that is part of the delegated event. If any DOM elements match the selector, then jQuery invokes the callback function.

Section 2.9

Custom events

Before we leave our discussion of jQuery events, I did want to mention how easy it is to create and manage your own event types. Hint: it's super-easy. To bind an event to a custom event type, all you need to do is include any string that is not already the name of an event and make it the first argument to `on`, as in:

```
$(".listener").on('awesomeness', function() {
  console.log("awesomeness has happened");
}
```

Sample 2-9-1:

A custom event by definition has to be triggered within your application, since it's not part of the existing set of browser events. You trigger the event with the conveniently named `trigger` method, as in:

```
$('#awesome-thing').trigger('awesomeness');
```

Sample 2-9-2:

The `trigger` method takes an optional second argument, which is a list that is unrolled and passed as successive arguments to all the handlers for the event being triggered.

You can use `trigger` to explicitly trigger a known event type, as well, in which case the expected event also happens. If for some reason you want to trigger handlers but not the default behavior, use the `triggerHandler` method instead of `trigger`.

As we'll see, Rails uses custom events to allow you to easily manage parts of the Ajax request life cycle by creating callbacks that respond to events that Rails sends out at various parts of an Ajax request. In our case, we could use custom events to allow an application that uses the selector to augment its behavior without having to change the selector code itself. If we were to add a line like the following to `addEventHandler`:

```
$("#" + this.determineId("add_button")).trigger("item_added",
    [new_item_name, new_item_id]);
```

Sample 2-9-3:

Then other code could augment the item add behavior by doing something like this:

```
$("#user_activity_ids_add_button").on(
    "item_added", function(event, item_name, item_id) {
  // do something else
});
```

Sample 2-9-4:

Notice how we added additional elements to the `trigger` call, so that they could be picked up by the callback function in `on`.

Section 2.10

You Autocompleted Me

At this point, our autocomplete widget works great. Well, except for the actual autocompleting part, which we haven't done yet. Luckily, it's not that hard with the help of jQuery UI, which is a separate library distributed by the jQuery project that provides useful widgets and behaviors on top of jQuery. In addition to the autocomplete widget, jQuery UI also has a date picker, progress bar and slider, along with support for tabbed interfaces, show one detail out of many accordion looks, modal dialogs, and fancier buttons. As if that weren't enough, there's also a bunch of animated effects, support for common event types like drag-and-drop, plus predefined look-and-feel themes. In other words, it's got a lot of useful stuff.

To get jQuery UI, head to `http://jqueryui.com/home`. The download links are in the upper right of the home page (at least, as of today), and we want the link marked `Stable`. As I write this, the stable version number is 1.8.16. Since the jQuery UI has a lot of disparate pieces and a single project might not want all of them, the jQuery UI team also allows you to create a custom download using just the features you need. We're not going to bother with that, just download the stable version.

The stable version is distributed as a zip file, and the zip has several files. For our purposes, two are interesting. The jQuery UI javascript file is in the zip at `js/jquery-ui-1.8.16.custom.min.js`, and there's a stylesheet at `css/jquery-ui-1.8.16.custom.css`. Plus the zip contains a bunch of images that parts of jQuery UI use but that we're not talking about right now. Exactly how you integrate that into your application depends on what you are using.

In a generic application, you'd need to include three jQuery UI files on a page that uses jQuery UI. One is jQuery itself, which you probably already have in your project, but which is included in the jQuery UI download anyway. The second is the jQuery UI file, which will be in the `js` directory of the zip file, named something like `js/query-ui-1.8.21.custom.min.js`. Finally, the `css` directory of the zip file will contain the CSS for the theme you are using, named something like `css/smoothness/jquery-ui-1.8.21.custom.css`. If you pick a different theme, then the subdirectory name will match that theme.

In Rails 3.1 and up, it's now relatively easy to add jQuery UI by including the `jquery-ui-rails` gem, available at https://github.com/joliss/jquery-ui-rails. If you are using the gem, you don't

need to download jQuery UI, the gem will contain it. After including the gem, all you need to do is add `//=require jquery.ui.all` to the `application.js` manifest and `/* require jquery.ui.all */` to the `application.css` manifest. The `readme` file for the gem also has instructions on only including some of the jQuery UI modules.

```
*= require_tree ../../../vendor/assets/stylesheets
```

Sample 2-10-1:

Once the setup is in place, adding the autocomplete feature is easy. Basically it's one line of additional code, and a helper method to convert our universe of data to an array of autocomplete-able values:

Filename: app/assets/javascripts/autocompleteSelector.js (Branch: section_6_8)

```
  universeValues: function() {
    var result = [];
    for(var property in this.universe) {
      result.push(this.universe[property]);
    }
    return result;
  },

  textInput: function() {
    input = $("<input type='text'/>")
        .attr("id", this.determineId("autocomplete"))
        .attr("name", this.field + "[autocomplete]");
    input.autocomplete({source: this.universeValues()});
    return input;
  },
```

Sample 2-10-2: The autocomplete jquery UI widget

The line starting `input.autocomplete` is doing the heavy lifting by invoking the jQuery UI `autocomplete` method, which converts the input field to an autocomplete. The `source` option can take an array of values and use those as the set of items to display in the autocomplete window. In this case, the `universeValues` method loops over our universe data structure and creates an array of all the property values – remember, our `universe` data structure is of the form `{1: "Hiking"}`.

Amazingly, that's it – putting this on the page results in a form element that looks like this.

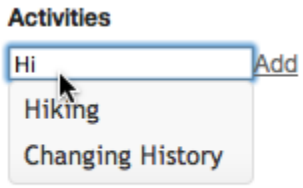

Figure 1: Autocompleting Widget

Section 2.11

Retrospective

We've created a standalone widget that manages autocomplete selection, starting with a basic function, and eventually refactoring it to a reusable object that manages its own state while keeping in sync with the form elements on the page. After we created our object, we learned how jQuery handles events so that the object could respond to user behavior. We learned how to simulate events for testing purposes and also how to bind events to functions that are executed when the event happens.

Chapter 3

Debugging JavaScript and the JavaScript Developer Tools

I'd like to take a short break between longer chapters to go over the tools that are available to each and every front-end browser developer. Bluntly, they are awesome.

One of the most significant advances in the JavaScript toolbox was the advent of Firebug and other JavaScript debuggers. All web developers should thank Joe Hewitt for the initial Firebug implementation. Suddenly, the lights were on and it was possible to inspect and modify the results of browser interactions. Having a working knowledge of what you can do in a JavaScript debugger will go a long way toward making your JavaScript work more effective.

We're going to focus on the WebKit developer tools window, which is available in Chrome and Safari. If you primarily use Firefox, the Firebug extension provides most of the same features. I'm not aware of an Internet Explorer extension that provides the same depth of features. Please note that these tools are under consistent development and at any given time are likely to have slightly different versions in Chrome and Safari. Apple has detailed documentation of the Safari developer tools (http://developer.apple.com/library/safari/navigation/, and Google has a somewhat less detailed version of the tools on Chrome http://code.google.com/chrome/devtools/docs/overview.html.

Brief aside: I think this section is now a little out of date. Safari 6 seems to have changed the layout of the developer tools a little bit, and the Chrome team keeps adding new stuff – you can get the latest Chrome tools with Chrome Canary, which is a near-nightly update to the tools available at https://tools.google.com/dlpage/chromesxs/. Point is, there's an update coming to this part soon.

To see why these tools are so useful, think about all the things you can do in a dynamic web page. You could change text, colors, positions, visibility. However, when you go to view source on the page, that the HTML has not changed.

The changes to the styles and HTML don't show up in the source because they are added to the DOM tree by jQuery after the page loads. Which begs the question, how can we verify that the changes were made? Which brings us to the JavaScript developer tool window.

Section 3.1

Heading over to the JavaScript developer tool window

In Chrome, you access the developer tools from the Monkey Wrench menu, the tools submenu and Developer tools. In Safari, you need to turn the Developer Menu on in the advanced preferences, and then select "Show Web Inspector" from the Develop menu. In both browsers, you can also right-click on any element, and select `Inspect Element` from the pop-up menu. That will give you the web inspector with the object you clicked on already in focus. If you open the Time Travel adventures page in Safari or Chrome, and you right click the "More about time travel" link, then you will see an inspector window that is something like this – depending on your settings, the inspector may be a standalone window or it may wind up as the bottom of your regular browser window.

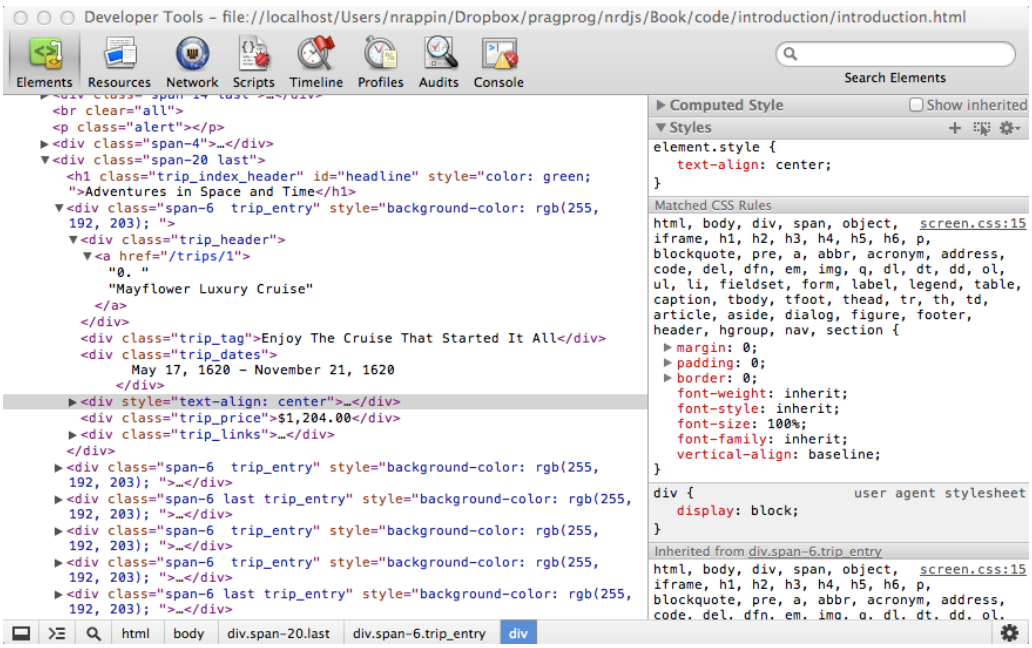

Figure 2: The WebKit Inspector

You'll notice that the left panel of this pane shows the current headline text, the background color for the `trip_entry` element, and the `0.` counter. In this snapshot, the inspector is in "Elements" mode. For our JavaScript debugging-purposes, that's the mode you'll spend the most time in – not that the other modes don't have their charms.

