The Pragmatic Programmers

Web Development Recipes

Brian P. Hogan,
Chris Warren,
Mike Weber,
Chris Johnson,
and Aaron Godin

edited by Susannah Davidson Pfalzer

Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at *pragprog.com/my_account*. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at *pragprog.com/titles/wbdev/errata*, or by using the links at the bottom of each page.

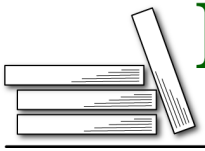
Thank you for being part of the Pragmatic community!

Andy & Dave

Web Development Recipes

Brian P. Hogan
Chris Warren
Mike Weber
Chris Johnson
Aaron Godin

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-93435-683-8
Printed on acid-free paper.
Book version: B1.0—August 17, 2011

Contents

Preface

It's no longer enough to know how to wrangle HTML, CSS and a bit of JavaScript. Today's web developer needs to know how to build interactive interfaces, write testable code, integrate with other services, and sometimes even do some server configuration, or at least a little bit of backend work. This book is a collection of more than 40 practical recipes that range from clever CSS tricks that will make your clients happy to server-side configurations that will make life easier for you and your users. You'll find a mix of tried-and-true techniques and cutting edge solutions, all aimed at helping you truly discover the best tools for the job.

Who's This Book For?

If you make things on the web, this book is for you. If you're a web designer or front-end developer who's looking to expand into other areas of web development, you'll get a chance to play with some new libraries and workflows that will help you be more productive, and you'll get exposed to a little bit of that server-side stuff along the way.

If you've been spending a lot of time on the backend, and you need to get up to speed on some front-end techniques, you'll find some good stuff here as well, especially the sections on workflow and testing.

One last thing—a lot of these recipes assume you've had a little experience writing client-side code with JavaScript and jQuery. If you don't think you have that experience, read through the recipes anyway and pick apart the provided source code. Consider the more advanced recipes as a challenge.

What's in This book?

We've included a a bunch of great stuff to get you started on the path to more advanced web development. Each recipe poses a general problem, then lays out a specific solution to a scenario you're likely to encounter, whether it's how to test your site across multiple web browsers, how to quickly build and automatically deploy a simple static site, how to create a

simple contact form that emails results, or how to configure Apache to redirect URLs and serve pages securely. We'll take you through both the how and the why, so you can feel comfortable using these solutions in your projects. Since this is a book of recipes, we can't go into a lot of detail about more complex system architecture, but you'll find some suggestions on where to go next in each recipe's Further Exploration section.

We've organized the recipes into chapters by topic, but you should feel free to jump around to the topics that interest you. Each chapter contains a mix of beginner and intermediate recipes, with the more complex recipes at the end of each chapter.

In Chapter 1, *Eye Candy Recipes*, on page 1, we cover some ways you can use CSS and other techniques to spice up the appearance of your pages.

In Chapter 2, *User Interface Recipes*, on page 17 you'll use a variety of techniques to craft better user interfaces, including JavaScript frameworks like Knockout and Backbone, and you'll look at how to make better templates for sending HTML emails.

In Chapter 3, *Data Recipes*, on page 97 you'll look at ways you can work with user data. You'll construct a simple contact form, and you'll take a peek at how to build a database-driven application using CouchDB's CouchApps.

In Chapter 4, *Mobile Recipes*, on page 125, you'll take user interfaces a step further and look at ways you can work with the various mobile computing platforms out there. You'll spend some time with jQuery Mobile, look at how to handle multitouch events, and dig a little deeper into how to determine how and when to serve a mobile version of a page to our visitors.

In Chapter 5, *Workflow Recipes*, on page 131, we'll show you ways you can improve your processes. We'll investigate how SASS can make your life easier when managing large stylesheets, and we'll introduce you to CoffeeScript, a new dialect for writing JavaScript that produces clean, compliant results.

In Chapter 6, *Testing Recipes*, on page 153, you'll create more bullet-proof sites by using automated tests and we'll show you how to start testing the JavaScript code you write.

Finally, we'll turn our attention to moving into production in Chapter 7, *Hosting and Deployment Recipes*, on page 167. We'll walk you through building a virtual machine so you have a testing environment to try things out before you set up your real production environment, and we'll cover how

to set up secure sites, do redirects properly, and protect your content. We'll also show you how to automate the deployment of websites, so you won't accidentally forget to upload a file.

What You Need

We'll be introducing you to many new technologies in this book. Some of these are fairly new and somewhat subject to change, but we think they're compelling and stable enough to talk about at an introductory level. That said, web development moves quickly. We've taken steps to ensure you can still follow along, by providing copies of the libraries we use in these recipes with the book's source code.

We've tried to keep the prerequisites to a minimum, but there are a few things you'll want to get set up before you dig in.

HTML5 and jQuery

We'll use HTML5-style markup in our examples. For example, you won't find any self-closing tags in our markup, and you might see some new tags like <header> or <section> in some of the examples. If you're not familiar with HTML5, you may wish to read *HTML5 and CSS3: Develop with Tomorrow's Standards Today* [Hog10].

We'll also use jQuery, as many of the libraries we introduce in these recipes rely on it. Our code examples will fetch jQuery 1.6.2 from Google's content delivery network.

The Shell

You'll work with various command-line programs in these recipes whenever possible. Working on the command-line is often a huge productivity boost, because a single command can replace multiple mouse clicks, and you can write your own scripts to automate these command line tools. The *shell* is the program that interprets these commands. If you're on a Windows machine, you'll use the Command Prompt. If you're on OSX or Linux, that's the Terminal.

Shell commands will look something like this:

```
$ mkdir javascripts
```

The $ represents the prompt in the shell, so you're not meant to type it in. The commands and processes you'll use are platform-independent, so whether you're on Windows, OSX, or Linux, you'll have no trouble following along.

Ruby

Several recipes in this book require that you have the Ruby programming language installed. We'll be using some tools that require Ruby to run, such as Rake and Sass. We've included a short appendix that walks you through installing Ruby which you can find on Appendix 1, *Installing Ruby*, on page 187.

QEDServer

Several of the recipes in this book make use of an existing product management web application. You can work with this application by installing QEDServer, a standalone web application and database that requires very little setup. QEDServer works on Windows, OSX, and Linux. All you need is a Java Runtime Environment. Whenever you see us refer to our "development server", we're talking about this. It gives us a stable web application backend for our demonstrations, and it gives you a hassle-free way to work with AJAX requests on your local machine.

The examples of this book will run against the version of QEDServer that we've bundled with the book's code examples, which you should download from the book's website.

To use QEDServer, you start the server with `start.bat` on Windows or `./start.sh` on OSX and Linux. This creates a `public` folder which you can use for your workspace. If you create a file called `index.html` in that public folder, you can view it in your web browser by visiting http://localhost:8080/index.html.

Online Resources

The book's website[1] has links to an interactive discussion forum as well as a place to submit errata for the book. You'll also find the source code for all the projects we build. Readers of the eBook can click on the box above the code excerpts to download that snippet directly.

We hope you enjoy this book, and that it gives you some ideas for your next web project!

Brian, Chris, CJ, Mike, and Aaron

1. http://pragprog.com/titles/wbdev/

CHAPTER 1

Eye Candy Recipes

A solid application is great, but a couple of extra touches on the user interface can make a huge difference. If they're easy to implement, that's even better.

In this section, we'll use CSS to style some buttons and text, and we'll do some animations using CSS and JavaScript.

Recipe 1

Styling Buttons and Links

Coming soon...

Recipe 2

Styling Quotes with CSS

Coming soon...

Recipe 3

Creating Animations With CSS3 Transformations

Problem

For many web developers, Flash has been the go-to tool for developing sites with animations, but these animations aren't visible from the iPad, iPhone, and other devices that don't support Flash. In cases where the animation is really important to our customers, we'll need a non-Flash workaround.

Ingredients

- CSS3
- jQuery

Solution

With the advent of CSS3 transitions and transformations, we now have the option to define animations natively, without having to use plugins like Flash. These animations will only work with newer mobile browsers and the latest versions of Firefox, Chrome, Safari, and Opera, but the logo itself will be visible for all users, even if they don't see the animations. To make the animation work in other browsers, we would continue to rely on the Flash version.

Our current client's website originally had its logo done in Flash so that a "sheen" could be seen crossing the logo when the user loaded the page. He just got a new iPad, and he's frustrated that his animation doesn't display, but even more worried that his logo doesn't even show up. While the missing effect wouldn't break the entire site, the missing logo does remove some of the site's branding. We're going to make the logo visible in all browsers and add back the animation for browsers that support CSS3 transformations.

Let's start with the markup for the header that contains our logo. We'll add a class to the <img> tag so we can access it from the stylesheet later.

Download csssheen/index.html

```
<header>
  <div class="sheen"></div>
  <img src="logo.png" class="logo">
```

```
</header>
```

To get this effect, we're going to create a semi-transparent, angled and blurred HTML block that moves across the screen once the Document Object Model (DOM) is loaded. So let's start by defining our header's basic style. We want a blue banner that crosses the top of our content. To do this we give our header the desired width, and position the logo in the upper left corner of our header.

Download csssheen/style.css

```
body {
  background: #CCC;
  margin: 0;
}

header {
  background: #436999;
  margin: 0 auto;
  width: 800px;
  height: 150px;
  display: block;
  position: relative;
}

header img.logo {
  float: left;
  padding: 10px;
  height: 130px;
}
```

With our basic layout in place, we can add the decorative elements for the animation. Let's first create the blurred HTML element, but since this is an extra effect and has absolutely nothing to do with the content of our site, we want to do it with as little extra HTML markup as possible. We'll make use of the with the "sheen" class that we defined in our markup to make this work.

If we look at our page now (Figure 1, *The "sheen" is visible and unstyled*, on page 6) we see that we've added a thin, white, transparent line that's taller than our header. We're off to a great start. Now we want to reposition the sheen element so that it's blurred, starts left of the header and is slightly angled.

This is where things get a little tricky. Because the various browsers are still deciding how to support transformations and transitions we have to add the specific browser prefixes to ensure that each browser picks up on the style change. So even though, for what we're doing at least, each style

Figure 1—The "sheen" is visible and unstyled

declaration has exactly the same arguments, we need to add the various prefixes to ensure that each browser applies the style. We also want to add a non-prefixed style definition so our style will work when the CSS3 spec is agreed upon. For example you'll see that we don't declare an -o-box-shadow style because newer versions of Opera don't even recognize that style anymore, and Firefox 4+ no longer uses the -moz-box-shadow style, but still recognizes it and converts it to just box-shadow. However we still keep the -moz-box-shadow style in place to support Firefox 3. In the code, on page 6 we have to sacrifice clean code for functionality:

Download csssheen/style.css

```
header .sheen {
  content: '';
  height: 200px;
  width: 15px;
  background: rgba(255, 255, 255, 0.5);
  float: left;
  -moz-transform:     rotate(20deg);
  -webkit-transform:  rotate(20deg);
  -o-transform:       rotate(20deg);
  position: absolute;
  left: -100px;
  top: -25px;
  -moz-box-shadow:    0 0 20px #FFF;
  -webkit-box-shadow: 0 0 20px #FFF;
  box-shadow:         0 0 20px #FFF;
}
```

With our styles in place we're almost ready to animate our sheen element. Next we'll add the transition declarations, which we'll use for controlling the animation. For now, we'll have to rely on browser specific prefixes.

Download csssheen/style.css

```
header .sheen {
  -moz-transition:    all 2s ease-in-out;
  -webkit-transition: all 2s ease-in-out;
  -o-transition:      all 2s ease-in-out;
  transition:         all 2s ease-in-out;
```

```
}
```

The transition definition takes 3 arguments; the first tells the browser which CSS attributes should be tracked. For our example we only want to track the left attribute since we're animating the sheen as it travels across the header. This can also be set to all to control the transition of any attribute changes. The second parameter defines how long the animation takes, in seconds. This value can be a decimal, like 0.5s, up to multiple seconds for a longer transition when slower changes are desired. The final argument is the name of the timing function to use. We just use one of the default functions, but you can define your own as well. Caeser[1] is a tool that we can use to help define our own function.