In this mode the big panel on the left is the HTML source of the page being inspected. There are at least three wonderful things about this panel. The first thing you might notice is that if the inspector window is active, then as you run your mouse over the content pane, the DOM element whose content you are mousing over will be highlighted on the actual browser page with a little tool tip giving the CSS selector of the element and its size in pixels. Very handy for determining the exact extent of a container.

The panel is also live: as the DOM changes in response to user actions and JavaScript, the content pane will reflect those changes. Finally, it's editable, you can go into any part of the DOM, HTML text, HTML attribute, class name, what have you, change it and have the change show up on the page. Selecting an element and right-clicking on it brings a popup menu that allows for some further editing options, including the ability to edit the entire tag at once and copy the entire tag to the clipboard. In Chrome, you can also make a selection that triggers a breakpoint if the element is changed. Among other things, this is an excellent way to experiment with new styles, and also to try to get exact sizes and placements correct.

When you select an element in the left side content panel, the right side style panel changes to display information about that element's styles. The sub-pane displayed in the figure is "Styles," and it shows the CSS selector and styles for every CSS rule known to the page that matches the selected element. Styles that are overridden by more specific rules are shown, but with a strikethrough.

The line in the CSS file where the rule is defined is also shown, clicking on that note takes you to a listing of that file inside the web inspector. This pane is also editable, and any change will be displayed in the browser. Also individual styles that are part of rules can be unchecked to temporarily turn them off. The "Computed Style" pane above shows a list of all styles that apply to the selected element. Below, the inspector has "Metrics," which displays a schematic box diagram with margin, border, and padding displayed. Clicking on any part of the box allows you to live edit the border, margin, or padding of the element. Other panels show the JavaScript properties of the selected element, and any event listeners defined for it.

The status bar at the bottom has three buttons followed by a breadcrumb trail. The breadcrumb trail is the entire ancestor list of the element currently selected in the left pane,

containing the CSS class and DOM ID of each element all the way back up to the `html` tag at the top. The buttons... let me do them out of order to leave the most important for last. The leftmost button, the one that looks like a half-filled square, toggles the inspector from being a separate window to being the bottom half of the browser window for the page that it is attached to. The rightmost button, which looks like a magnifying glass or a slanted lollypop, when clicked turns blue, and in that state any element you click on in the main browser will automatically be selected in the inspector window.

The middle button, which looks like an arrow and three lines or possibly some sort of emoticon for a puppy (no, really, look at it sideways, arrow-side down...), opens the console. You can also access the console from the top-level button, helpfully labeled "Console." The only difference between the top button and the bottom one is that the top button gives you a whole window of just console, where the bottom button splits the screen between the inspector and console. Both Chrome and Safari both also have menu items to open the console.

Section 3.2

The Console is Your New Best Friend

As amazing and useful as the inspector window is, as you get more involved with coding JavaScript, you'll find the console to be your best friend. First and foremost, any JavaScript errors that occur as your page is loaded are logged to this console. Since JavaScript errors often prevent a library form loading, this should be the first place you look when something that you are completely, totally sure should be loaded is acting as though it's not there at all. Odds are that an earlier syntax error prevented something from being loaded. Check the console window before you spend an hour or two trying to figure out why your JavaScript isn't behaving.

If you are executing your JavaScript in Chrome, Safari, or Firefox with Firebug, then the console exists as an object that you can access from your JavaScript. The most common use is the method `console.log`, which takes either a string or a JavaScript object and prints it to the console. In other words, no more need to use `alert` for "I got here" style debugging. Yay! The console also provides `warn` and `error` messages which also write to the console but with either a yellow warning icon or a red error icon.

Mostly, though, the console behaves as an interactive command line into the browser DOM and imported JavaScript on the page that it's attached to. You can query the JavaScript code to get and modify the state of the objects. If jQuery is loaded in your page, than you can use jQuery, which makes modifying the DOM from the console prompt easy. The value of any expression you type in is displayed in the prompt. Also at the prompt, you can type `dir` which takes an object and displays its entire property listing. The special variable `$0` returns the currently selected element in the inspector pane, while `$1` returns the previously selected element, and so on through `$4`.

At the top right of the "Elements" pane is a search box, which does a text search of the live DOM source. The rest of the top level contains buttons that link to other panes of information. Most of these are not really relevant to our interests here, but I should mention the "Scripts" pane, which shows the source of any of the JavaScript files that are currently loaded. This pane also acts as a symbolic debugger, you can set break points and watch expressions, and pause execution.

Section 3.3

Go Forth And Inspect

There's more to the developer tools than I've listed here, and there's always more stuff being added. Using these tools effectively is very important in becoming a more effective client-side designer and developer.

Now, let's move on and look at another problem for us to solve...

Chapter 4

Ajax

After you deliver the autocomplete widget, you don't hear from Doctor What in a while. But what is time to a person who runs a Time Travel Agency? Eventually, though, you get another request.

> Dear sir or madam,
>
> I'd like to use the "More Details" link on the home page to add some dynamic data about how many people have already bought the trip. We sell oodles of these trips, though, so we're going to have to update that information dynamically from the server. What do you call that... AppleJacks? We need some of that.
>
> Sincerely,
>
> Doctor What

Doctor What, of course, is referring to Ajax. Let's explore how we can use jQuery and Jasmine to bring the full power of the dynamic Internet to even the smallest part of your web page.

In the beginning, JavaScript was basically an annoyance. In the early days, limited power, the lack of tools, and browser incompatibilities made using JavaScript for anything more complicated than a pull-down menu an exercise in total frustration.

The limited power problem largely came to an end with the creation of the `XMLHttpRequest` object (which, perhaps surprisingly, originally debuted in slightly different form in Internet Explorer.) With `XMLHttpRequest`, JavaScript could call a server and request information. For the first time, JavaScript's ability to change page elements had a point, because when combined with the ability to update information from a server, a web page could have much more dynamic behavior.

The first widely deployed application that used this dynamic facility extensively was GMail, deployed on April 1, 2004 (remember, everybody thought it was an April Fool's joke). It's hard

to regain the sense of how different GMail was from previously existing web applications – it was doing things that seemed impossible. Soon afterwords, Jesse James Garrett coined the term Ajax from the phrase "Asynchronous JavaScript and XML." The term stuck, growing to encapsulate pretty much any JavaScript that manipulates the user-visible page, whether or not a server is involved.

We'll start using Ajax to augment some of the examples we've already seen, using jQuery to manage the server interaction, and Jasmine for testing.

Section 4.1

Ajax Basics

To fulfill Dr. What's feature request, we need to augment our show/hide details toggle to get the dynamic information from an Ajax call. We'll want to make a call to the server to get the information to display. For the purposes of this chapter, we'll assume that the server is returning fully formatted HTML. Later, we'll talk about having the server return plain data and doing the formatting on the client side.

Starting with the Jasmine side, we're going to modify the spec to validate that the text actually comes from the server via an Ajax call. Here's the new spec:

Filename: spec/javascripts/togglerSpec.js (Branch: section_7_1)

```
describe("clicking a show description link", function() {
  beforeEach(function() {
    loadFixtures("one_index_trip.html");
    toggler = new Toggler();
    toggler.init();
    spyOn(toggler, 'getDescription').andCallThrough();
    spyOn($, 'ajax').andCallFake(function(ajaxParams) {
      ajaxParams.success("Description");
    });
    $(".detail_toggle").click();
  });

  it("shows the trip description", function() {
    expect($('.detail')).not.toHaveClass("hidden");
```

```
});

it('changes the link action to "Hide"',function() {
  expect($('.detail_toggle')).toHaveText("Hide Details");
});

it("calls the ajax with the correct url", function() {
  expect(toggler.getDescription.mostRecentCall.args[0])
      .toEqual("/trips/1/description");
  expect(toggler.getDescription.mostRecentCall.args[1])
      .toHaveClass("detail");
});
```

Sample 4-1-1: Specs pointing at an ajax toggler

The basic spec is the same – we simulate a click on the element and verify that its associated element is now visible. We are also checking that the text of the new element is what we expect. Exactly what we're doing in the last couple lines of the `beforeEach` method will be described in a bit. For the moment all we need to know is that those lines are how we set the returning Ajax text for our test purposes and how we verify that the URL we expected has been called.

Section 4.2

Calling the server

To make this pass, we have to make an Ajax call and use the value of that call as the new description text. Here's some code to do exactly that. We're extending the module pattern version of the toggler code from chapter 4.

Filename: app/assets/javascripts/toggler.js (Branch: section_7_1)

```
  getDescription: function(url, target) {
    $.ajax({
      url: url,
      success: function(data) {
        target.html(data);
      }
```

```
    })
  },

  toggleOnClick: function(event) {
    this.$link = $(event.target);
    this.$link.text(this.isDetailHidden() ? this.hideText : this.showText);
    this.detailElement().toggleClass(this.hiddenClass);
    if(!this.isDetailHidden()) {
       this.getDescription(this.$link.attr("href") + "/description",
          this.detailElement());
    }
    event.preventDefault();
  },
```

Sample 4-2-1: Specs pointing at an ajax toggler

The important detail is in the `getDescription` method, which calls the jQuery utility method `$.ajax`. You will probably not be surprised to learn that the `$.ajax` method makes an Ajax call. This particular call is about as minimalist as the `$.ajax` method gets. The basic version of the `$.ajax` method takes a single argument. Which seems simple, except the single argument is an object which can have a bunch of key/value pairs used as options to specify the call. Technically, all the options are optional, but it's rather hard to have a valuable Ajax call without the `url` option, which specifies what URL to send the call to.

The `success` option specifies a callback invoked if the Ajax call returns successfully from the server, meaning that the status code is `200`. The callback function itself can take up to three arguments, the first being the data returned from the server – in our case, the HTML string. The second argument is a string representation of the return status, and the third argument is the jQuery `XMLHttpRequest` object (`jqXHR` is a shortcut name), which is a wrapper around the natively defined `XMLHttpRequest` object. In addition, the jQuery `XMLHttpRequest` object is the return value of the `$.ajax` method itself. In most jQuery practice that I've seen, it's more typical to deal with the returned data via a callback like `success` than by returning the `jqXHR` object. That said, in some ways the Ajax call is easier to test if it's entirely encapsulated rather than having a callback included within the method definition.

The other options to `$.ajax` fall mostly into three basic categories: options that affect the request itself, options that affect the local browser processing around the request, and

lifecycle callbacks. As usual, this is not a complete list, but a description of the most important options.

Most of the time, you will use the `$.ajax` options that specify the details of the request itself. The `data` option takes an object and sends that object's properties as the form data or query string, depending on the HTTP method. You can specify the HTTP method with the `type` option, which defaults to `GET`, but accepts the other verb forms, such as `POST`, as capitalized strings. If the `data` value is not a string, it is converted to one.

You can also set arbitrary headers with the `headers` option, which takes an object whose keys are header keys that the server will understand. If you expect the request to be challenged via HTTP authentication then you can use the `username` and `password` options to send authentication information to the server. You can also use the `timeout` option to specify the amount of time that jQuery should wait for the server before giving up. The value is in milliseconds.