Next, we need to add a style declaration that defines where we want the sheen to end up. In this case it should end on the right side of the header. We *could* attach this to the hover event:

```
header:hover .sheen {
  left: 900px;
}
```

But if we did, then the sheen is going to go back to it's starting spot when the user hovers away from the header, We want to make this a one time deal, so we're going to have to use a little bit of JavaScript to change the state of the page. We'll add a special class to our stylesheet called loaded which positions the sheen all the way at the end of the logo, like this:

Download csssheen/style.css

```
header.loaded .sheen {
  left: 900px;
}
```

Then we'll use jQuery to add that class to the header which will trigger the transition.

```
$(function() { $('header').addClass('loaded') })
```

When looking at Figure 2, *The "sheen" is styled but still visible outside of the header*, on page 8 you may be thinking that all you're doing is moving a blurry bar across the screen. But now that we're done styling the sheen, we can clean up the overall look by adding a single style tweak. We'll add a style of overflow: hidden;, which will hide the part of the sheen that hangs over the header.

1. http://matthewlein.com/ceaser/

Figure 2—The "sheen" is styled but still visible outside of the header

Download csssheen/style.css

```
header {
  overflow: hidden;
}
```

With all of our styles in place, we can trigger the entire animation just by changing a CSS class on an element. We no longer have to rely on a JavaScript animation suite or Flash for adding smooth animations to our websites.

This approach has the added advantage of saving our users' bandwidth. While this doesn't affect most users, we don't always know when a user might visit our site from an iPad or another mobile device using cellular coverage. This approach means less files to download and therefore faster load times for our visitors. We should always keep site optimization in mind when developing websites.

In browsers that don't support these new style rules, our site will simply display the logo image. By separating style from content we get the benefit of backwards compatibility and better accessibility for users with screen readers, as the <img> tag contains the alternative text.

To make the animation work on all browsers, we could simply use this solution as a fallback to the original Flash solution, placing our <img> within the <object> tag that embeds the Flash movie.

Further Exploration

We only covered a few of the transformations and transitions that are available to us. There are other transformation options available like scaling and skewing. We can also get more fine-grain control over how long each transformation takes, or even which transformations we actually want to transition. Some browsers also give you the ability to define your *own* transitions. The built-in control that we're finally getting over animations is very exciting and well overdue.

Also See

- Recipe 1, *Styling Buttons and Links*, on page 2
- Recipe 2, *Styling Quotes with CSS*, on page 3
- Recipe 28, *Building Modular Stylesheets With Sass*, on page 134

Recipe 4

Creating Interactive Slideshows With jQuery

Problem

Just a few years ago, you'd probably create a Flash movie if you wanted to have an animated slideshow on your website. Simple tools would make this an easy process, but maintaining the photographs in the slideshow often means rebuilding the Flash movie. Additionally, many mobile devices don't support the Flash player, which means they can't see the slideshows at all. We need an alternative solution that works on multiple platforms and is easy to maintain.

Ingredients

- jQuery
- The jQuery Cycle Plugin[2]

Solution

We can build a simple and elegant image slideshow using jQuery and the jQuery Cycle plugin. This open-source tool will give our users a nice slide show and only requires a browser with JavaScript support.

There are many JavaScript based image cycling plugins out there, but what sets jQuery Cycle apart from the rest is it's ease of use. It has many built-in transition effects, and provides controls for the user to navigate through images. It's well-maintained and has a very active developer community. It's the perfect choice for our slideshow.

Our current home page is somewhat static and boring, so our boss wants us to build a slideshow showcasing the best of our company's photographs. We'll take some sample photographs and build a simple prototype to learn how the jQuery Cycle plugin works.

We'll start by creating a simple home page template that will hold our image slideshow named `index.html`, containing the usual boilerplate code:

2. https://github.com/malsup/cycle

Download image_cycling/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>AwesomeCo</title>
      </head>
      <body>
        <h1>AwesomeCo</h1>
      </body>
    </html>
```

Next,we'll create an images folder and and place a few sample images our boss gave us to use for the slideshow, which you can find in the book's source code folder under the image_cycling folder.

Next, we add jQuery and the jQuery Cycle plugin to our <head> section right below the <title> element. We also need to add a link to a file called rotate.js which will contain all of the JavaScript we'll need to configure our image rotator.

Download image_cycling/index.html

```
<script type="text/javascript"
  src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js">
</script>
<script type="text/javascript"
  src="http://cloud.github.com/downloads/malsup/cycle/jquery.cycle.all.2.74.js">
</script>
<script type="text/javascript" src="rotate.js"></script>
```

Then, we add a with an id of slideshow and add the images inside, like this:

Download image_cycling/index.html

```
<div id="slideshow">
  <img src="images/house-light-slide.jpg" />
  <img src="images/lake-bench-slide.jpg" />
  <img src="images/old-building-slide.jpg" />
  <img src="images/oldbarn-slide.jpg" />
  <img src="images/streetsign-with-highlights-slide.jpg" />
  <img src="images/water-stairs-slide.jpg" />
</div>
```

When we look at our page in the browser we will see something like Figure 3, *These images aren't cycling yet.*, on page 12. This also shows us what our page will look like if the user does not have JavaScript. We see that all of the content is available to the user, so they don't miss out on anything.

We haven't added the functionality to trigger the jQuery Cycle plugin so we just see the images listed in order. Let's add the JavaScript to initialize the

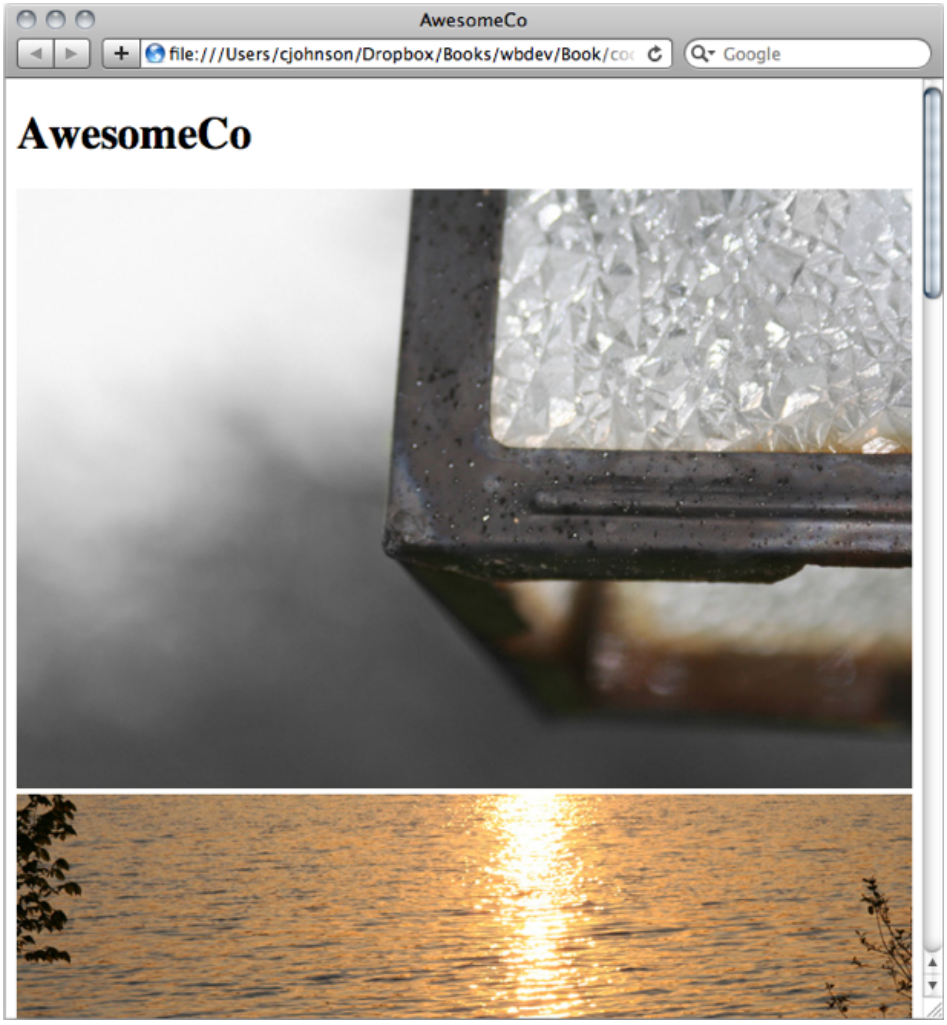


Figure 3—These images aren't cycling yet.

plugin and start the slideshow. Let's create the file rotate.js and add this code, which configures the jQuery Cycle plugin:

Download image_cycling/rotate.js

```
$(function() {
  $('#slideshow').cycle({fx: 'fade'});
});
```

The jQuery Cycle plugin has many different options. We can make the images fade, fade with zooming, wipe or even toss as they transition. You can find

the full list of options on jQuery Cycle's website[3]. Let's stick with the fade function because it is simple and elegant. We defined it with the snippet inside of the cycle() call.

```
fx: 'fade'
```

Now that we have all the pieces in place, lets look at our page again. This time we only see one image and after a few seconds we begin to see the images rotate.

Adding Play and Pause Buttons

We now have a working slideshow. We show our boss the final product and she says "That's great, but I would really like to have a Pause button to let customers pause the slideshow on an image they like". Lucky for us the jQuery Cycle plugin has this ability built right in.

We'll add these buttons to the page with JavaScript, since they're only needed when the slideshow is active. This way, we don't present useless controls to users who don't have JavaScript support. To do this, we'll create two functions: setupButtons() and toggleControls(). The first function will add our buttons to the page and attach click() events to each. The click events will tell the slideshow to either pause or resume. We'll also want the click() events to call toggleControls() which will toggle the buttons so only the relevant one is shown.

Download image_cycling/rotate.js

```
var setupButtons = function(){
  var slideShow = $('#slideshow');

  var pause = $('<span id="pause">Pause</span>');
  pause.click(function() {
   slideShow.cycle('pause');
    toggleControls();
  }).insertAfter(slideShow);

  var resume = $('<span id="resume">Resume</span>');
  resume.click(function() {
    slideShow.cycle('resume');
    toggleControls();
  }).insertAfter(slideShow);

  resume.toggle();
};

var toggleControls = function(){
```

3. http://jquery.malsup.com/cycle/options.html

```
        $('#pause').toggle();
        $('#resume').toggle();
};
```

Notice that we are setting variables to our jQuery selectors. This allows us to manipulate the DOM in a much more efficient manner. We are also taking advantage of the way jQuery returns a jQuery object for almost all methods performed on a jQuery object, so we can chain our insertAfter() function onto our click() binding.

Let's check out the page in the browser again. We can see the "Pause" button show up on the page like in Figure 4, *Our Rotating Images With Controls*, on page 15. Once our slideshow starts, we can click the "Pause" button, and we'll see the "Resume" button replace the "Pause" button as the transitions stop. When we click the "Resume" button, the images will begin to change again.

Further Exploration

This slideshow was easy to implement, and with all of the options that are provided at the plugin's website[4] we can extend the slideshow to include even more functionality.

To enhance the visual experience, the cycle plugin has many transition settings, such as a shuffle, a toss or an uncover transition. We can change our slideshow to use any of these by changing the value of the fx: option in our cycle() call. We can also cycle other elements besides images, including more complex HTML regions. These are just some of the possibilities baked into the Cycle Plugin, so go explore and try them out.

4. http://jquery.malsup.com/cycle/options.html

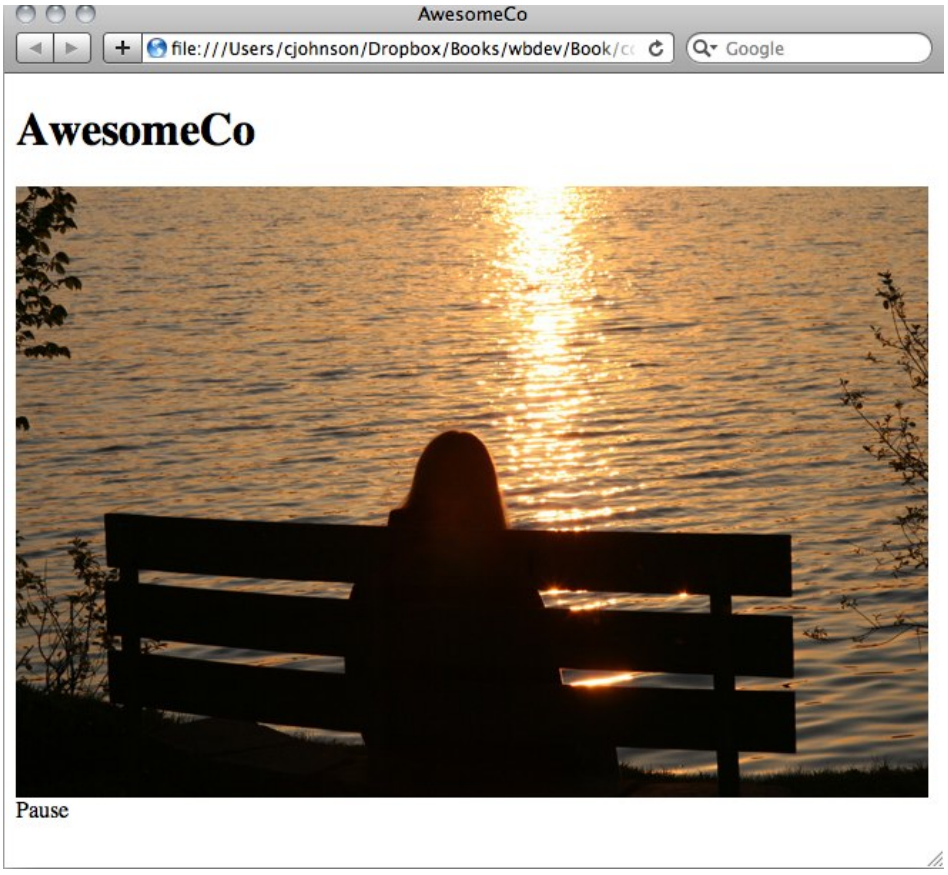


Figure 4—Our Rotating Images With Controls

Recipe 5

Creating and Styling Inline Help Dialogs

Coming soon...

CHAPTER 2

User Interface Recipes

Whether you're delivering static content or presenting an interactive application, you have to create a usable interface. This collection of recipes explores the presentation of information as well as some new ways to build more maintainable and responsive client-side interfaces.

Recipe 6

Creating an HTML Email Template

Problem

Building HTML emails is a bit like traveling back in time. A time before CSS, when everyone used tables for layout and <font> tags reigned supreme. A lot of the best practices we've come to know and love just aren't usable in HTML emails simply because the email readers don't handle them. Testing a web page on multiple browsers is easy compared to the amount of testing we have to do when we create an email that will be read in Outlook, Hotmail, GMail, or Thunderbird, not to mention the various mail applications on mobile devices.

But our job isn't to complain about how difficult things are going to be; our job is to deliver results. And we've got a lot of work to do. Not only do we need to produce readable HTML emails, we need to ensure we don't get flagged as spam.

Ingredients

- A free trial account on Litmus.com for testing emails

Solution

We need to design an HTML template for the invoices our accounting system sends out. The application developers will take this template and handle all of the real content, but we'll need to figure out how to code up the template so that it's readable in all of the popular email clients. We also need to avoid techniques that might get us marked as junk messages, and we need to easily test our email on multiple devices. To do that, let's first look briefly at the constraints we face so we can devise a workable solution.

HTML Email Basics

Conceptually, HTML emails aren't that difficult. After all, creating a simple HTML page is something we can do without much effort. But just like web pages, we can't guarantee that the user will see the same thing we created.

Each email client does something a little different when presenting email messages to its users.

For starters, many web-based clients, like GMail, Hotmail, and Yahoo often strip out or ignore stylesheet definitions from the markup. Google Mail actually removes styles declared in the <style> tag, in an attempt to prevent styles in emails from colliding with the styles it uses to display its interface. We also can't rely on an external stylesheet, because many email clients won't automatically fetch remote files without first prompting the user. So we can't really use CSS for layout in an HTML email.

Google Mail and Yahoo either remove or rename the <body> tag in the email, so it's best to wrap the email in another tag that can stand in for the <body>.

Some clients choke on CSS shorthand declarations, and so any definitions we do use need to be spelled out. For example:

```
#header{padding: 20px;}
```

might be ignored by older clients, so instead, we need to expand it out to the more verbose

```
#header{
  padding-top: 20px;
  padding-right: 20px;
  padding-bottom: 20px;
  padding-left: 20px;
}
```

Desktop clients like Outlook 2007 and Lotus Notes can't handle background images, and Lotus Notes can't display PNG images. That might not seem like a big deal at first, but there are millions of enterprise users who use that as their primary client.

These aren't the only issues we'll run into, but they are the most prevalent. The Email Standards Project[1] has comprehensive lists of issues for the various email clients.

We need to build something that is usable, readable, and effective on multiple platforms, and the best approach is going to be using good old trusty HTML with table-based layouts.

Partying Like It's 1999

When it comes down to it, the most effective HTML emails are designed using the most basic HTML features:

1. http://www.email-standards.org/

- They're built with simple HTML markup with minimal CSS styling.
- They're laid out with HTML tables instead of more modern techniques.
- They don't use intricate typography.
- The CSS styles are extremely simplistic.

In short, we'll need to develop emails as if the last ten years of web development didn't happen. With that in mind, let's code up a very simple email template using tables for layout.

Our invoice will have the typical items that we typically find on an invoice. We'll have a header and footer, as well as sections for our address and the customer's billing address. We'll have a list of the items the customer purchased, and each line will have the price, quantity, and subtotal. We'll need to provide the grand total for the invoice as well, and we'll have an area to display some notes to the customer.

Since some web-based email clients strip out or rename the `<body>` element, we'll need to use our own top-level element to act as the container for our email. To keep it as bullet-proof as possible, we'll create an outer table for the container, and place additional tables inside of that container for the header, footer, and the content. Figure 5, *Our Invoice Mockup*, on page 21 gives a rough example of how we'll mark this up.

Let's start by writing the wrapper for the email template, using an HTML 4.0 doctype:

Download htmlemail/template.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1" http-equiv="content-type">
  <title>Invoice</title>
</head>
<body>
  <center>
    <table id="inv_container"
      width="95%" border="0" cellpadding="0" cellspacing="0">
      <tr>
        <td align="center" valign="top">
        </td>
      </tr>
    </table>
  </center>
</body>
</html>
```

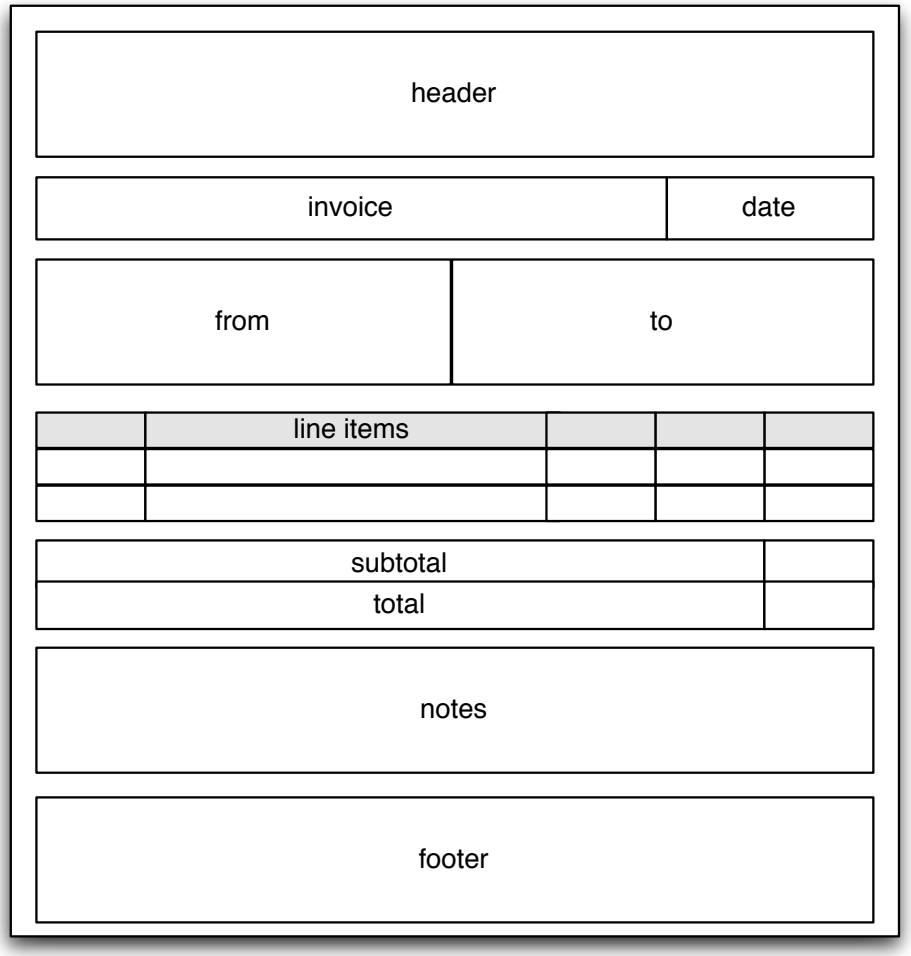


Figure 5—Our Invoice Mockup

To ensure that our invoice shows up centered in the email client, we have to resort to using the very old, and very deprecated tag, since it's the only approach that comes close to working across all of the various clients. Don't worry though; we won't be using <blink>.

Next, we need to create the header. We'll use one table for our company name and a second table with two columns for the invoice number and the date.

Download htmlemail/template.html

```
<table border="0" cellpadding="0" cellspacing="0" width="100%">
```

```
  <tr>
    <td align="center" bgcolor="#5d8eb6" valign="top">
    <h1><font color="white">AwesomeCo</font></h1>
    </td>
  </tr>
</table>

<table border="0" cellpadding="0" cellspacing="0" width="98%">
  <tr>
    <td align="left"  width="70%"><h2>Invoice for Order #533102 </h2></td>
    <td align="right" width="30%"><h3>December 30th, 2011</h3></td>
  </tr>
</table>
```

Since some of the web-based clients strip out CSS, we'll have to use HTML attributes to specify the background and text color. The first table has a width of 100%, but the second table has a width of 98%. Since our tables are centered on the page, this gives us space on the left and right edges so that the text isn't touching the edge of the outer table.

Next, let's add another table that contains the "From" and "To" addresses:

Download htmlemail/template.html

```
<table id="inv_addresses" border="0" cellpadding="2" cellspacing="0" width="98%">
  <tr>
    <td align="left" valign="top" width="50%">
      <h3>From</h3>
      AwesomeCo Inc. <br>
      123 Fake Street <br>
      Chicago, IL 55555
    </td>
    <td align="left" valign="top" width="50%">
      <h3>To</h3>
      GNB <br>
      456 Industry Way <br>
      New York, NY 55555
    </td>
  </tr>
</table>
```

Next, we'll add a table for the invoice itself.