The `dataType` option allows you to specify what kind of value you are expecting from the server. By default, jQuery will try and guess based on the MIME type of the response. The options are: `html`, `json`, `jsonp`, `script`, `text`, and `xml`. These are just data types except `jasonp` and `script`. We'll discuss `json` and `jasonp` more in the next chapter. If the data type is `script`, then the result is expected to be JavaScript and is immediately evaluated. There was a time when instantly evaluated JavaScript was a major mechanism by which Ajax worked – the Rails RJS functionality depended on it – but it's much less common now because evaluating random code willy-nilly is not always the smartest idea, and it became easier to put event handlers in the client side code itself.

The `$.ajax` method has a few options that affect how the Ajax request is managed on the client side. By default, jQuery's Ajax call is asynchronous, meaning that normal code processing continues and the callback functions are invoked at an arbitrary later time depending on when the server data in returned. Sometimes you might want jQuery to block pending the server return, which you can do with `async: false` in the settings object.[2] By default, jQuery caches Ajax requests that are data, meaning not script or JSONP. That behavior can be overridden with the setting `cache: false`.

The other important local processing option is `context`, which specifies a JavaScript object which is accessible as `this` within any of the callback functions for that Ajax request. If the

2. However, `async:false` is deprecated in jQuery 1.8.

context is not specified, then inside the callback functions `this` refers to the `jqXHR` object for the request.

Section 4.3

The Ajax Lifecycle

As part of each Ajax request, jQuery triggers a series of lifecycle events. These events come in two flavors – local events which you set callbacks for on a request-by-request basis when you make the Ajax call, and global events that are triggered by any Ajax call. You can set callbacks for a global event using the regular jQuery event binding function `on`.

Let's talk about the local events first. There are four of them, of which three will be invoked during any one Ajax request cycle. The first is `beforeSend`, which – let's hear it for well named methods – is called before the Ajax request is sent off to the server. There are a few useful things you can do in a `beforeSend` callback. You can modify the data or headers in the request before it goes out the door. Also, you can block the request from happening at all. If `beforeSend` returns false, then the Ajax request ends right then and there. One way that you can use this mechanism is for validation – verify that a form submission has valid values and block the request if not.

What happens next depends on the resulting response. Either the `success` event and callback is invoked or the `error` callback is invoked. In order for `success` to be invoked, the server must return a `200` response code, and any post-processing of the data must be successful. By post-processing, I mean any parsing that might be done as a result of the response being treated as JSON, XML, or a script to evaluate. If there is an error along this chain, then the `error` callback is invoked. You are guaranteed to never have both the `success` and `error` callbacks triggered in the same request. Similarly, you are always guaranteed to have the `complete` event and callback triggered after whichever of `error` and `success` gets invoked.

If you don't like specifying all the callback functions inside the `$.ajax` call, and admittedly that can get kind of ugly if the callbacks are complex, more recent versions of jQuery allow you to specify the callbacks separately. The `$.ajax` method returns a jqXHR object. In addition to using that object to get at the response after the callback has executed, jQuery allows you to use that object to specify lifecycle events using what jQuery calls a *deferred object*. The deferred object protocol effectively allows you more flexibility in specifying callbacks. In this

case, our `getDescription` could be written with the success callback chained to the `$.ajax` call:

```
getDescription: function(url, id) {
  $.ajax({url: url}).success(function(data) {
    $("#" + id).html(data);
  });
}
```

Sample 4-3-1:

Further callbacks (including further `success` callbacks) can be chained indefinitely. (Although it's worth noting that the method name `success` here will be deprecated in jQuery 1.8 in favor of the name `done`.) You can even take hold of the jqXHR object and attach callbacks to it later:

```
getDescription: function(url, id) {
  jqxhr = $.ajax({url: url});

  // work

  jqxhr.success(function(data) {
    $("#" + id).html(data);
  });
```

Sample 4-3-2:

If you specify a callback after the Ajax call has returned, then it is executed immediately. In addition to the potential readability benefits, separating the callbacks allows you to, for example, put conditional logic outside the callback functions. This can make the callbacks themselves simpler, and therefore easier to verify.

jQuery has global events that allow you to respond to all Ajax calls in your application. Global events apply to all Ajax requests unless you specifically block them for a specific request by adding the option `global: false`. There are six global events, four of which are the global versions of the same events that can be triggered locally. The global versions are named `ajaxSend`, `ajaxSuccess`, `ajaxError`, and `ajaxComplete`. These events are triggered immediately after the associated local event, and can be bound using the `on` method. For example, you could bind a particular DOM element to always display text when an Ajax call is in progress with something like the following:

```
$("#status-light").on("ajaxSend", function() {
  $(this).html("Sending...");
})

$("#status-light").on("ajaxComplete", function() {
  $(this).html("Done");
})
```

Sample 4-3-3:

Those code snippets, however, might behave oddly if multiple Ajax requests are in flight at the same time. To manage that particular possibility, jQuery offers two additional global Ajax events. The `ajaxStart` event is fired at the very beginning of a new Ajax call – even before `beforeStart` – if there is not already a pending Ajax request at that moment. On the other side, the `ajaxStop` event is fired after `ajaxComplete` if the Ajax request has ended and there are no other currently pending requests. Binding the above snippet to the `ajaxStart` and `ajaxStop` events would make the behavior of the status indicator more robust.

Section 4.4

Ajax Shortcuts

If you find yourself doing the same Ajax options over and over again, jQuery provides some pre-defined shortcut methods that may help. Two of the simplest are `$.get` and `$.put`, which are methods that take up to four arguments: the url to call, any data that is sent as query string or post data, a callback function, and the data type. So the following two method calls are equivalent:

```
$.get("http://www.timetravel.com", {id: 3},
    function(data) { $("#result").html(data) }, 'text');

$.ajax({
    url: "http://www.timetravel.com",
    type: 'GET',
    data: {id: 3},
    success: function(data) { $("#result").html(data) }
});
```

Sample 4-4-1:

You can get even one step simpler if the request is a `GET` and the response is JSON or JavaScript, by using the `$.getJSON` or `$.getScript` methods. The JSON method is identical to the regular `$.get`, except that the response is parsed as JSON and passed to the callback function as a JavaScript object. The `$.getScript` method executes the result from the server as JavaScript and in the global scope. Generally this eliminates the need to also have a callback script, but you still can include one if needed.

If all you want to do is replace text, jQuery provides the `load` method, which is actually a method of the jQuery object. The method takes three arguments, a URL, optional data for the request, and a callback that is executed on completion after the normal behavior of the method. The normal behavior of the method makes the Ajax call, then sets the `html` of the DOM element referenced by the jQuery object – typically you'd only have one – to that HTML.

In other words, the above `GET` calls could also be written as:

```
$("#result").load("http://www.timetravel.com", {id: 3});
```

Sample 4-4-2:

Ajax parameters that are common to any call in your system can be set as default behavior using the `$.ajaxSetup` method, as in `$.ajaxSetup({type: "POST"})`, to make `POST` the default Ajax verb rather than `GET`.

jQuery also allows you to calculate the query string from a given object using the `$.param` function, so that `$.param({foo: 3, bar:2})` results in `foo=3&bar=2`. The `param` function will also handle nested objects and arrays. You can convert an entire form to a text string using `serialize`, as in `$("#user-form").serialize()`. Any child elements of the element in the jQuery object that are form elements are identified and added to the string. If you want to convert the form to an array of name/value pairs, you can use the method `serializeArray`.

Section 4.5

Ajax on Rails

This section specifically deals with integrating jQuery with a Rails application. If you have no interest in Rails, feel free to move to the next section.

Making Ajax easier has always been one of Rails core strengths. Rails' facility with Ajax was a primary reason why the framework caught on initially. While not everything the Rails framework has provided to work with JavaScript has been successful – you'll remember RJS, or if you are lucky, you won't – making the Ajax experience easy for the programmer has always been one of Rails' goals. Early in Rails' development, that meant allowing a Ruby programmer to avoid messy JavaScript for a long stretch of time. Now that the JavaScript tools have evolved significantly, the goal is more a seamless transition between Rails and JavaScript.

Rails jumped on the unobtrusive JavaScript train in Rails 3.0, providing a simple interface allowing any JavaScript toolkit to attach its own Ajax calls onto the common Rails structure. In Rails 3.1, the default library is jQuery, but a variety of different libraries have defined interfaces. We'll talk about the jQuery flavor here.

In earlier versions of Rails, the helper methods `link_to_remote` and `form_for_remote` were defined and produced a multi-line glob of nigh-impenetrable JavaScript to produce an Ajax call back to the server. Rails 3 removed both of those calls. Instead, the non-Ajax versions of these methods, `link_to` and `form_for`, grew a `:remote => true` option, which produces the attribute `data-remote = true` in the resulting HTML.

By itself, the `data-remote` attribute doesn't do anything. However, the adapter script provided by Rails adds jQuery support for binding any link with `[data-remote=true]` and causing the click event to trigger an Ajax call. Here is a section of the default jQuery adapter code:

```
$.ajax({
  url: url, type: method || 'GET', data: data, dataType: dataType,
  // stopping the "ajax:beforeSend" event will cancel the ajax request
  beforeSend: function(xhr, settings) {
    if (settings.dataType === undefined) {
      xhr.setRequestHeader('accept', '*/*;q=0.5,
        ' + settings.accepts.script);
    }
    return fire(element, 'ajax:beforeSend', [xhr, settings]);
  },
  success: function(data, status, xhr) {
    element.trigger('ajax:success', [data, status, xhr]);
  },
  complete: function(xhr, status) {
```

```
    element.trigger('ajax:complete', [xhr, status]);
  },
  error: function(xhr, status, error) {
    element.trigger('ajax:error', [xhr, status, error]);
  }
});
```

Sample 4-5-1:

In addition to `data-remote`, there are a few other options that you can include in the Rails handler. Rails offers support for the option `:confirm`, which takes a string argument, and becomes the HTML attribute `data-confirm`. If present, the user is forced to OK a JavaScript alert box with the string in order for the Ajax call to take place. The `:method` option specifies the HTTP request verb, and is expressed in the HTML as the `data-method` attribute. I trust you see the pattern at this point. You can use the option `:disable_with` to disable form elements, replacing their value with the value of the option, such as `:disable_with => 'Pending'`. Finally, although it doesn't seem to have explicit support as an option from Rails, adding `data-type` as an HTML option specifies the format type of the result, more-or-less equivalent to using a file extension on the URL of the request. The default data type is `js`.

As far as Rails is concerned, Ajax requests that are sent via the `data-remote` mechanism are treated like any other Rails request – a controller and action are identified, and the response is based on the format type specified in the request. (I believe this is a subtle change from Rails 2, where Ajax calls always followed the `js` format path.) It's incumbent on the controller to send the right data based on the format, specified with the file extension or the `data-type` parameter. It's less common in Rails 3.x, but you can also use the `xhr?` method in the controller to identify an Ajax request, that method returns true if `XMLHttpRequest` is part of the request header.