Download htmlemail/template.html

```
<table border="0" cellpadding="2" cellspacing="0" width="98%">
  <caption>Order Summary</caption>
  <tr>
    <th bgcolor="#cccccc" align="left" valign="top">SKU</th>
    <th bgcolor="#cccccc" align="left" valign="top">Item</th>
    <th bgcolor="#cccccc" valign="top">Price</th>
    <th bgcolor="#cccccc" valign="top" width="10%">QTY</th>
    <th bgcolor="#cccccc" valign="top" width="10%">Total</th>
```

```
  </tr>
  <tr>
    <td valign="top">10042</td>
    <td valign="top">15-inch MacBook Pro</td>
    <td align="right" valign="top">$1799.00</td>
    <td align="center" valign="top">1</td>
    <td align="right" valign="top">$1799.00</td>
  </tr>
  <tr>
    <td valign="top">20005</td>
    <td valign="top">Mini-Display Port to VGA Adapter</td>
    <td align="right" valign="top">$19.99</td>
    <td align="center" valign="top">1</td>
    <td align="right" valign="top">$19.99</td>
  </tr>
</table>
```

This is an actual data table, so we'll make sure it has all of the right attributes, like column headers and a caption.

Then we'll add one for the total. We need to use a separate table for this because, believe it or not, some email clients still have trouble displaying tables with rows that span multiple columns.

Download htmlemail/template.html

```
<hr>
<table border="0" cellpadding="2" cellspacing="0" width="98%">
  <tr>
    <td align="right" valign="top">Subtotal: </td>
    <td align="right" valign="top" width="10%">$1818.99</td>
  </tr>
  <tr>
    <td align="right" valign="top">Total Due: </td>
    <td align="right" valign="top"><b>$1818.99</b> </td>
  </tr>
</table>
```

We'll place another simple table to display the invoice notes next.

Download htmlemail/template.html

```
<table border="0" cellpadding="0" cellspacing="0" width="98%">
  <tr><td align="left">
    <h2>Notes</h2>
    <p>Thank you for your business!</p>
  </td></tr>
</table>
```

And finally, we'll add the footer, which we define as a single celled table with full width, just like the header.

Download htmlemail/template.html

```
<table id="inv_footer" border="0" cellpadding="0" cellspacing="0" width="100%">
  <tr>
    <td align="center" valign="top">
    <h4>Copyright &copy; 2011 AwesomeCo</h4>
    <h4>
       You are receiving this email because you purchased
       products from us.
    </h4>
    </td>
  </tr>
</table>
```

The footer is a good place to explain to the user why they got the email in the first place. For an invoice it's pretty obvious, but for a newsletter, we'd use this area to give the reader some links to manage his or her subscription or opt out of future mailings.

With that, we've created a simple, but readable HTML invoice. But what about those clients that can't handle HTML emails?

Supporting The Unsupportable

Not every HTML email client supports HTML email, and as we've learned, even those that do are inconsistent, We should provide a way for people to read the content on those devices, and the most common solution is to provide a link at the top of the message that links to a copy of the email that we host on our servers. When users click the link, they'll be able to read the message in their web browser of choice.

In our case, we can simply place a link to a copy of the invoice within the user's account. We'll want to place a link like that right at the top of the email, above the content table so it's easily visible. As a bonus, some mail programs provide a preview, and this will let them jump right into the invoice without opening the email.

Download htmlemail/template.html

```
<p>
  Unable to view this invoice? <a href="#">View it in your browser instead</a>.
</p>
```

Third-party systems like MailChimp and Campaign Monitor provide this functionality by hosting the HTML email on their servers as static pages.

We could also construct a multipart email, sending both a plain text version of the invoice as well as the HTML version. When we do this, we're actually inserting two bodies into the email and using a special set of headers in the email that tell the email client that the email contains both text and HTML

Joe asks:

Couldn't we just use semantic markup instead of tables?

Many standards-focused developers choose to avoid using tables in favor of semantic markup that relies on CSS to manage the layout. They're not concerned with the mail clients stripping out the CSS because the email will still be readable and accessible.

Unfortunately, if your stakeholders insist that the design of the email must be consistent across clients, standards-based web development techniques won't cut it. That's why we used a table-based approach in this recipe.

versions. To do that effectively, we'd need to develop and maintain a text version of the invoice in addition to our HTML version. Alternatively, we could just place a link to the web page version of the invoice that we're hosting.

Sending multipart emails is beyond the scope of this recipe, but most web-based frameworks and email clients have options for sending out multipart messages. Wikipedia's entry on MIME[2] has a good overview on how multipart messages work.

Styling with CSS

We used tables for layout because we can't rely on floating or absolute positioning with CSS, since many web-based email clients strip out CSS styles. To be honest, those web-based clients aren't stripping things out because their developers are mean-spirited standards-haters. They're doing it because if they allowed CSS, there's potential for the email's contents to conflict with styles in the web-based application.

However, there are two reasons we may still want to try to use CSS to make things look nicer. First, we want things to look nicer for people who actually have email clients that support CSS. But second, we can reuse this invoice template for the static page we talked about in *Supporting The Unsupportable*, on page 24

Since many email clients strip off the `<head>` section of our document, we'll just place our style information in a `<style>`tag right above out container table.

2. http://en.wikipedia.org/wiki/MIME

Let's remove the margins around our heading tags so we can remove some of the wasted space. Let's also apply a background color and a border to our table, and add some space between each of the inner tables, except for the footer, so things aren't so crowded.

Download htmlemail/template.html

```
<style>
  table#inv_addresses h3,
  table#inv_footer h4{
    margin: 0;
  }

  table{
    margin-bottom: 20px;
  }

  table#inv_footer{
    margin-bottom: 0;
  }

  body{
    background-color: #eeeeee;
  }

  table#inv_container{
    background-color: #ffffff;
    border: 1px solid #000000;
  }
</style>
```

With the styles in place, we have an invoice that looks like the one in Figure 6, *Our completed invoice*, on page 27. We're not done though; we need to test things.

Testing Our Emails

Before we can show this off to the client, we need to see how this email works in some email readers. We can send it around a bit to our colleagues or we could create accounts at GMail, Yahoo! Mail, Hotmail, and others to see how things look, but manual testing is time- consuming.

Litmus[3] provides a suite of tools that help people test web pages and emails. They support a wide range of email clients and browsers, including mobile devices. While the service is not free, they do provide a trial account we can use to ensure that our invoices work as expected.

Within a Litmus account, we can create a test that lets us choose the target clients. We can then email our invoice to some addresses Litmus provides,

3. http://litmus.com/

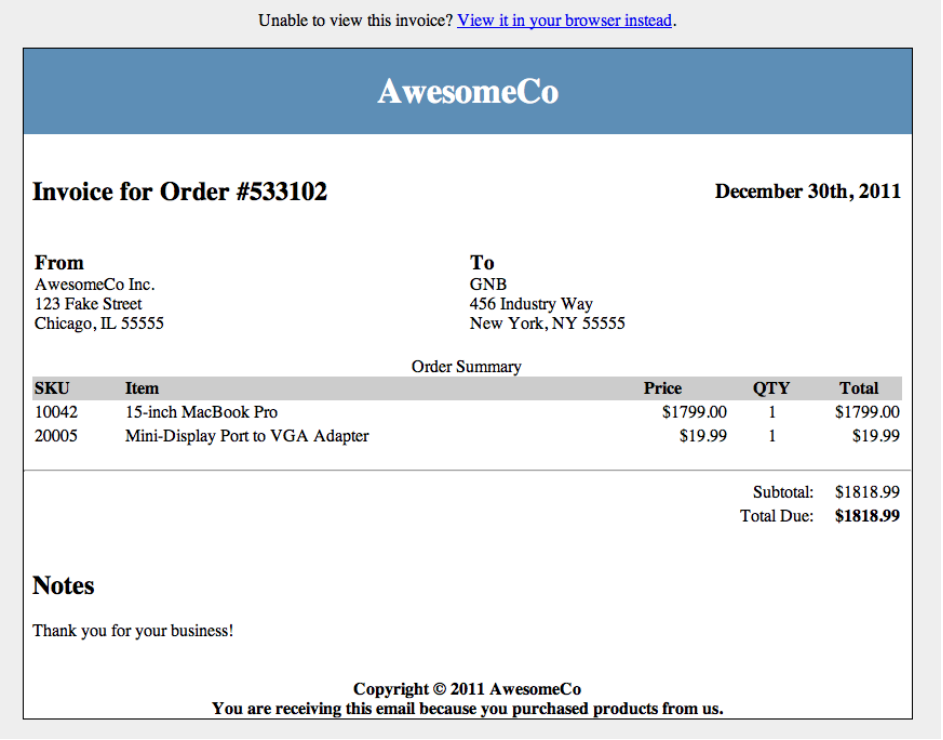


Figure 6—Our completed invoice

or we can just upload our HTML file through the web interface. Using the HTML upload doesn't provide a text fallback so some of the test results will only show the HTML source, not a text fallback, but for our test, it's good enough.

Litmus takes our email, renders it on the target email clients, and provides us with a detailed report that looks like Figure 7, *The results of our test*, on page 28

With the code we've written, it looks like we have an email invoice that looks fairly consistent across the major platforms and looks readable on most of the others.

Images and Emails

We didn't talk about images in this recipe for two reasons. First, we'd need to host our images on a server and include absolute links into the email. The second reason is that most email clients turn images off, since many companies use images to track whether or not the email was opened.

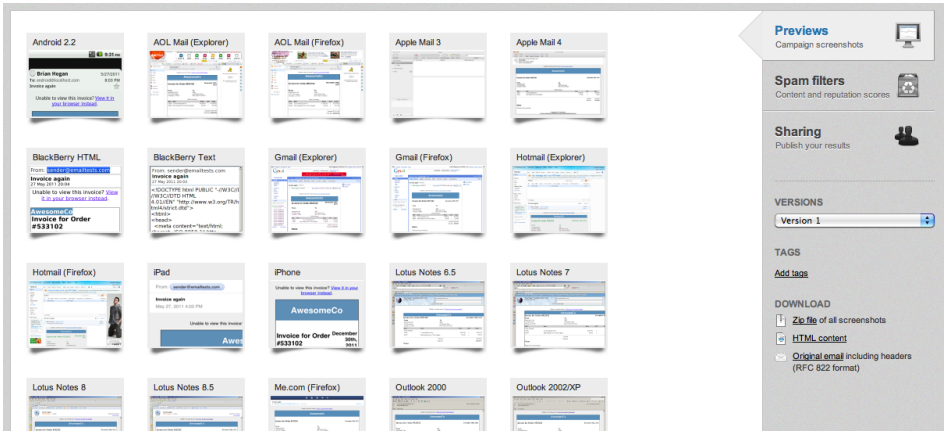


Figure 7—The results of our test

If you do decide to use images in your emails, you'll want to ensure that you follow a few simple rules:

- Be sure to host the images on a server that will be available and don't change the URLs to the images. You never know when someone will open the email you sent.
- Since images are often disabled by default, make sure you specify useful and descriptive `alt` attributes on your images.
- Place the images into your email with regular `<img>` tags. Many email clients don't support images as table-cell backgrounds, and even less support images as CSS backgrounds.
- Because images are often blocked by default, it's a really bad idea to use images as the entire content of your email. It may look really nice but it causes accessibility problems.

Images in emails can be very effective when used properly. Don't be afraid to use them, just be mindful of the issues you will encounter.

Further Exploration

Our simple email template presents a readable invoice to our recipients, but it doesn't have to be as engaging as a marketing announcement or a newsletter might need to be. For that, we'd need to do more styling, more images, and more "exception handling" for various email clients.

MailChimp[4] knows a thing or two about sending emails. After all, that's their business. If you're looking to learn more about email templates, you can dig in to the Email templates they've open-sourced[5]. They're tested on all of the major clients, too, and have some well-commented source code that gives more insight into some of the hacks we have to employ to make things work well across all of the major email clients.

4. http://www.mailchimp.com
5. https://github.com/mailchimp/Email-Blueprints

Recipe 7

Swapping Between Content with Tabbed Interfaces

Problem

We sometimes have multiple, similar pieces of information that we want to display together, such as a phrase in multiple languages, or code examples in several programming languages. We could display them one after another, but that can take up a lot of space, especially with longer content. There has to be an easier way to let our users easily switch and compare without taking up an unnecessary amount of screen space.