The next question is what to do with the response once the client receives it. In Rails 3, the answer is "whatever you want, but do it in your own JavaScript". Unless you specify a data type of `script`, in which case JavaScript code coming from the server is automatically `eval`-ed without the need for a further JavaScript handler. As you can see in the JavaScript snippet above, the adapter defines a series of custom events that convert the jQuery Ajax lifecycle events to local events that are bound to the sending DOM element. We can define handlers for these events in our JavaScript to manage the incoming HTML or data.

The most important event to work with is the `ajax:success` event, which is called after a successful Ajax call has returned. The function takes four arguments – the event itself, the

JSON data returned by the server, if any, the response status code, and the XHR object itself. Inside the event handler, you do whatever JavaScript you need to get the dynamic functionality for your app. In simple cases, this would involve setting the `html` of some DOM element to the HTML returned from the server. In more complex cases, you might have JSON data to parse and turn into markup.

You bind this method the same way you'd do any other jQuery binding. Since the Ajax call itself is taken care of by Rails, all we have to do here is deal with success.

```
$('.data-toggle').on('ajax:success', function(evt, data, status, xhr) {
  var clickedLink = $(this);
  var wrapper = clickedLink.closest('.container');
  var dataDiv = wrapper.find('.dynamic-data');
  dataDiv.html(xhr.responseText);
  dataDiv.addClass("data-updated");
});
```

Sample 4-5-2:

As you might expect, the `ajax:error` event is triggered if the server returns an error response, while the `ajax:complete` event is guaranteed to be triggered after the success or error event, no matter what. The `ajax:beforeSend` event is invoked before the Ajax call is made, and gives you one chance to back out. If whatever handler you invoke for `ajax:beforeSend` returns false, then the Ajax call is not made, and none of the other three events fire.

Now that we've spent several paragraphs discussing the Rails UJS Ajax mechanism, the obvious next question is whether it's worth using relative to the jQuery default Ajax workflow. When initially announced in 2009, it seemed super-cool and a clear advantage over the Rails 2 behavior. Which is still true. What is less clear, though, is whether the Rails 3 mechanism is superior to just using jQuery and binding a regular click event to a handler that makes an Ajax call.

Here are two reasons why the Rails version might be superior:

- It offers a consistent way to see whether a link triggers a remote call, both in the source code and the HTML output. You don't need to guess whether there is a jQuery event defined somewhere.

- You get all four sub-events defined for you every time and bound to the DOM elements that triggered the call. Whether or not you use them, it seems like the consistency here has a value.

That said, if you are already comfortable with the jQuery solution and have a standard that works for you, I'm not sure it's so much better that you need to drop everything and switch all your links over to the Rails mechanism.

Section 4.6

Testing Ajax with Jasmine Spies

Back in our original Jasmine spec for the Ajax call, I did a little handwave over four lines of code and said we'd talk about them later. Now, it's later. Let's look at those lines of code again.

```
spyOn(dataToggler, 'getDescription').andCallThrough();
spyOn($, 'ajax').andCallFake(function(ajaxParams) {
  ajaxParams.success("Description")
});
expect(dataToggler.getDescription.mostRecentCall.args[0])
    .toEqual("/trips/1/description");
expect(dataToggler.getDescription.mostRecentCall.args[1])
    .toHaveClass("detail");
```

Sample 4-6-1:

What we are doing here is using Jasmine spies to allow us to write specs for our Ajax calls without actually having to make Ajax calls. An Ajax call, by definition, breaks the boundary of the client-side JavaScript application. As such, Ajax is notoriously hard to unit test. A common solution to that problem is just not to test Ajax logic. We can do better by using Jasmine spies.

The way to text Ajax calls, like any third-party dependency, is by encapsulating the dependency it in its own module, and then faking the interaction with that module in the test, a process generically called *mock object testing*.[3] Alternately, you can fake the responses of

3. Pedantic side note – there is more to mock object testing than just blocking third party libraries.

the third party library globally, which allows you to test a dependency that is not encapsulated.

Jasmine Spies

The idea behind any mock object package is to allow the testing code to replace part or all of the implementation of an object with a fake implementation that returns a canned value. One classic reason for using the fake implementation is the case where the real version of the object is too slow or too non-deterministic for use in a test suite. Ajax calls, since they require a network connection, meet both criteria. In this case, rather than testing that the Ajax call returns a certain value or places the code in a particular state, we are testing that the method wrapping the Ajax call is actually called by the application with a particular set of arguments. In other words, when using mock objects, we are no longer testing the end state of the application, instead we are testing the behavior of the application during the life of the test.

More generally, you can also use mock objects to replace any object that is outside the scope of the current test. A typical usage in this mode isolates a particular method under test and specifies values for any data called from the method using mock object declarations. In this way, the mock objects focus the test to the particular method, and shield it from the details or even the existence of the methods being called.

In Jasmine terms, all mock objects are created by the method `spyOn`. In mock object parlance *spies* are a particular kind of mock object declaration where the initial creation of the mock object is separate from any expectation set on the use of the object being mocked. A Jasmine spy targets a particular function and can be specified in a couple of different ways. The most common way to specify a spy is to pass `spyOn` two arguments, the first being an object and the second being the string name of a method of that object. For example:

```
var cheeseburger = {
  availableCheeses: function() {
    // Ajax call to cheese server
  };
};

it('spies on the cheese server', function() {
  spyOn(cheeseburger, 'availableCheeses');
  cheeseburger.availableCheeses();
```

```
  expect(cheeseburger.availableCheeses).toHaveBeenCalled();
});
```

Sample 4-6-2:

In this case, the object is `cheeseburger` and the method being spied on is `availableCheeses`. When `cheeseburger.availableCheeses` is called in the next line of the test, then the spy intercepts the call, logs its details, and by default returns `undefined`. Finally, on the third line, we use the spy functionality to validate our expectation that the `cheeseburger.availableCheeses` method has been called. Obviously, this is an overly simplistic use of spies. Typically we would call a method in our test that itself invoked the method being spied on, as we do in our original example, where we spy on `dataToggler.getDescription` and `$.ajax` which are invoked as part of the click handler.

If you are familiar with other mock object frameworks such as the one used by RSpec, then this example may look a little strange. In other mock frameworks the first and third lines are effectively combined, as in RSpec's `cheeseburger.should_receive(:available_cheeses)`. In that line, setting up the fake version of the method (as in line 1 of the Jasmine version) also sets up the expectation that the method will be called (as in line 3 of the Jasmine version). By separating the declaration and the expectation Jasmine gains a little bit of flexibility, and a lot of readability. Specs using Jasmine spies can have the same structure of set up data, do something, validate results, that any other spec would have without placing an expectation to be validated among the data setup.

If you haven't used other tools, the relevant point is that Jasmine's spies are flexible and powerful. In particular, you may be interested in having the spy respond to being called in a specific way, and Jasmine provides four alternatives, all of which are methods called on the spy after it's created.

A common use case is for the spy to return a static value that is a plausible return value for the method being stubbed. This allows you to test methods that rely on an actual return value without having to invoke the method being spied on. In Jasmine, you define this behavior with the line `spyOn(cheeseburger, 'availableCheeses').andReturn(["Cheddar"])`, where the argument to the `andReturn` method is the value that the spy returns when it is called.

A related use case is when you want to simulate a failure condition where the method being spied on would throw an exception. In Jasmine, you would define the exceptional spy behavior with `spyOn(cheeseburger, 'availableCheeses').andThrow(CheeseNotAvailableException)`, again where the argument to `andThrow` is the exception that

you want thrown. Failure conditions are often hard to set up while running specs, so this is a convenient way to simulate failure.

Rather than return a static value by using `andReturn`, you can have your spy invoke it's own callback function in place of the spied method by using Jasmine's `andCallFake` method:

```
spyOn(cheeseburger, 'availableCheeses').andCallFake(function() {
  var d = new Date();
  if(d.getDay() == 0) {
    return ["Cheddar", "Swiss"]
  } else {
    return ["Provolone", "Gorgonzola"]
  }
});
```

Sample 4-6-3:

We're using this behavior in our actual spec to allow the `$.ajax` call to dynamically invoke the success callback of the Ajax method. Here, let's look at the spy declaration and the actual method declaration together:

```
spyOn($, 'ajax').andCallFake(function(ajaxParams) {
  ajaxParams.success("Description");
});

getDescription: function(url, target) {
  $.ajax({
    url: url,
    success: function(data) {
      target.html(data);
    }
  });
},
```

Sample 4-6-4:

The `andCallFake` method takes the parameters of the original call and passes them to your function. In our case, that means the parameters sent to the `$.ajax` call itself, namely, the object with the keys `url` and `success`. Inside the fake method, we explicitly invoke the success

callback, passing in the data that we are pretending was returned from the server. This is a flexible mechanism for testing Ajax callbacks.

In some cases you want the spy behavior of being able to verify that a particular method was called, but you don't have any objection to invoking the method itself. In other words, you want to literally spy on the method without actually stubbing it and replacing its implementation. A common use case would be where you have an application with several distinct layers, such as a Model/View/Controller architecture. When testing an element in one layer, you would typically use a spy to ensure that it calls the correct API method of another layer, but you don't necessarily need to go to the trouble of stubbing the method. In Jasmine, you can spy but keep the method's true behavior with `spyOn(cheeseburger, 'availableCheeses').andCallThrough()`.

Once you have the spy, there are basically two matchers that allow you to validate expectations about your application's behavior. The two are closely related, `toHaveBeenCalled` and `toHaveBeenCalledWith(arguments)`. Both of these are called as methods of a Jasmine `expect` itself called on the function object being spied on, as in:

```
expect(cheeseburger.availableCheeses).toHaveBeenCalled();
```

Sample 4-6-5:

The only difference between the two matchers is that the first matcher only tests that the spy has been invoked once, while the second one requires that it be invoked exactly once with the specific list of arguments. If you want more flexibility in the expectation you have for the spy, you need to use the spy's properties directly. The spy keeps track of the number of times it was invoked in the property `callCount`, the arguments to each call in the property `argsForCall`, as in `argsForCall[0]` to get the first call to the spy. A shortcut for the last call is `mostRecentCall.args`. So if you wanted to ensure that the spy is invoked twice, you could try this.

```
expect(cheeseburger.availableCheeses.callCount).toEqual(2);
```

Sample 4-6-6:

In our example, we use this facet of Jasmine spies to test that the correct url and target element are passed to the `getDescription` method that invokes the Ajax call. In this way, we get a kind of a chain of responsibility. One Jasmine spec specifies how the method calling Ajax should be invoked, while another assumes that the method is invoked properly and specifies what happens after a successful call.

We can use Jasmine's flexibility to test each argument separately:

```
expect(dataToggler.getDescription.mostRecentCall.args[0])
    .toEqual("/trips/1/description");
expect(dataToggler.getDescription.mostRecentCall.args[1])
    .toHaveClass("detail");
```

Sample 4-6-7:

It's easiest to test the DOM element by checking whether it has the expected `detail` class rather than by testing equality, largely because the Jasmine test and the actual code will each create separate jQuery objects pointing to the same DOM element, making a pure equality test problematic.

Section 4.7

Ajax Retrospective

We showed how to do Ajax calls from jQuery using the setup of a very simple call back to the server to show or hide a particular DOM element. In this chapter, we assumed that the return value from the server would be HTML to be inserted in the browser page.

An Ajax call as managed by jQuery allows you to specify options to the Ajax call, including the context in which the callbacks will be evaluated. Ajax calls have a life cycle, allowing you to specify callback functions for success and failure. The jqXHR object gives you even more flexibility in specifying callbacks. If you are using Rails, the Rails library that interacts with jQuery gives you custom events to respond to rather than using the jQuery callbacks.

Testing Ajax is a little bit challenging because Ajax is by definition dependent on an external server. We showed two ways to short-circuit that dependency while testing. Jasmine provides spies that allow you to replace any function with a canned value and set expectations on the application behavior with respect to that functions. In addition, the `jasmine-ajax` extension allows you to replace the Ajax response as a whole with a canned response body. Both mechanisms have value for different testing goals.

Now let's see what happens when we ask the server to start sending the client data rather than HTML.

Chapter 5

JSON

You hear from Dr. What again, this time via a pneumatic tube left on your desk, presumably as the good doctor flew by during his time travels.

> Dear sir,
>
> I really like that star rating thing that Amazon does, where they show the average rating and also a histogram. It'd be great if you could put that into our display page for each trip. Remember, Time Is Money. Oh, and also, we want the user to be able to rate the trip via clicking on the stars.
>
> Yours 'Til Niagara Falls in the year 2345,
>
> Doctor What

We quickly decide that the best way to do this is to make the server responsible for sending data to the client and allow the client to build the page segment in the browser.

From this chapter moving forward, we're changing the relationship between the server side of our application and the client side. Until now, we've had a web application with some dynamic elements. The server side is responsible for a great deal of view-layer logic, and also sends DOM elements to the client directly as HTML markup. The client-side exists primarily to tweak the HTML elements in response to user action, but the bulk of the application smarts live on the server.

A different way to structure a web application is with a thinner server that sends data to the client, and the client handles the details of view logic and creating DOM elements. There are many potential advantages to this structure. For one, it reduces the load on your server, since the server isn't calculating view results. It's also a saner separation of concerns – on the face of it, the idea that the server would be responsible for the details of the client side layout seems absurd. If the server is just responsible for data, then changing the client mode from a client website to a native mobile app or a web service gets a lot easier.

Done right, you have a clean separation of responsibilities and an easy to maintain web application. Done wrong, you have two parallel Model View Controller (MVC) architectures, one on the server, and one on the client, both of which need to be changed when the requirements change. In order for this to work on the JavaScript side, you need an easy way to transmit data between the client and server, and you need a better way to create HTML than piecing tags together from concatenated JavaScript strings. We'll be using the JavaScript Object Notation (JSON) for the data structure and a template language called Mustache for the template tool.

In this chapter, we're going to start small, and only manage a part of a page via data from the server. Later, in Book 3, we'll use Backbone.js to move all the display logic to the client-side.

Section 5.1

Retrieving Data with JSON

Our first step in creating this ratings functionality is managing the rating data itself. We're not going to concern ourselves with the server side, and we're going to have the server return the simplest version of the data possible, just the id of the trip in question, and an array of the number of people who have rated the trip for each star value one through five.

What we need is a combination of JavaScript objects that can retrieve JSON from the server with an Ajax call, perform any math logic needed to display the histogram, and then create the markup needed to display the histogram and rating stars. Naturally, we start with some tests.

Filename: spec/javascripts/ratingSpec.js (Branch: section_8_1)

```
describe("Rating things", function() {

  beforeEach(function() {
    this.incomingJSON = '{"values": [0, 2, 4, 6, 8, 10], "id": 3}';
  });

  describe("calculates values from JSON", function() {

    beforeEach(function() {
      this.rating = new Rating();
```

```
      this.rating.parseJson(this.incomingJSON);
    });

    it("has the histogram values", function() {
      expect(this.rating.values[0].count).toBe(2);
      expect(this.rating.values[0].rating).toBe(1);
      expect(this.rating.values[4].count).toBe(10);
      expect(this.rating.values[4].rating).toBe(5);
    });

    it("calculates total rating count", function() {
      expect(this.rating.totalRatings).toBe(30);
    });

    it("has histogram percentages", function() {
      expect(this.rating.values[0].percentage).toBe(2.0 / 30 * 100);
      expect(this.rating.values[4].percentage).toBe(10.0 / 30 * 100);
      expect(this.rating.values[4].width).toBe(10.0 / 30 * 100 * 3);
    });

    it("calculates average rating", function() {
      expect(this.rating.averageRating).toBe(110.0 / 30);
      expect(this.rating.ratingPercentage).toBe(110.0 / 30 / 5 * 100);
    });
  });
});
```

Sample 5-1-1: Our first rating tests

What we are doing here is writing a function that transforms our very simple JSON data object into a static JavaScript object with more detailed status, such as the percentage of the total that each rating represents and the average rating. The `beforeEach` method has the basic API. We're creating a new `Rating` object. Not to give away the surprise, but we'll be using the module pattern to build our ratings. That object will then parse the incoming JSON string and convert it to our own data representation. Note that we're pre-calculating percentages, averages, and the like, because doing so allows a much cleaner integration with the template tool we'll be discussing.

Here is the part of our new object that deals with JSON parsing.

Filename: app/assets/javascripts/rating.js (Branch: section_8_1)

```
var Rating = (function() {

  var Constructor = function(element) {
    this.element = element;
    this.totalRatings = 0;
    this.totalStars = 0;
    this.values = [];
    this.id = 0;
  };

  Constructor.prototype = {
    parseJson: function(jsonData) {
      var jsonObject = $.parseJSON(jsonData);
      this.parseValues(jsonObject);
      this.calculatePercentages();
      this.calculateAverage();
      return this;
    },

    parseValues: function(jsonObject) {
      this.id = jsonObject.id;
      var self = this;
      $.each([1, 2, 3, 4, 5], function(index, value) {
        self.values[index] = {};
        self.values[index].count = jsonObject.values[value];
        self.values[index].rating = value;
        self.totalRatings += self.values[index].count;
        self.totalStars += (value * self.values[index].count);
      });
    },

    calculatePercentages: function() {
      var self = this;
      $.each([1, 2, 3, 4, 5], function(index, value) {
        self.values[index].percentage =
            self.values[index].count * 1.0 / self.totalRatings * 100;
```

```
        self.values[index].width = self.values[index].percentage * 3;
      });
    },

    calculateAverage: function() {
      this.averageRating = this.totalStars * 1.0 / this.totalRatings;
      this.ratingPercentage = this.averageRating / 5.0 * 100;
    }
  }
  return Constructor;
})();
```

Sample 5-1-2: Rating data object

I think we have seen nearly everything in this example except for the `$.parseJSON` method, which, as you might imagine, takes in a JSON string and parses it to the resulting actual JavaScript object. In this case, the incoming object is just a list of counts, which our code iterates over to calculate various averages and percentages. The resulting digested object will look something like this:

```
{values: {1: {count: 2, rating: 1, percentage: 0.06667},
          2: {count: 4, rating: 2, percentage: 0.13333},
          3: {count: 6, rating: 3, percentage: 0.2},
          4: {count: 8, rating: 4, percentage: 0.26666},
          5: {count: 10, rating: 5,  percentage: 0.3333},
  averageRating: 3.6665,
  totalRatings: 30,
  totalStars: 110}
```

Sample 5-1-3:

Now that we have the data, we'll need a way to get it on the page, and we'll also need a form to allow the user to get rating data back to the server. First, though, let's fulfill our FCC educational programming requirement and talk about JSON a bit.

Section 5.2

Hello, JSON

I sometimes think the hardest thing about JSON is pronouncing it. For the record, there is no official pronunciation. I most often hear it as rhyming with "jay-bond" with an accent on the second syllable to distinguish it from some guy named Jason.

The format itself is deliberately simple. For the most part, JSON is a syntactically legal subset of JavaScript, specifically the part of JavaScript that defines objects. Despite the syntax being derived from JavaScript, the idea behind JSON is that JSON parsers would be simple to create for any language, making JSON an easy way to exchange data between languages – less fussy about spacing then YAML, much easier for humans to read and write than XML. Because it's so easily parsed into JavaScript, JSON has become the format of choice for web services to send data to JavaScript clients.

JSON has two basic concepts – sets of key/value pairs and arrays. A JSON string looks very much like a JavaScript object definition, for example:

```
{"ratingCount": 12,
  "tripName": "Visit Shakespeare",
  "ratedByCurrentUser": false,
  "ratingHistogram": [0, 3, 4, 2, 1, 2]}
```

Sample 5-2-1:

A JSON string begins with a curly brace, has some key value pairs, and ends with a curly brace. The key/value pairs consist of a key, a colon, and then the value. All of this is very similar to the way you would define an object in regular JavaScript. In JSON, however, the keys must be actual strings delimited by double quotes. (Strictly speaking, that is, some JSON parsers will let you get away without quoting the keys.) The value side can be a scalar value, which means a string, a number, or the special keywords `true`, `false`, or `null`. The value can also be an array, meaning square brackets with values separated by commas or another object, also defined as curly braces surrounding key/value pairs. And that's largely it. If you are into parser diagrams, http://www.json.org has a complete set of diagrams for the format, and it almost fits comfortably on a single screen.

In jQuery, the utility method `$.parseJSON` is used to convert a JSON string into the equivalent JavaScript object. The usage is about as simple as it gets:

```
x = $.parseJSON('{"ratingCount": 12}');
x.ratingCount
```

Sample 5-2-2:

You can also use the Ajax shortcut method `$.getJSON` to query a remote URL and pass the parsed JSON object directly to a callback method when the result is successful. In this case, jQuery will only invoke the callback only if the Ajax call returns successfully and the resulting JSON parse happens without error.

```
$.getJSON("/trips/1/ratings", function(data) {
  // data is the parsed object
});
```

Sample 5-2-3:

Section 5.3

Mustache and Templates

Now that we've converted our raw data into data that is ready to be presented to the user, we have two basic problems. First, we need to convert that data to HTML markup, and second we need to be able to specify that the markup is what we might expect. In a slight break from normal procedure, we'll discuss the code solution first, and then discuss how to test it.

The most direct way to create HTML markup is by building raw strings in JavaScript. Something like this, which we can think of as the way oversimplified version of displaying ratings.

```
var markup = '<div class="rating">';
$.each([1, 2, 3, 4, 5], function(index, value) {
  markup += '<div id="value_' + value + '">';
  markup += '<span>' + value + '</span>';
  markup += '<span>' + ratingObject[value] + '</span>';
  markup += '</div>';
})
markup += '</div>';
```

Sample 5-3-1: Please don't ever do this...