Ingredients

- jQuery

Solution

We can use CSS and JavaScript to take the content on our page and turn it in to a slick tabbed interface. Each section of content will have a tab generated for it based on its class, and only one will be displayed at a time. We'll also make sure that we can have any number of tabs that we want, so our design is very flexible. In the end we'll have something that looks like Figure 8, *Our Tabbed Interface*, on page 31

We've been asked to display product descriptions in multiple languages in an attempt to reach a wider audience. We'll build a simple proof-of-concept page so we can determine the best approach going forward.

Building the HTML

Let's start by building out the HTML for the elements we want to show our users. As a proof of concept, let's use two pieces of text, one in English and its Latin translation.

Download swapping/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Swapping Examples</title>
    <script type="text/javascript"
```

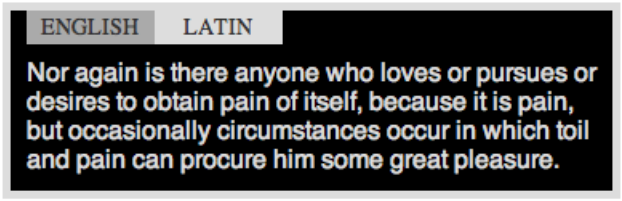


Figure 8—Our Tabbed Interface

```
      src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js">
    </script>
    <link rel="stylesheet" href="swapping.css" type="text/css" media="all" />
    <script type="text/javascript" src="swapping.js"></script>
  </head>
  <body>
    <div class="examples">
      <div class="english example">
        Nor again is there anyone who loves or pursues or desires
        to obtain pain of itself, because it is pain, but occasionally
        circumstances occur in which toil and pain can procure him some
        great pleasure.
      </div>

      <div class="latin example">
        Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
        do eiusmod tempor incididunt ut labore et dolore magna aliqua.
        Ut enim ad minim veniam, quis nostrud exercitation ullamco
        laboris nisi ut aliquip ex ea commodo consequat.
      </div>
    </div>
  </body>
</html>
```

We've set up the basic structure of our elements. There's an examples div that'll hold each of the sections we want to display. Inside that are our example divs that contain the actual content we want users to switch between.

Now, let's get some JavaScript pulled together to create a tabbed interface so our users can toggle between these two examples. We'll use the jQuery library to give us some helper methods and shortcuts.

Creating the Tabbed Interface

First, we need to create a function that will manage calling the different pieces of our JavaScript puzzle. We'll call it styleExamples().

Joe asks:

Couldn't we use jQuery UI Tabs to do this?

Yes, we definitely could, but there's a lot in the UI Tabs that we won't be using, like event hooks. Creating our own tabs lets us focus on keeping things light and gives us greater insight into how things work.

Download swapping/swapping.js

```
function styleExamples(){
   $("div.examples").each(function(){
     createTabs(this);
     activateTabs(this);
     displayTab($(this).children("ul.tabs").children().first());
   });
 }
```

We locate all of the tags that have a class of examples, which will be our containers, and pass each container to a function called createTabs() which creates the tabbed interface our visitors will use to toggle between examples. We'll just cover createTabs() right now, and we'll talk about the rest of the functions soon.

Download swapping/swapping.js

```
function createTabs(container){
  $(container).prepend("<ul class='tabs'></ul>");
  $(container).children("div.example").each(function(){
    var exampleTitle = $(this).attr('class').replace('example','');
    $(container).children("ul.tabs").append(
      "<li class='tab "+exampleTitle+"'>"+exampleTitle+"</li>"
    );
  });
}
```

First, we create an unordered list that will hold our tabs and prepend it to the container that holds the examples.

Then we fetch each of the examples in the container, and loop over them. Our examples have two classes on them: the title of the example, and the example class. We just want the title, so we we grab the class with .attr('class'), then replace 'example' with nothing, and use trim() to remove any extra whitespace that's left over. This gives us the title of each example which we'll display in each tab. We then place the title inside of <li> tags which we append to the unordered list we created initially.

If we open this page in our browser now, nothing will happen because the styleExamples() function isn't being called yet, so none of the JavaScript is being executed. Let's take care of that next.

Switching Between Tabs

Our content is being converted to a tabbed interface, but we don't yet have a way to let our users switch between the different tabs. We'll fix that by first calling styleExamples() when the page loads, which converts the example divs in to our tabbed interface:

Download swapping/swapping.js

```
$(function(){
  styleExamples();
});
```

If we load the page in the browser now, we'll see an unordered list with "english" and "latin" in it. That's great, but it doesn't do much for us yet. Let's write a function that displays between the content of our different examples. First we'll hide all of the examples, then display the one we want to see.

Download swapping/swapping.js

```
 function displayTab(element){
  tabTitle = $(element)
    .attr('class')
    .replace('tab','')
    .replace('selected','').trim();

  container = $(element).parent().parent();
  container.children("div.example").hide();
  container.children("ul.tabs").children("li").removeClass("selected");

  container.children("div."+tabTitle).slideDown('fast');
  $(element).addClass("selected");
}
```

We take the class from tab selected english down to just english and assign it to the variable tabTitle, which we will soon use to find the right to display. Now we want to remove everything from sight.

Download swapping/swapping.js

```
container = $(element).parent().parent();
container.children("div.example").hide();
container.children("ul.tabs").children("li").removeClass("selected");
```

We get the container and hide all of the example divs inside it. We also remove the 'selected' class from every li, even if they don't have one. We do this because it's much simpler to just hit everything at once, rather than inspect-

ing each element, and it increases the readability of the code. Now we're ready to display the requested example.

Download swapping/swapping.js

```
container.children("div."+tabTitle).slideDown('fast');
$(element).addClass("selected");
```

Now we'll access the example that we want to show by finding the div with a class that matches the class passed in to the function, and display it using jQuery's slideDown() functions. We could also try using .show() or .fadeIn(), or getting other animation functions from jQuery UI. Finally, we'll set a class of selected on our current <li> to let our CSS indicate which tab is currently displayed.

Now, we have our displayTab(), but nothing is using it. When a user clicks on one of the example titles we want to switch to that example, so we need to make clicking the s actually call displayTab().

Download swapping/swapping.js

```
 function activateTabs(element){
  $(element).children("ul.tabs").children("li").click(function(){
    displayTab(this);
  });
}
```

This simply takes in our container, locates the s that we created in createTabs(), and sets them to call displayTab() when they're clicked.

Styling the Tabs

Finally, let's go back to styleExamples() where we'll see how all of the functions we've written are called when the page loads, building up our styled examples.

Download swapping/swapping.js

```
function styleExamples(){
   $("div.examples").each(function(){
     createTabs(this);
     activateTabs(this);
     displayTab($(this).children("ul.tabs").children().first());
   });
 }
```

The final call to displayTab() sets the first of the tabs as our default tab, hiding all the rest and displaying it when the page finishes loading.

Now that we have all of the behavior wired up, let's apply a little CSS to it to make it look more like the interface we want.

Download swapping/swapping.css

```
li.tab {
  color: #333;
  cursor: pointer;
  float: left;
  list-style: none outside none;
  margin: 0;
  padding: 0;
  text-align: center;
  text-transform: uppercase;
  width: 80px;
  font-size: 120%;
  line-height: 1.5;
  background-color: #DDD;
}

li.tab.selected {
  background-color: #AAA;
}

ul.tabs {
  font-size: 12px;
  line-height: 1;
  list-style: none outside none;
  margin: 0;
  padding: 0;
  position: absolute;
  right: 20;
  top: 0;
}

div.example {
  font-family: "Helvetica", "san-serif";
  font-size: 16px;

}
div.examples {
  border: 5px solid #DDD;
  font-size: 14px;
  margin-bottom: 20px;
  padding: 10px;
  padding-top: 30px;
  position: relative;
  background-color: #000;
  color: #DDD;
}
```

That's it. We now have some generic code we can use to build out our real site so we can easily switch the product descriptions between different languages.

This solution saves quite a bit of space, and it's something we often see used on sites where space is limited. Some sites use this technique to show product information, reviews, and related items as tabs, while still making that information viewable in a linear format when JavaScript is unavailable.

Further Exploration

What if we wanted to always load a specific tab on the page? For example, if we were displaying code examples in Ruby, Python, and Java and a user of our site wanted to see the Python examples, it'd be nice if they didn't have to click the Python tab every time they visited a new page. We'll leave that up to you to explore on your own.

Also See

- Recipe 8, *Accessible Expand and Collapse*, on page 37

Recipe 8

Accessible Expand and Collapse

Problem

When we need to present long, categorized lists on a website, the best way to do it is with nested, unordered lists. However, when users are presented with this kind of layout it can be hard to quickly navigate, or even comprehend, such a large list. So anything we can do to assist our users will be appreciated. Plus, we want to make sure that our list is accessible in case JavaScript is disabled or a user is visiting our site with a screen-reader.

Ingredients

- jQuery
- jQuery Template

Solution

A relatively easy way to organize a nested list, without separating the categories into separate pages, is to make the list collapsible. This means that entire sections of the list can be hidden or displayed to better convey selective information. At the same time, the user can easily manipulate which content should be visible.

For our example, we'll start with an unordered list that displays our products grouped by sub-categories.

Download collapsiblelist/index.html

```
<h1>Categorized Products</h1>
<ul class='collapsible'>
  <li>
    Music Players
    <ul>
      <li>16 Gb MP3 player</li>
      <li>32 Gb MP3 player</li>
      <li>64 Gb MP3 player</li>
    </ul>
  </li>
  <li class='expanded'>
    Cameras & Camcorders
```

```
    <ul>
      <li>
        SLR
        <ul>
          <li>D2000</li>
          <li>D2100</li>
        </ul>
      </li>
      <li class='expanded'>
        Point and Shoot
        <ul>
          <li>G6</li>
          <li>G12</li>
          <li>CS240</li>
          <li>L120</li>
        </ul>
      </li>
      <li>
        Camcorders
        <ul>
          <li>HD Cam</li>
          <li>HDR-150</li>
          <li>Standard Def Cam</li>
        </ul>
      </li>
    </ul>
  </li>
</ul>
```

We'll want to be able to indicate that some of the nodes should be collapsed or expanded from the start. It would be tempting to simply mark the collapsed nodes by setting the style to display: none. But that would break accessibility since screen-readers ignore content hidden like this. Instead we're going to rely on JavaScript to toggle each node's visibility at runtime. We did this by adding a CSS class of "expanded" to set the initial state of the list.

If we knew the user wanted to look at "Point and Shoot Cameras" when they first reached this page, for example, this markup wouldn't show the limited list yet. Right now it will show the full categorized product list as seen in Figure 9, *Our Full Categorized List*, on page 39. But once the list is made collapsible, the user would only see the names of the products they were looking for, as shown in Figure 10, *Our Collapsed List*, on page 39. Then, without navigating away from the page, they can still choose to look at any of our other product categories.

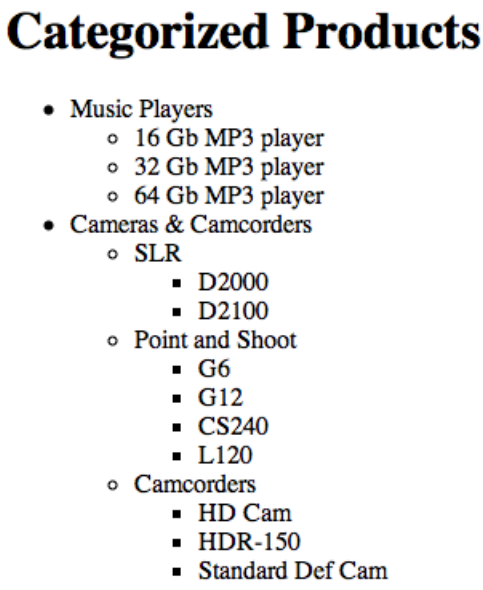


Figure 9—Our Full Categorized List

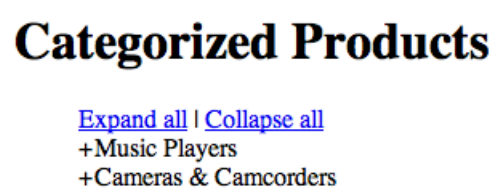


Figure 10—Our Collapsed List

Next we need to write the JavaScript for adding our collapsible functionality, as well as some Expand all and Collapse all helper links at the top of the list. Notice that we're adding these links via the JavaScript code as well. Like the collapsible functionality itself, we don't want to change the markup unless we know this code is going to be used. This also gives us the advantage of being able to easily apply this behavior to any list on our site without having to change any markup beyond adding a .collapsible class to a <ul> element.

To start things off we will write a function that toggles whether a node is expanded or collapsed. Since this is a function that will act on a DOM object, we will write it as a jQuery plugin. That means we will assign the function

definition to the jQuery.fn prototype. We can then trigger the function within the scope of the element that it was called against. The function definition should be wrapped within a self executing function so we can use the $ helper without worrying about whether or not the $ helper has been been overwritten by another framework. Finally to make sure that our jQuery function is chainable and a responsible jQuery citizen, we return this. This is a good practice to follow when writing jQuery plugins; *our* plugin functions will work the same way that we expect other jQuery plugins to work.

Download collapsiblelist/javascript.js