Building up complicated strings in JavaScript starts out ugly and quickly degenerates to truly horrific, especially as you need to add dynamic parts inside HTML elements. The only thing it has going for it is that building the HTML this way requires no dependencies on any external library.

Another option, which we've already seen, involves building the complex element in jQuery:

```
var $markup = $("<div />", {"class": rating});
$.each([1, 2, 3, 4, 5], function(index, value) {
  var $innerDiv = $("<div />", {id: "value_" + value});
  var $valueSpan = $("<span />").html(value);
  var $countSpan = $("<span />").html(ratingObject.count);
  $innerDiv.append($valueSpan, $countSpan);
  $markup.append($innerDiv);
});
```

Sample 5-3-2:

That's a little better, at the very least the amount of egregious string concatenation has been minimized. But still, this has flaws. The structure of the code is somewhat fighting the structure of the JavaScript, and if you have designers who know HTML but aren't comfortable with jQuery, it's not going to be easy for them to work with this text.

Enter the Mustache

If you've done any dynamic web development, then you are familiar with the concept of a template language, where the static parts of the HTML markup are part of the template as-is, and a special markup signifies that dynamic content needs to be evaluated and possibly interpolated into the output. Typically, these templates are evaluated with an interpreter that manages the dynamic content against a set of objects known as the context that provides the data to be inserted.

If your web development has been in Ruby, Java, or C#/ASP, you are probably most familiar with the use of the delimiters `<%` and `<%=` to denote dynamic content, and dynamic content for output. This is the tag pattern introduced by Active Server Pages (ASP) in 1999, and mimicked by Java Server Pages (JSP) and Embedded Ruby (ERb), among, I'm sure, other tools. Although this pattern is relatively simple, it's also been criticized for making the pages look cluttered. More importantly, most of the tools using this pattern allow the view template to execute arbitrary code within the template, making it very tempting for the view to get complex and

to break encapsulation by interacting with other code layers. Complex view logic is not just hard to test, it's also hard to maintain in general.

We're going to go another way with the examples in this book, and use a simpler template tool that deliberately limits the kind of logic you can use in the template in the name of guiding us to put logic in actual application code, where it can be tested, refactored, reused, and otherwise treated kindly. Tools in this category include the Velocity engine and Liquid.

There are several different template solutions for JavaScript, the one we will be using is called *Mustache*. The general mustache website is at http://mustache.github.com with the JavaScript library at https://github.com/janl/mustache.js.

Mustache has a lot going for it besides the awesome name. It's dead simple to learn. Plus, it's implemented in a number of different languages, opening up the possibility that the same mustache template can be used by your server-side framework and by your JavaScript code. The downside is that it's a bit of a purist's tool – fine by me, I'm a bit of a purist – and as such, may be limiting for some of you. If you like the basic Mustache syntax, but want a couple of extra bonus features, I advise you to checkout Handlebars.js at http://www.handlebarsjs.com.

Installing Mustache is pretty simple, all you need to do is have the `mustache.js` file in your project. Here's what a Mustache template looks like:

```
<div class="rating">
  {{#values}}
    <div id="value_{{value}}">
      <span>{{value}}</span>
      <span>{{count}}</span>
    </div>
  {{\values}}
</div>
```

Sample 5-3-3:

With that template placed in a string variable – we will discuss a couple of ways to make that happen – you'd then use that template by calling the following:

```
Mustache.to_html(template, object);
```

Sample 5-3-4:

The `template` is the string representation of the mustache template, and the object is the data used to resolve the template. Any JavaScript object will do. Mustache will merge the template and the object and return the resulting string, suitable for inserting into the DOM using, say, jQuery and the `html` method.

Using your mustache

Mustache is a ridiculously simple template language, almost simple to a fault. It's meant to be logic-less, meaning that most complex view logic is handled outside the template. In our case that means in the surrounding JavaScript. That's a good thing, a feature, which makes it easier to test and maintain logic that manages parts of the display.

Like other templating tools, Mustache allows you to take static text and interpolate it with dynamic sections based on a set of data. In Mustache, a dynamic section is delimited by a double curly brace, as in `{{name}}`. A mustache, naturally, is a sideways curly brace.

At its simplest, Mustache will assume that the text inside the braces is a property of the attached object, and will replace the entire tag with the value of that property, converted to a string. So if you have the following code:

```
var template = "Hello, {{first}} {{last}}";
var name = {first: "Ron", last: "Swanson"};
var result = Mustache.to_html(template, name);
```

Sample 5-3-5:

The resulting string is `Hello, Ron Swanson`.

You can not place an arbitrary JavaScript expression inside the mustache braces, just the name of a property of the presented object. In Mustache version 0.4 and up you can use dot-notation to dereference a complex object, like so:

```
var template = "Hello, {{name.first}} {{name.last}}";
var name = {name: {first: "Ron", last: "Swanson"}};
var result = Mustache.to_html(template, name);
```

Sample 5-3-6:

Mustache has one major control structure, a block that begins with the sequence `{{#SOMETHING}}` and ends with the sequence `{{\SOMETHING}}`. What happens inside the control structure depends on the value of the property.

If the property is `false` or a value that evaluates to false, then the block is skipped. If the value is merely `true` the block is evaluated normally, with no other special logic. This lets you conditionally include part of the template based on the values in the object.

If the property value is itself an object with properties, then the block is evaluated in the scope of that object. In other words, the complex object example could also be written as follows:

```
var template = "Hello, {{#name}}{{first}} {{last}}{{/name}}";
var name = {name: {first: "Ron", last: "Swanson"}};
var result = Mustache.to_html(template, name)
```

Sample 5-3-7:

Notice the usage of the `{{#name}}`/`{{/name}}` block. Inside that block, the properties `{{first}}` and `{{last}}` are evaluated as part of the `name` object.

But wait, there's more. If the property value at the top of the block is an array, then mustache will execute the block section once for every element of the array. Inside the block, the template is evaluated in the context of the array element. If the array element is not a complex object, the syntax `{{.}}` acts as a loop control variable.

```
var array = [2, 4, 6, 8, 10]
var template = "{{#array}} The element is {{.}} {{/array}}"
```

Sample 5-3-8:

And if that's not enough, there's one more override of the `{{#property}}` syntax. If the property value is a callable function, then the function is invoked with two arguments. The first argument is the unrendered block of text between the beginning of the `{{#property}}` block and the ending `{{/property}}`, and the second argument is a function that allows you to render the internal text, should you want the processed text. In otherwords, given this somewhat contrived example:

```
var obj = {
  "date": "Dec 21, 2011",
  "headline": function(text, renderFunc) {
    return "<h1>" + renderFunc(text) + "</h1>";
  }
```

```
};
var template = "{{#headline}}Headlines for {{date}}{{/headline}}";
```

Sample 5-3-9:

The `headline` function would be called with the first argument `Headlines for {{#date}}`, which is rendered inside the function by calling `renderFunc`. This is powerful, but somewhat hard to see a practical use case where it wouldn't make more sense to do the processing in JavaScript outside of the Mustache template.

Inside the funtion, the `this` keyword refers to the object being rendered. You can also put a layer of indirection in the object to effectively pre-process the rendering:

```
var obj = {
  "date": "Dec 21, 2011",
  "headline": function() {
    return function(text, renderFunc) {
      return "<h1>" + renderFunc(text) + "</h1>";
    }
  }
}
```

Sample 5-3-10:

Granted that looks kind of silly, but basically it means that `renderFunc` will only be invoked once, later calls to the `headline` property will have already calculated the value.

The last bit of syntax is the inverted block, `{{^property}}` to `{{/property}}`, in which the enclosed part of the template is only processed if the property is false or an empty array.

Incorporating mustache templates

Once you have defined your Mustache templates, you need to incorporate them into your JavaScript code so that the templates are available to be used. There are at least four different ways to do that, each of which has advantages, let's cover them briefly.

The easiest way to put Mustache in your code is the one we've already seen, namely make the Mustache templates string literals in your code:

```
var template = "Hello, {{first}} {{last}}";
var name = {first: "Ron", last: "Swanson"};
var result = Mustache.to_html(template, name);
```

Sample 5-3-11:

This has a clear simplicity advantage for very short template snippets, and it's more readable than building markup as a series of concatenated JavaScript strings. (Though it's generally slower than the concatenated strings.) However, since JavaScript doesn't have multiline strings, longer templates are very hard to follow written this way, especially if they have boolean or enumerable blocks. In CoffeeScript, which does have multiline strings, this mechanism is more feasible.

Option two is to store the template server-side and recover it via an Ajax call, something like this, where the `/server/template_url` would be replaced by whatever URL you'd need to call.

```
var template;
$.get('/server/template_url/', function(data) {
  template = data;
});
var name = {first: "Ron", last: "Swanson"};
var result = Mustache.to_html(template, name);
```

Sample 5-3-12:

There are a couple of advantages here. The template can be stored as a separate file server-side, and you don't have to include the template as part of your JavaScript download, and you can lazily retrieve it as needed. This is especially useful in a case where your server and client side code is sharing the Mustache templates, or in a case where the template itself is highly dynamic – a user-generated blog template, for example. The main downside is the potential for extra server traffic, and for server lag to make rendering the template slow. Grabbing and caching the templates as part of the jQuery document ready callback is often a good move here.

Another mechanism is to include the templates inside the HTML retrieved from the server. The trick here is to include the template inside a `script` tag with a specific DOM ID and a type of `text/html`. The contents of the tag will not be rendered as part of the page, but will be accessible using jQuery. So, with an HTML page that includes the following:

```
<script id='name_template' type="text/html">
  Hello,
  {{#name}}
    {{first}} {{last}}
  {{\name}}
</script>
```

Sample 5-3-13:

The following JavaScript uses the template:

```
var template = $("#name_template").html();
var name = {first: "Ron", last: "Swanson"};
var result = Mustache.to_html(template, name);
```

Sample 5-3-14:

This mechanism has a couple of advantages. It's easy to use within you JavaScript code, and relatively easy to embed the templates inside the downloaded HTML, though obviously the server-side code needs to have minimal knowledge of the templates to include them. I admit that I'm not a fan of having the bare templates be part of the HTML source, but I think you can get around that by using the `src` attribute of the `script` tag to place the template in a separate file, though I'm not aware of that particular variation actually in use.

Functionally, this plan works great. I have some aesthetic problems with it – it feels a little bit like a hack to me, but it's also possible I just need to get over it. If you like this structure, the JavaScript library ICanHas.js, available at http://icanhazjs.com provides some automated support.

The last mechanism we'll discuss is specific to the Rails asset pipeline, though you could probably use the general idea in your own code. This tool is Ruby gem called `hogan_assets`, available at https://github.com/leshill/hogan_assets. Although similar tools exist for other JavaScript template libraries. The idea here is that every Mustache template goes in a file, the `hogan_assets` gem wraps that file in a JavaScript function, and adds that JavaScript code to the asset package.

To make this work, you need to add a few lines to your manifest in `assets/javascript/application.js`

```
//= require hogan.js
//= require_tree .
//= require_tree ./templates
```

Sample 5-3-15:

Then any file in the `templates` directory that ends in `.mustache` can be rendered like so:

```
HoganTemplates["templates/template_name"].render(context);
```

Sample 5-3-16:

This version has many of the advantages of the other versions – the templates are stored in their own files, they are easily accessible to the JavaScript code, they don't bleed into the visible source. That said, because of the dependency on Rails, we're not going to use this gem in the rest of our code in this book – I do like using it in a Rails context, though.

Section 5.4

Putting template and data together

Let's put the template together with the JavaScript code that displays it, and add in the code that triggers a form submission and redraws the rating display. Our Mustache template has two parts, the top part that draws the familiar star display for ratings, and a bottom part that handles the histogram, like so:

Figure 3: Rating Display

The histogram part is simpler, let's look at that first:

```
<table>
  {{#values}}
```

```
    <tr>
      <td style="width: 10px">{{rating}}</td>
      <td style="width: 10px">{{count}}</td>
      <td>
        <div style="background-color: blue; width: {{width}}px; height: 10px">
        </div>
      </td>
    </tr>
  {{/values}}
</table>
```

Sample 5-4-1:

There's nothing fancy here in either the Mustache template or the HTML and CSS that's displaying the histogram. Each rating type gets a table row, and a `span` element is sized based on the rating's percentage of the whole. The sizing comes from the `width` property, which is explicitly calculated by the `Rating` object when we parse the JSON, so that the calculation is not in the template.

The form part of the template is simple from an HTML perspective, but it does have some more elaborate CSS dependencies to make the whole thing look nice.

```
<script id="rating_template" type="text/html">
  <p class="stars">
    <span class="empty">
      <span class="full" style="width: {{ratingPercentage}}%"/>
    </span>
  </p>
  <form action="/trips/{{id}}/update_rating">
    <input type="hidden" name="id" value="{{id}}" />
    <input type="hidden" name="rating" value="" class="form_rating" />
  </form>
  <div class="star_container">
    {{#values}}
      <a href="#rate_{{rating}}" class="star rating_{{rating}}">{{rating}}</a>
    {{/values}}
  </div>
```

Sample 5-4-2:

I don't want to get too deep into the particulars of the CSS, interested parties are directed to the file `code/time_travel/app/assets/stylesheets/trips.css.scss` for full details. A couple of quick points, though:

- The stars are done with CSS spans with background images, the empty stars on the bottom, full on the top, with the extent to which the full image is revealed governed by the `ratingPercentage` property.
- The form just has a hidden field for the id of the trip and the rating. We'll be filling in the rating value via JavaScript.
- We set up a separate anchor target for each rating value, in a loop. Once again, the items are stacked in the CSS, and their widths staggered via the `rating_{{rating}}` CSS classes so that each anchor is only exposed for the same area as the associated star. The inner text of the anchor is hidden offscreen by the CSS, but it's convenient to have the value.

The template itself has very little logic, and nothing that can really be tested in a Jasmine kind of realm. (You'd want to visually inspect the layout and such, of course.) We're already testing that the template gets the data we want, but we can also test that the template is invoked at all:

Filename: spec/javascripts/ratingSpec.js (Branch: section_8_4)

```
describe("acquires jasmine from ajax", function() {

  beforeEach(function() {
    this.rating = new Rating($("<div id='trip_3'>"));
    affix("#rating_template");
    $("#rating_template").text("{{totalStars}}");
    spyOn($, 'ajax').andCallFake(function(ajaxParams) {
      ajaxParams.success(incomingJSON);
    });

  });

  it("knows its url", function() {
    expect(this.rating.url()).toEqual("/trip/3/rating.json");
  });
```

```
    it("can get its template", function() {
      expect(this.rating.template()).toEqual("{{totalStars}}");
    });

    it("can acquire data", function() {
      this.rating.acquireJson();
      expect($(this.rating.element).html()).toEqual("110");
    });

  });
```

Sample 5-4-3:

The only item in the test that we haven't already seen is where we are using Justin Searls' `jasmine-fixture` library in lieu of creating a separate file fixture. The line creates a DOM element with properties based on the arguments to the `affix` method and places the element in the DOM where Jasmine tests can see it. In this case, we are just creating a minimal fake template so we can tell if the template code is called when we ask to `acquireJson`.

The three specs here are specifying that the rating object knows what URL to call to get its data, that it knows how to extract its template from the DOM, and that all those pieces are put together properly.

Here's sample code to pass the tests.

Filename: app/assets/javascripts/rating.js (Branch: section_8_4)

```
    url: function() {
      return "/"+ $(this.element).attr("id").replace("_", "/") + "/rating.json";
    },

    renderTemplate: function(template) {
      return Mustache.to_html(this.template(), this);
    },

    reloadData: function(data) {
      this.parseJson(data);
      $(this.element).html(this.renderTemplate(this.template()));
    },
```

```
  template: function() {
    return $.trim($("#rating_template").html());
  },

  acquireJson: function() {
    var self = this;
    $.ajax({
      url: this.url(),
      dataType: 'text',
      success: function(data) {
        self.reloadData(data);
      }
    });
  },
```

Sample 5-4-4:

I don't think there are any new features in this code. When asked to acquire data, the rating object makes an Ajax call, then calls the `reloadData` method, which parses the data, and merges it with the template. There are helper methods to access the URL for the Ajax call and the template itself.

Finally, we need to load all this code on `document.ready`, and we need a click handler to submit a form when the stars are clicked on. The click handler has two parts. One part is outside the `Rating` object, and sets up the click targets, and creates a new `Rating`:

Filename: app/assets/javascripts/rating.js (Branch: section_8_4)

```
var loadClickHandlers = function() {
  $(document).on('click', '.star_container .star', function(event) {
    event.preventDefault();
    var $ratingElement = $(this).parents(".rating");
    var rating = new Rating($ratingElement);
    rating.updateJson(this);
  })
}
```

Sample 5-4-5:

The actual click object is a DOM element contained by the `.rating` div element we used to seed the rating display. Then we use the jQuery method `parents` to walk up the DOM tree to find the `.rating` element as an ancestor of the click target. Having created the rating object, we then call the `updateJson` method, which is similar to the `acquireJson` method.

Filename: app/assets/javascripts/rating.js (Branch: section_8_4)

```
    updateJson: function(clickedElement) {
      var $form = this.element.find("form");
      $form.find(".form_rating").val($(clickedElement).text());
      self = this;
      $.ajax({
        url: $form.attr('action'),
        dataType: 'text',
        type: 'post',
        data: $form.serialize(),
        success: function(data, status) {
          self.reloadData(data);
        }
      });
    }
```

Sample 5-4-6:

This time, however, we're doing downward searches from the rating element using jQuery's `find` method. We're locating the `form` element itself, as well as the hidden field inside it so that we can set the value of the hidden field with the value from the click target. We then send the form data back to the server, expecting an identically structured JSON package to the one we got before, but with new data.

Finally, we need to initialize everything at `document.ready`:

Filename: app/assets/javascripts/rating.js (Branch: section_8_4)

```
var loadRatings = function() {
  $(".rating").each(function() {
    var rating = new Rating(this);
    rating.acquireJson();
  })
};
```

```
$(function() {
  loadRatings();
  loadClickHandlers();
});
```

Sample 5-4-7:

At this point, we have a working ratings widget. I'd like to follow up on a couple of items: how jQuery lets you traverse the DOM and jQuery objects, and then talk about a possible refactoring of the setup code.

Section 5.5

Traversing jQuery

When you are working with jQuery, there are two different data structures that you will frequently want to manage: the DOM tree itself, and the jQuery object containing elements matched by a selector. Naturally, jQuery provides many support methods for walking through these structures.

Walking Up, Down, and Sideways in the DOM Tree

Let's start with the DOM tree. If you have a jQuery object corresponding to a particular DOM element, you can go in three directions: you can search down through all descendant elements, you can search up through all ancestor elements, or you can go sideways and catch sibling elements.

Down is the easiest direction. The jQuery `find` method takes a jQuery selector as an argument and returns every DOM element that is both a descendent of one of the elements in the jQuery object and matches the selector. For example, the `updateJson` code we just saw:

```
var $form = this.element.find("form");
$form.find(".form_rating");
```

Sample 5-5-1:

In this snippet, we are finding a `form` tag that is subordinate to the rating element, and then an element with a DOM class of `form_rating` underneath the form. You would use `find` to limit

the scope of a selection. One common use case would be having the same HTML structures on a page multiple times – we'd see that if we had multiple trips on a single page, each with its own rating display. In a case like that, the `find` method allows us to act on one particular part of the page secure in the knowledge that our selector won't grab DOM elements from any other part of the page.

A slightly more specialized downward DOM search is the `children` method, which acts just like `find` but is limited to only elements that are direct children of the items in the jQuery object. Unlike `find`, the `children` method can be called with no arguments, in which case it returns all the immediate children of all the DOM elements in the jQuery object. The `children` method does not include text that is not part of a child element, to get that text as well as child elements, you use the `contents` method.

Traveling up the DOM tree is similar to traveling down in that jQuery provides separate methods for just one level of search and for multiple level searches. The multiple level method is `parents` – we use it in the rating widget to find the base rating element from the click target:

```
var $rating_element = $(this).parents(".rating");
```

Sample 5-5-2:

When the `parents` method is called on a jQuery object with a selector as an argument, it returns any DOM element in the tree that is an ancestor of an element in the object and which matches the selector. The `parents` method can also be called without an argument, in which case it returns all the ancestors of the jQuery objects. The related method `parent` is similar, but rather than walk all the way up the DOM tree, it only returns DOM elements that are immediate parents of the elements in the jQuery object. The return set of objects is ordered with those closest to the original elements first.

There are a couple of useful variants of the `parents` search. The `closest` differs from the `parents` method in two ways: the search starts with the element itself, rather than a parent, and the first matching parent element stops the search. The returned jQuery object has at most one DOM element – if there are no elements matching the selector argument, then the resulting jQuery object is empty.

The `parentsUntil` method takes a selector argument and, starting with the parent of the original element returns a jQuery object containing all parent elements up to, but not including the first ancestor that matches the selector. Finally, the method `offsetParent` finds

the closest ancestor object whose position is explicitly set with a CSS attribute that is `absolute`, `fixed`, or `relative`. This can be useful for graphics and placement operations.

The basic method for traveling sideways in the DOM tree is `siblings`. It returns a jQuery object of all the elements that are siblings to the original objects. It takes an optional selector argument which filters the siblings to those that match the selector. The original DOM element is not part of the returned list, only those other elements that share a parent object.

jQuery also provides searches that only go one way in the tree, three related methods that look toward later siblings, and three that look back toward previous ones. The methods `next` and `prev` return one element for each element in the original object – either the immediate predecessor or successor of the object among its siblings. Both methods take an optional selector argument, used for filtering. The `nextAll` and `prevAll` methods return all elements in the desired direction, again with an optional selector as a filter, while the `nextUntil` and `prevUntil` methods take a selector argument and return all elements in the desired direction until and not including the first element to match the selector argument.