```
(function($) {
  $.fn.toggleExpandCollapse = function(event) {
    event.stopPropagation()
    if (this.find('ul').length > 0) {
      event.preventDefault()

      this.toggleClass('collapsed').toggleClass('expanded').
        find('> ul').slideToggle('normal')
    }

    return this
  }
})(jQuery)
```

We will bind the toggleExpandCollapse() to the click event for all <li> elements, including the elements with nothing underneath them, also known as *leaf nodes*. That's because we want the leaf nodes to do something crucial; absolutely nothing. Unhandled click events bubble up the DOM so if we only attach a click observer to the <li> elements with .expanded or .collapsed classes, the click event for a leaf node would bubble up to the parent <li> element, which *is* one of our collapsible nodes. That means the code would trigger that node's click event, which would make it collapse suddenly and unexpectedly, and we'd be liable for causing undue harm to our users' fragile psyches. To prevent this Rube Goldberg-styled catastrophe from happening we call event.stopPropagation(). Adding an event handler to all <li> elements ensures the click event will never bubble up and nothing will happen, just like we expect. For more details on event propogation, read *Why not just return false?*, on page 41.

As mentioned at the beginning of the chapter, we want to give our users helper links that appear at the top of the list to toggle all of the nodes. We can create these links within jQuery and prepend them to our collapsible list. Because building HTML in jQuery can become verbose, we're better off moving the click event logic into separate helpers to prevent the prependToggleAllLinks() functions from becoming unreadable.

Joe asks:
Why not just return false?

In a jQuery function return false works double duty by telling the event not to bubble up the DOM tree and not to do whatever the element's default action is. This works for most events, but there are times where we want to make the distinction between stopping event propogation and preventing a default action from trigger. Or we may be in a situation where we always want prevent the default action, even if the code in our function somehow breaks. That's why at times it may make more sense to call event.stopPropagation() or event.preventDefault() explicitly rather than waiting until the end of the function to return false [a].

a. http://api.jquery.com/category/events/event-object/

Download collapsiblelist/javascript.js

```
function prependToggleAllLinks() {
  var container = $('<div>').attr('class', 'expand_or_collapse_all')
  container.append(
      $('<a>').attr('href', '#').
      html('Expand all').click(handleExpandAll)
    ).
    append(' | ').
    append(
      $('<a>').attr('href', '#').
      html('Collapse all').click(handleCollapseAll)
    )
  $('ul.collapsible').prepend(container)
}

function handleExpandAll(event) {
  $('ul.collapsible li.collapsed').toggleExpandCollapse(event)
}

function handleCollapseAll(event) {
  $('ul.collapsible li.expanded').toggleExpandCollapse(event)
}
```

We can quickly create a DOM object by wrapping a string representing the element type we want, in this case an <a> tag, in a jQuery element. Then we set the attributes and HTML through jQuery's API. For simplicity's sake we're going to create two links that say "Expand all" and "Collapse all" that are separated by a pipe symbol. The two links will trigger their corresponding helper functions when they're clicked.

Finally we will write an initialize function that gets called once the page is ready. This function will also hide any nodes that were not marked as .expanded, and add the .collapsed class to the rest of the <li> elements.

Download collapsiblelist/javascript.js

```
function initializeCollapsibleList() {
  $('ul.collapsible li').click(function(event) {
    $(this).toggleExpandCollapse(event)
  })
  $('ul.collapsible li:not(.expanded) > ul').hide()
  $('ul.collapsible li ul').
    parent(':not(.expanded)').
    addClass('collapsed')
}
```

We bind the click event to all of the <li> elements that are in a .collapsible list. We also added the expand/collapse classes to all of the <li> elements, except the products themselves. These classes will help us when it comes time to style our list.

When the DOM is ready we'll tie it all together by initializing the list and adding the "Expand all"|"Collapse all" links to the page.

Download collapsiblelist/javascript.js

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

Since this is a jQuery plugin, we can easily add this functionality to any list on our site by adding a .collapsible class to an unordered list. This makes the code easily reusable so that any long and cluttered list can be made easy to navigate and understand.

Further Exploration

If we start out by building a solid, working foundation without JavaScript, we can build upon that foundation to add in extra behavior. And if we write the JavaScript and connect the behavior into the page using CSS classes rather than adding the JavaScript directly to the HTML itself, everyhing is completely decoupled. This also keeps our sites from becoming too JavaScript dependent, which means more people can use your sites when JavaScript isn't available. We call this *progressive enhancement* and it's an approach we strongly recommend.

When building photo galleries, make each thumbnail link to a larger version of the image that opens on its own page. Then use JavaScript to intercept

the click event on the image and display the full-sized image in a lightbox, and then use JavaScript to add any additional controls that are only useful when JavaScript is enabled, just like we did in this recipe.

When you're building a form that inserts records and updates the values on the screen, create the form with a regular HTTP POST request first, and then intercept the form's submit event with JavaScript and do the post via AJAX. This sounds like more work, but you end up saving a lot of time; you get to leverage the form's semantic markup and use things like jQuery's serialize() method to prepare the form data, rather than reading each input field and constructing your own POST request.

Techniques like this are well-supported by jQuery and other modern libraries because they make it easy to build simple, accessible solutions for your audience.

Also See

- Recipe 9, *Interacting with web pages using Keyboard Shortcuts*, on page 44
- Recipe 11, *Displaying Information with Endless Pagination*, on page 58

Recipe 9

Interacting with web pages using Keyboard Shortcuts

Problem

Web site visitors expect to use their mouse to interact with a web site, but using the mouse isn't always the most efficient way. Keyboard shortcuts are becoming increasingly common on sites like GMail and Tumblr as a way to improve accessibility and allow users to quickly and comfortably perform common tasks. We want to bring this functionality to our site, but we need to make sure we don't interfere with the normal expected behavior of our application, like our search box.

Ingredients

- jQuery

Solution

Keyboard shortcuts use JavaScript to monitor the page for certain keys to be pressed by binding a function to the document's keydown event. When a key is pressed we check if it's one of the keys we are using for a shortcut, then call the specified function for that key.

We have a site with a large number of blog entries on it on a variety of topics. After some usability testing we saw that users decide if they want to read the entry by scanning the title and part of the first sentence. If they're not interested, they scroll on to the next article. Because some entries are very long, users end up doing a lot of scrolling to get to the next article. We'd like to create some basic shortcuts that will let users move easily between the entries on the page, navigate between pages, and quickly use the search box. We'll work with an interface that looks like the one in Figure 11, *A basic page with a search box and multiple entries*, on page 45.

Getting Set Up

First we will add the ability to scroll between entries on the current page. We'll start by creating a page containing several items that all share a class of `entry` and use the "j" key to go to the previous record and "k" to go to the

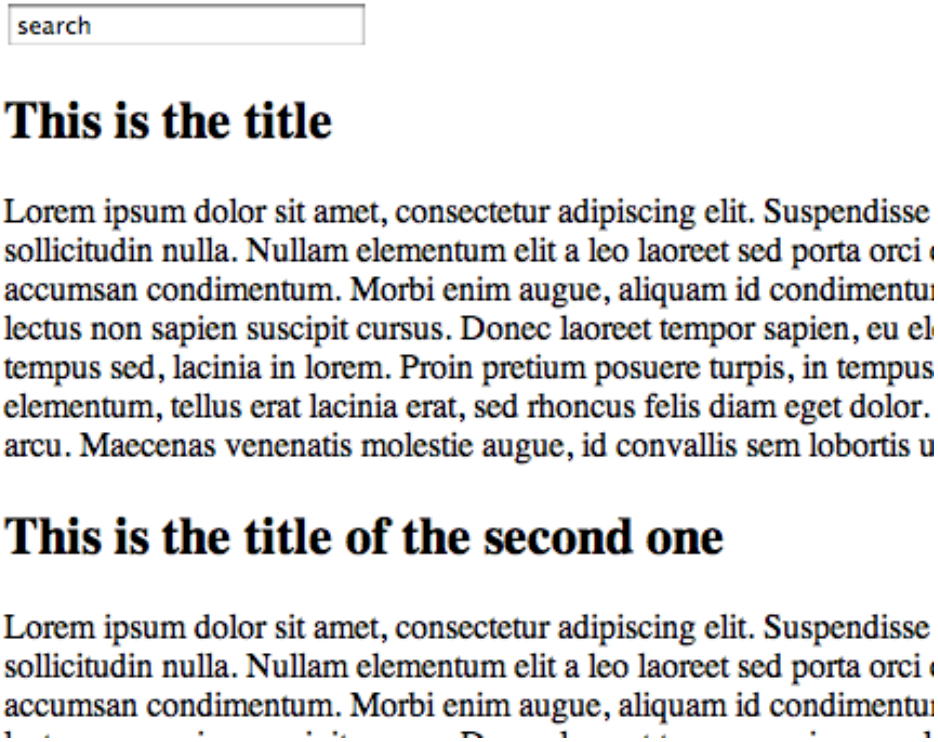


Figure 11—A basic page with a search box and multiple entries

next one. These letters are commonly used for previous and next records on many applications, so it's not a bad idea to follow the convention. After that we'll handle navigating between pages using the right and left arrows, followed by creating a shortcut to use the search box.

Let's start by creating a prototype that has a search box and a few search results so we have something we can test our keyboard navigation on.

Download keyboardnavigation/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript"
      src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js">
    </script>
    <script type="text/javascript"
      src='keyboard_navigation.js'></script>
  </head>
  <body>
    <p>Make this page longer so you can tell that we're scrolling!</p>
```

```
    <form>
      <input id="search" type="text"size="28" value="search">
    </form>
    <div id="entry_1" class="entry">
      <h2>This is the title</h2>
      <p>Lorem ipsum dolor sit amet...</p>
    </div>
    <div id="entry_2" class="entry">
      <h2>This is the title of the second one</h2>
      <p>In hac habitasse platea dictumst...</p>
    </div>
  </body>
</html>
```

Due to size constraints this example page is very short. To see the full effect as we scroll between elements on the page, add a few more sections. Make sure the content is longer than your browser can display at once so that you can see the effect of scrolling between entries.