Managing a list of jQuery objects

There are a number of different things you can do with the jQuery object itself. Most of these are done to filter or isolate specific elements within the DOM tree. This section is an overview of all the mechanisms, but is not a complete listing – several of these methods have more obscure alternate forms that take different arguments.

The basic method for filtering a jQuery object is, naturally, `filter`. The method has two primary forms, the first takes a jQuery selector and returns a new jQuery object that contains all the DOM elements from the original set that match the selector:

```
var $divs = $('div');
var headers = $divs.filter(".header");
```

Sample 5-5-3:

That example is a little contrived, normally, you'd manage that functionality with the single selector `div.header`. The filter object comes in handy when there might be a lot of code between the creation of the object and the filtering, or for selectors that aren't necessarily CSS based, or which might be dynamic and changing. A second form of `filter` takes a function as an argument. The function is successively called with each DOM element in the

original jQuery object as an argument, and returns an new jQuery object with all the elements for which the function returns `true`.

The opposite of `filter` is `not`, which has the same arguments but returns a list of the objects that don't match the selector or for which the function returns `false`. A refinement of `filter` is `has`, which takes a selector as an argument and returns every element in the original jQuery object which has a descendent element that matches the selector. The point here is that the descendants are being compared, but the original parents are being returned.

jQuery will also let you create the union of two different selectors using the `add` method. One form of the `add` method takes a selector argument and creates a new jQuery object containing all the elements of the first object and every DOM element on the page that matches the selector. You can also use `add` with a second jQuery object as the argument, in which case the returned object contains the contents of the two objects joined together.

All of these methods, including the DOM search methods such as `find` have the property that the jQuery object returned from the method is different from the one that the method was called on. As mentioned very early on in our discussion of jQuery, jQuery is designed to allow for long chains of method calls. During a chain of calls, jQuery maintains a history stack of each successive jQuery object as modified by the method being called.

You can get to that stack in a couple of different ways. The one that I can see being somewhat useful is the method `andSelf`, which appends the previous state of the call history to the current state, as in the following snippet.

```
$("li.selected").prevAll().andSelf();
```

Sample 5-5-4:

At the end of this snippet, the jQuery object contains a list of all items that are previous siblings of the `selected` list items, via `prevAll` call, and the selected items themselves, via the `andSelf` method.

jQuery also provides the `end` method, which changes the jQuery object back to its state before the most recent filter. In other words, you would filter the jQuery object, do some operation on the filtered set, then use `end` to return to the unfiltered set of objects. So:

```
$("li").find(".selected").removeClass("selected")
  .end().addClass("unselected");
```

Sample 5-5-5:

The `find` call returns a list of selected list elements, and then the `end` call returns the value of the jQuery object previous to that filter, namely the set of all list elements in the page, so the `addClass` call is applied to all list elements.

In case it's not clear, I think that as much as chaining methods is part of jQuery's design, using `end` to change the state of the chain is nuts. It's hard to imagine a benefit of chaining that's worth how hard it is to read a line of jQuery code that uses `end` to manage the state of the jQuery object.

jQuery also provides a few functions that allow you to treat the jQuery object more like an array. The `each` method, which we've already seen, takes a function argument, and calls the function once for each item in the jQuery object. Within the functional argument to `each`, the variable `this` refers to the DOM element being invoked. The `map` method also takes a functional argument and calls it for each element in the jQuery object. The `map` method however, returns a new jQuery object containing the return values of each successive call to the functional argument of `map`.

There are a few methods that let you convert the jQuery object to a known subset. The method `first` returns a new jQuery object containing only the first element in the original jQuery object, and the similar method `last` returns a new jQuery object containing the last method. The general form of the method is `eq`, which takes an integer argument. If the integer is zero or positive, then the new jQuery object contains the element at that index in the original object. If the integer is negative, then we return the element counting back from the end of the original object, with `-1` indicating the last element, `-2` the next to last, and so on. If you want a subset of more than one element, you can use the `slice` method, which takes a start and end index, with the same values as `eq`. If the second argument is left off, then the subset continues until the end of the original object.

The method `is` returns a boolean true or false. It takes a few types of arguments. If the argument is a selector, jQuery object, or DOM element, `is` returns true if any element of the jQuery object matches the selector. If the argument is a function, the function is called for every element in the jQuery object, and the `is` returns true if the function returns true for any element in the object.

Section 5.6

JSON Retrospective

JSON has become important as an interchange between servers and clients. As more of the complexity of a web application is pushed to the client, the server can become little more than an API returning JSON data to be processed by the client.

As the client becomes more complex, it becomes useful to have some structure or framework to build client logic around. Which is where Backbone.js and Ember.js come in. Stay tuned for Books 3 and 4, available in the nick of time from a website near you, as long as that website is http://www.noelrappin.com.

Chapter 6

Acknowledgements

A number of people helped in the creation of this book, whether they knew it or not.

As you may know, this book began life with a non-me publisher. Technical reviewers of the manuscript at that point included Pete Campbell, Kevin Gisi, Kaan Karaca, Evan Light, Chris Powers, Wes Reisz, Martijn Reuvers, Barry Rowe, Justin Searls, Stephen Wolff. Kay Keppler and Susannah Pfalzer acted as editors. Thanks to all of them.

Avdi Grimm provided a lot of useful tools for self-publishing.

I've been lucky enough to work with people who were willing to share JavaScript experience with me, including, but not limited to: Dave Giunta, Sean Massa, Fred Polgardy, and Chris Powers.

Justin Searls and his Jasmine advocacy and toolkit have been a huge help.

One of the great things about self-publishing is that people take the time to help improve the book by pointing out mistakes and typos. Thanks to Pierre Nouaille-Degorce, Sean Massa, Vlad Ivanovic and Nicolas Dermine for particular efforts.

Emma Rosenberg-Rappin helped me design the noelrappin.com web site, pick cover fonts, and also did some copyediting.

Elliot Rosenberg-Rappin inspired the idea of a spaceship cover and laughed at some of my jokes.

This book, and many other wonderful things, would not exist without my wife Erin, who has been nothing but supportive through the ups and downs of this project. She's amazing.

Chapter 7

Colophon

Master Space and Time with JavaScript was written using the PubRx workflow, available at https://github.com/noelrappin/pub_rx. The initial text is written in MultiMarkdown, home page http://fletcherpenney.net/multimarkdown/. PubRx augments MultiMarkdown with extra descriptors allowing for page layout effects, such as sidebars, that Markdown does not manage on its own.

PDF conversion is managed by PrinceXML http://www.princexml.com, with a little bit of additional processing by PubRx. EPub and Mobi generation is managed by using the Calibre http://calibre-ebook.com/ command line interface, using a method described by Avdi Grimm's Orgpress https://github.com/avdi/orgpress.

For the PDF version, the header font is Museo and the body font is Museo Sans. Both are free fonts from the exljbris font foundry http://www.exljbris.com/. The code font is "M+ 1c" from M+ Fonts http://mplus-fonts.sourceforge.jp/. On the cover, the title font is Bangers by Vernon Adams http://code.google.com/webfonts/specimen/Bangers, and the signature font is Digital Delivery, from Comicraft http://www.comicbookfonts.com/, and designed by Richard Starkings & John 'JG' Roshell. Cover image credit is on the title page up front.

Section 7.1

Image Credits

Cover

The original image used as the basis of the cover is described at http://commons.wikimedia.org/wiki/File:Challenger1983.jpg as a scene of the Space Shuttle Challenger taken during STS–6 in 1983 with a 70mm camera onboard the shuttle pallet satellite. According to NASA policy, the photo is in the public domain.

Time Travel Adventure Images

Mayflower http://commons.wikimedia.org/wiki/File:Plymouth_Mayflower_II.jpg. Replica ship Mayflower II at the State Pier in Plymouth, Massachusetts, US. This work has been released into the public domain by its author, wikitravel:user:OldPine.

Shakespeare http://commons.wikimedia.org/wiki/File:Shakespeare.jpg. It may be by a painter called John Taylor who was an important member of the Painter-Stainers' Company. National Portrait Gallery, London, Public domain.

Mars http://commons.wikimedia.org/wiki/File:Mars_Hubble.jpg. NASA's Hubble Space Telescope took the picture of Mars on June 26, 2001, when Mars was approximately 68 million kilometers (43 million miles) from Earth. Public Domain.

Cubs http://commons.wikimedia.org/wiki/File:Patrick_Joseph_Moran_1908.jpg. Patrick Joseph Moran, Chicago baseball player, standing at home plate with bat in hand during baseball game. From the Library of Congress George Grantham Bain Collection - http://hdl.loc.gov/loc.pnp/cph.3c33635. Author unkown. Public Domain.

Lewis and Clark http://commons.wikimedia.org/wiki/File:Lewis_and_Clark.jpg. Lewis portrait by Charles Wilson Peale. Clarke portrait by Charles Wilson Peale. Public domain.

Xerox PARC http://commons.wikimedia.org/wiki/File:Xerox_Alto_mit_Rechner.JPG. Xerox Alto computer. Placed into public domain by user Joho345.

Washington Detail of Washington at the Delaware by Edward Hicks, http://commons.wikimedia.org/wiki/File:Washington_at_the_Delaware_c1849_Edward_Hicks.jpg. Public Domain.

Napoleon Portrait by Jacques-Louis David http://commons.wikimedia.org/wiki/File:Jacques-Louis_David_017.jpg Public Domain.

Everest Himalaya from the International Space Station. http://commons.wikimedia.org/wiki/File:Himalayas.jpg. Public Domain per NASA policy.

Beatles http://commons.wikimedia.org/wiki/File:The_Fabs.JPG. UPI photo, Public Domain in the United States.

Enigma http://commons.wikimedia.org/wiki/File:Enigma.jpg. Enigma machine on display at the NSA. Public Domain.

Marco Polo http://commons.wikimedia.org/wiki/File:Marco_Polo_-_costume_tartare.jpg. Public Domain.

Chapter 8
Changelog

Release 001: July 23, 2012 Initial release to those who signed up for early information.

Release 002: July 30, 2012

- Removed line numbers from code samples. I'm generally not referencing them in the body of the text, and they were really screwing up code layouts in Kindle and ePub.
- Book URL added to preface.
- Setup rake task alluded to in text actually added to code.
- Added images to Time Travel Adventure site and credits to colophon
- Added chapter on WebKit developer tools, though it needs more work and an update
- Fixed a couple of bugs in the autocomplete widget

Release 003: August 18, 2012

- Fixed typos and errors reported by Pierre Nouaille-Degorce, Sean Massa, Avdi Grimm, Vlad Ivanovic, and Ashish Dixit

Release 004: October 2, 2012

- Fixed typos and errors reported by Vlad Ivanovic