Catching Key Presses

We'll use jQuery to set up a few event handlers when the page loads. When someone presses one of our navigation keys we'll call the functions that navigate through the page. The $(document).keydown() method allows us to specify exactly what to call for different keys by using a case statement. Each case we define represents a different key by its key code.[7]

Download keyboardnavigation/keyboard_navigation.js

```
$(document).keydown(function(e) {
  if($(document.activeElement)[0] == $(document.body)[0]){
    switch(e.keyCode){
    // In Page Navigation
    case 74: // j
      scrollToNext();
      break;
    case 75: // k
      scrollToPrevious();
      break;
    // Between Page Navigation
    case 39: // right arrow
      loadNextPage();
      break;
    case 37: // left arrow
      loadPreviousPage();
      break;
```

7. To find other key codes check out the list at http://www.cambiaresearch.com/c4/702b8cd1-e5b0-42e6-83ac-25f0306e3e25/javascript-char-codes-key-codes.aspx

```
    // Search
    case 191: // / (and ? with shift)
      if(e.shiftKey){
        $('#search').focus().val('');
        return false;
      }
      break;
    }
  }
});
```

Before we check if one of our keys is pressed, it's important to make sure we're not interrupting normal user activity. The first line of our keydown function is `if($( document.activeElement)[0] == $(document.body)[0])`, which makes sure that the active element on the page is the body of the page itself. By doing this we avoid catching key presses when our user is typing in to a search box or a text area.

Scrolling

Scrolling between entries on our page involves getting a list of the current entries and knowing which one we last scrolled to. First we want to set everything so that when we first scroll on the page we go to the first entry on the page.

Download keyboardnavigation/keyboard_navigation.js

```
$(function(){
  current_entry = -1;
});
```

When the page loads, we set a variable called `current_entry` to `-1`, meaning that we haven't scrolled anywhere yet. We use `-1` because we are going to figure out which entry to display by loading all objects on the page with a class of `.entry` and picking the correct one based on its index in the resulting array. JavaScript arrays are zero-based, so the first entry will be at the 0 position.

Download keyboardnavigation/keyboard_navigation.js

```
$(function(){
  current_entry = -1;
});
```

In the Catching Key Presses section we defined the functions to call when certain keys were pressed. When the "j" key is press we want to scroll to the next entry on the page, so we call the `scrollToNext()` function.

Download keyboardnavigation/keyboard_navigation.js

```
function scrollToNext(){
  if($('.entry').size() > current_entry+1){
    current_entry++;
```

```
    scrollTo(current_entry);
  }
}
```

In scrollToNext() we first check that we're not trying to scroll to an entry that doesn't exist by ensuring that incrementing the current_entry counter won't be larger than the number of entries on the page. If there's an entry to scroll to, we increase the current_entry by 1 and call scrollToEntry().

Download keyboardnavigation/keyboard_navigation.js

```
function scrollToEntry(entry_index){
  $('html,body').animate(
    {scrollTop: $("#"+$('.entry')[entry_index].id).offset().top},'slow');
}
```

scrollToEntry() uses the jQuery animation libraries to scroll our view to the ID of the specified entry. Since the current_entry represents the index of the entry we want to display, we grab the ID of that entry and tell jQuery to scroll there.

When the user presses the "k" key, we call a similar function called scrollToPrevious().

Download keyboardnavigation/keyboard_navigation.js

```
function scrollToPrevious(){
  if(current_entry > 0){
    current_entry--;
    scrollTo(current_entry);
  }
}
```

scrollToPrevious() makes sure that we aren't trying to load a smaller entry than 0, since that will always be the first entry on the page. If we're not on the first entry, then we reduce the current_entry by 1 and once again call scrollToEntry().

Now that our users have the ability to scroll between entries on the page, it can be very easy to quickly review the content of the page. But once they get to the end of the page, they'll need to be able to move to the next page of records. Let's work on that next.

Pagination

Navigating between pages can happen in a variety of ways. For this example we'll assume that the desired page is indicated by the page=1 querystring in the URL, however this could easily be changed to work with p=1, entries/2 or anything else you might encounter.

To keep our code nice and clean, let's write a function called getQueryString() that pulls the page number out of the URL.

Download keyboardnavigation/keyboard_navigation.js

```
function getQueryString(name){
  var reg = new RegExp("(^|&)"+ name +"=([^&]*)(&|$)");
  var r = window.location.search.substr(1).match(reg);
  if (r!=null) return unescape(r[2]); return null;
}
```

Now, let's build a getCurrentPageNumber() function that uses the getQueryString() function to check if page exists. If it does, we get it and turn it from a string to an integer, then return it. If it doesn't exist that means that no page is currently set. If this is the case we'll assume we're on the first page and return 1. It's important that we return an integer and not a string, because we're going to need to do math with the page number.

Download keyboardnavigation/keyboard_navigation.js

```
function getCurrentPageNumber(){
  return (getQueryString('page') != null) ? parseInt(getQueryString('page')) : 1;
}
```

Our keycode watcher is listening for the left and right arrows. When the user presses the right arrow, we call the loadNextPage() function, which figures out what page number we're on and directs the browser to the next one.

Download keyboardnavigation/keyboard_navigation.js

```
function loadNextPage(){
  page_number = getCurrentPageNumber()+1;
  url = window.location.href;
  if (url.indexOf('page=') != -1){
    window.location.href = replacePageNumber(page_number);
  } else if(url.indexOf('?') != -1){
    window.location.href += "&page="+page_number;
  } else {
    window.location.href += "?page="+page_number;
  }
}
```

We first determine our current page number, and then we increase page_number by 1 since we're going to the next page. Then we grab the current URL so we can update it and load the next page. This is the most involved part of the process because there are several ways the URL could be structured. First we check if the URL contains page=. If it does, as in http://example.com?page=4, then we just need to replace the current number using a regular expression and the replace() function. Since we'll need to replace the page number when going to the previous page, we have a replacePageNum-

ber() function so if our URL structure changes we only have to update our code in one place.

Download keyboardnavigation/keyboard_navigation.js

```
function replacePageNumber(page_number){
  return window.location.href.replace(/page=(\d)/,'page='+page_number);
}
```

If the URL doesn't contain page= then we need to add the entire parameter to the querystring. Next,we check if the URL contains other parameters. If it does they'll be listed after the "?" in the URL, so we check for "?". If it exists, as in http://example.com?foo=bar then the page number will be added to the end. Otherwise we need to create the querystring ourselves, which is done in the final else of the if... else block.

We use a similar, though simpler, technique to load the previous page. After figuring out the current page number and reducing it by 1, we just need to make sure that we're not trying to load a page number that is less than 1. So we check if the new page_number is greater than 0. If it is, we update page= with the new number and we're on our way.

Download keyboardnavigation/keyboard_navigation.js

```
function loadPreviousPage(){
  page_number = getCurrentPageNumber()-1;
  if(page_number > 0){
    window.location.href = replacePageNumber(page_number);
  }
}
```

Now that we can move between pages and amongst entries let's create a way for users to quickly get access to the search box.

Navigating to the Search Box

The keyboard shortcut that makes the most sense for this is the "?" key, but that's done by pressing two keys, so we need to do things a little bit differently than our other shortcuts. First, we watch for the keycode of 191, which represents the "/" key. When this is pressed we call the shiftKey property on the event, which will return true if the Shift key is down.

Download keyboardnavigation/keyboard_navigation.js

```
case 191: // / (and ? with shift)
  if(e.shiftKey){
    $('#search').focus().val('');
    return false;
  }
  break;
}
```

If the Shift key was pressed, we retrieve the search box by using its DOM ID and call the `focus()` method to place the cursor inside the search box. We then erase any content current in it by calling `val('')`. Finally we call `return false;`, which prevents the "?" that was typed from being placed in to the search box.

Further Exploration

We've added some quick keyboard shortcuts that let our users navigate throughout our site without having to take their hands off of their keyboards. Once the framework is in place, adding new keyboard shortcuts is a breeze. You could use keyboard shortcuts to display a lightbox on a page that opens when the user presses the space bar. You could use keyboard shortcuts to pop up a console with information about ongoing tasks, or use them to reveal further content in a blog post.

Many of the other JavaScript based chapters in this book could have keyboard shortcuts added to them, such as browsing through the images in Recipe 4, *Creating Interactive Slideshows With jQuery*, on page 10 using the keyboard or scanning and expanding items in Recipe 8, *Accessible Expand and Collapse*, on page 37

Also See

- Recipe 4, *Creating Interactive Slideshows With jQuery*, on page 10
- Recipe 8, *Accessible Expand and Collapse*, on page 37
- Recipe 29, *Cleaner JavaScript with CoffeeScript*, on page 135

Recipe 10

Building HTML With jQuery Templates

Problem

Amazing interfaces require creating lots of dynamic and asynchronous HTML. Thanks to AJAX and JavaScript libraries like jQuery, we can change the user interface without reloading the page by generating HTML with JavaScript. Typical methods like string concatenation are hard to manage and are prone to error. We have to dance around mixing single and double quotes and often are left to use jQuery's `append()` method endlessly. Thankfully, new JavaScript plugins such as jQuery Templates allow us to write *real* HTML, render data with it, and insert it into the document.

Ingredients

- jQuery
- jQuery Templates[8]

Solution

The *jQuery Templates* plugin is a JavaScript template engine that is easy to implement. It works with existing applications using jQuery, and it lets us write client-side views with clean HTML that are abstracted away from the JavaScript code. It allows for conditional logic and builds off the existing `jQuery.each` method. Because we already use jQuery for many of our JavaScript applications, it makes sense to have a template language that is directly integrated.

With jQuery Templates, we can simplify HTML creation when generating new content. We will explore jQuery templates by working with a JavaScript-driven product management application.

The existing application lets us manage products by adding new ones to a list. The example uses our standard development server since the requests are all handled by JavaScript and AJAX. When the user fills in the form to

8. http://api.jquery.com/category/plugins/templates/

add a new product, it asks the server to save the product then renders a new product in the list. To add the product to the list, we have to use string concatenation which becomes awkward and hard to read, like this:

Download jquerytmpl/submit.html

```
var newProduct = $('<li></li>')
newProduct.append('<span class="product-name">' +
  product.name + '</span>')
newProduct.append('<em class="product-price">' +
  product.price + '</em>')
newProduct.append('<div class="product-description">' +
  product.description + '</div>')

productsList.append(newProduct)
```

Using jQuery templates is as easy as loading the script on the page. It relies on jQuery, so it must be loaded after jQuery has been loaded. We'll first want to download it from the plugin's website[9] or the book's code repository.

Rendering a Template

To refactor our existing application, we first need to learn how to render a template using jQuery Templates. The most simple way to is to make a call to the tmpl() function.

```
$.tmpl(templateString, data)
```

The function accepts two arguments; the first argument is a string of template HTML to be rendered against, and the second argument is the data to be injected into the HTML. The data variable is an object whose keys become the local variables in the template. Examine the following code:

```
var artist = {name: "John Coltrane"}
var rendered = $.tmpl('<span class="artist name">${ name }</span>', artist)
$('body').append(rendered)
```

The rendered variable contains our final HTML that has been spit back out from the tmpl method. To place the name property in our HTML, jQuery Templates uses a style of template tags that begin with the dollar sign. Inside the curly braces, we place the name of a property. The last line appends the rendered HTML to the <body>.

This is the simplest method for rendering a template with jQuery Templates. In our application, there will be more code related to sending a request to a server to retrieve the data, but the process for creating the template will be the same.

9. http://api.jquery.com/category/plugins/templates/

Replacing an Existing System

Now that we understand how to render a template, we can remove the old method of string concatenation from the existing application. Let's examine the old code to see what can be removed and what needs replacing.

Download jquerytmpl/submit.html

```
function renderNewProduct() {
  var productsList = $('#products-list')

  var newProductForm = $('#new-product-form')

  var product = {}
  product.name = newProductForm.find('input[name*=name]').val()
  product.price = newProductForm.find('input[name*=price]').val()
  product.description = newProductForm.find('textarea[name*=description]').val()

  var newProduct = $('<li></li>')
  newProduct.append('<span class="product-name">' +
    product.name + '</span>')
  newProduct.append('<em class="product-price">' +
    product.price + '</em>')
  newProduct.append('<div class="product-description">' +
    product.description + '</div>')

  productsList.append(newProduct)

  productsList.find('input[type=text], textarea').each(function(input) {
    input.attr('value', '')
  })
}
```

That messy code is a headache to read and even worse to maintain. Instead of using jQuery's append method to build up the HTML, let's use jQuery templates to render the HTML. We can write real HTML and render the data using jQuery Templates! Our first step to reducing the JavaScript madness is to build our template. Then, we'll render it with our product data in one simple step.

If we create a <script> element with a content type of text/html, then we can place jQuery Templates HTML inside of that element and then pull it out for our template. We'll give it an ID so that we can reference it in our JavaScript code with jQuery.

```
<script type="text/html" id="product-template">
  <!-- template HTML -->
</script>
```

Next, let's write the HTML for our template. We already have the product in object form, so we can use its properties as the variable names in our template, like this:

Joe asks:
Can I use external templates?

Inline templates are handy, but we want to remove the template logic from the server views. On our server, we would create a folder to hold all of our view files. Then, when we wanted to render one of the templates, we make a GET request with jQuery and fetch the template.

```
$.get("http://mysite.com/js_views/external_template.html",
  function(template) {
    $.tmpl(template, data).appendTo("body")
  }
)
```

This allows us to serve views separate from our client views.

```
<script type="text/html" id="product-template">
  <li>
    <span class="product-name">${ name }</span>
    <em class="product-price">${ price }</em>
    <div class="product-description">${ description }</div>
  </li>
</script>
```

With our template in place, we can go back to our previous code and rewrite how we're inserting the HTML. We can call the tmpl() method on any jQuery object. All we need to do is grab the template with jQuery and call the tmpl() method on it, passing our data.

```
var newProduct = $('#product-template').tmpl(product)
```

When we look at the results, things look pretty good, but we don't really need to show the description field if there's no description coming back from the server. We don't want to render the corresponding if the description isn't present. Thankfully, jQuery Templates allows for conditional statements. We can write an if statement to conditionally render the .

```
{{if description}}
  <div class="product-description">${ description }</div>
{{/if}}
```

Conditional statements for if else and else are also available. There is also an each operator that will iterate over an array for you, rendering HTML for each item in the array.

Using Iteration

Since we have been able to replace much of the existing code for building a new product, we have decided to make more of the application work using JavaScript. We want to replace the index page that shows products and their notes with some JavaScript code that does the same rendering. We will pass an array of products to the tmpl() method, and each product will have a notes property. The notes is an array that will be iterated over inside the template.

First, let's get the products and render them, assuming our sever returns a JSON array that looks like this:

Download jquerytmpl/index.html

```
$.getJSON('/products.json', function(data) {
  $('products-template').tmpl(data).appendTo('body')
})
```

Now, we need to build a template to render the products. Since jQuery Templates is already iterating through each product in the array, we don't need to do that inside the template. We do, however, want to iterate over the notes for each product and display them.

Download jquerytmpl/index.html

```
<script type="text/html" id="products-template">
  <li>
    <span class="product-name">${ $data.name }</span>
    <em class="product-price">${ $data.price }</em>
    <div class="product-description">${ $data.description }</div>
    <ul class="product-notes">
      {{each(index, note) $data.notes}}
        <li>note.text</li>
      {{/each}}
    </ul>
  </li>
</script>
```

When we pass an array to the tmpl() method, we have lost any creation of local variables in the template. In this case, we use the special $data variable which represents the current object when iterating.

JavaScript templates are a nice way to improve the organization of a JavaScript application. We learned how to render templates, use conditional logic, employ jQuery's each() each method, and render multiple objects with jQuery Templates implicit iteration. The jQuery Templates plugin is a simple way to remove string concatenation and build HTML in a semantic and readable way.

Further Exploration

The benefit to using jQuery templates is that we already use jQuery for most of our JavaScript code. However, for projects that don't use jQuery, we recommend using a standalone JavaScript template engine. The syntax for these is nearly the same as jQuery Templates, and they are as easy to use. We recommend using Mustache.js[10]. It's a great alternative to jQuery templates and will remove the jQuery dependancy.

Also See

- Recipe 8, *Accessible Expand and Collapse*, on page 37
- Recipe 11, *Displaying Information with Endless Pagination*, on page 58
- Recipe 13, *Snappier Cient-Side Interfaces with Knockout.js*, on page 69
- Recipe 14, *Organizing Code with Backbone.JS*, on page 79
- Recipe 20, *Building a Status Site with JavaScript and CouchDB*, on page 116

10. http://mustache.github.com/

Recipe 11

Displaying Information with Endless Pagination

Problem

To prevent information overload for our users, and to keep our servers from grinding to a halt, it's important to limit how much data is shown at once on our list pages. This is traditionally handled by adding pagination to these pages. That is, we only show a small subset of data to start with, while allowing the users to jump between the pages of information at their own discretion. What they see is a small part of all of the information that is potentially available to them.

As websites have evolved, web developers have learned that the majority of the time users go through these pages sequentially. They would actually be happy to scroll through an entire list of data until they found what they were looking for, or they reached the end of the dataset. We can provide that experience with *endless pagination*.

Ingredients

- jQuery
- jQuery Templates[11]
- QEDServer

Solution

By implementing endless pagination, we can provide an efficient way of managing our resources, while at the same time improving the end-user experience. Instead of forcing users to choose the next page of results and then reloading the entire interface, we load the next page of results in the background and add those results to the current page as the user scrolls toward the end of the page.

We want to add a list of our product line to our site but our inventory is much too big to reasonably load all at once. This means that we're going to

11. http://api.jquery.com/category/plugins/templates/

have to add pagination for this list and limit the user to seeing 10 products at a time. To make our users' lives even easier we're going to ditch the "Next Page" button and automatically load the following page when we think they're ready for it. From the user's perspective it will seem as if the entire product list has been available to them since they first loaded the page.

We'll use QEDServer and its product catalog to build a working prototype. We'll place all of our code in the public folder in QEDServer's workspace. Start up QEDServer and then create a new file called products.html in the public folder that QEDServer creates. You can look at *QEDServer*, on page iv for details on how QEDServer works.

To keep our code clean, we'll use the jQuery Templates library, which we discuss in Recipe 10, *Building HTML With jQuery Templates*, on page 52, so we'll download that and place it in the public folder as well.

We'll start out by creating a simple HTML5 skeleton in index.html that includes jQuery, the jQuery Templates library, and endless_pagination.js, which we'll create to hold our pagination code.

Download endlesspagination/products.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='utf-8'>
    <title>AwesomeCo Products</title>
    <link rel='stylesheet' href='/endless_pagination.css'>
    <script type="text/javascript"
      src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js">
    </script>
    <script type="text/javascript" src="jquery.tmpl.min.js"></script>
    <script src="/endless_pagination.js"></script>
  </head>
  <body>
    <div id="wrap">
      <header>
        <h1>Products</h1>
      </header>
    </div>
  </body>
</html>
```

For the body of this initial page, we add a content placeholder and a spinner image. The spinner is there so if the user ever *does* reach the end of the current page, it will appear as if the next page is already loading, which it should be.

Download endlesspagination/products.html

```
<div id='content'>
</div>
<img src='spinner.gif' id='next_page_spinner' />
```

QEDServer's API is set up to return paginated results and responds to JSON requests. We can see this by navigating to http://localhost:8080/products.json?page=2.

Now that we know what information we're getting from the server, we can start building the code that will update the interface by writing a function that takes in a JSON array, marks it up using a jQuery template and appends it to the end of the page. We'll put this code into a file named endless_pagination.js. We'll start by writing the functions that will do the heavy lifting. First we'll need a function that renders the JSON response into HTML.

Download endlesspagination/endless_pagination.js

```
function loadData(data) {
  $.tmpl("<div class='product'> \
      <a href='/products/${id}'>${name}</a> \
      <br /> \
      <span class='description'>${description}</span> \
    </div>", data).appendTo('#content');
}
```

As we loop through each product, our template will create a where the content is the name of the product as a link. Then the new items are appended to the end of the product list so they appear on the page.

Next, since we're going to request the next page when we reach the end of the current page, we're going to need a determine what exactly the next page is. We can do this by storing the current page as a global variable. Then when we're ready, we can build the URL for the next page.

Download endlesspagination/endless_pagination.js

```
var currentPage = 0;
function nextPageWithJSON() {
  currentPage += 1;
  var newURL = 'http://localhost:8080/products.json?page=' + currentPage;

  var splitHref = document.URL.split('?');
  var parameters = splitHref[1];
  if (parameters) {
    parameters = parameters.replace(/[?&]page=[^&]*/, '');
    newURL += '&' + parameters;
  }
  return newURL;
}
```

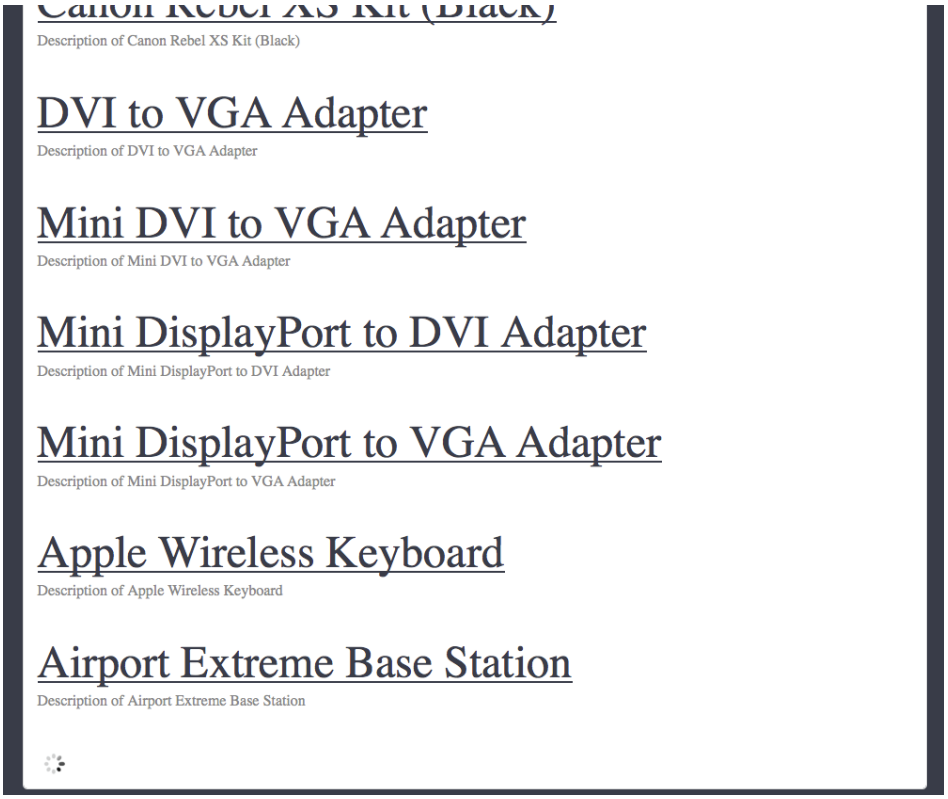


Figure 12—Reaching the Bottom of the Page

The `nextPageWithJSON()` function increments the `currentPage` variable and appends it to the current URL as a `page=` parameter. We also want to remember any other parameters that were in the current URL. At the same time, we want to make sure that the old `page` parameter, if it exists, gets overridden. This way we'll get the desired response from the server.

Now that we have functions in place to show new content and determine what the URL is for the next page, let's add the function that actually requests that content from our server. At it's core this function is just an AJAX call to the server. However we do need to implement a rudimentary way to prevent extra, unwanted calls to the server. We'll add a global variable called loadingPage that we initialize to 0. We'll increment it before we make the AJAX call, and set it back when we're done. This creates something called a *mutex*, or a locking mechanism. Without this lock in place we could poten-

tially make dozens of calls to the server for the next page, which the server would obligingly deliver, even if it's not really what we want.

Download endlesspagination/endless_pagination.js

```
var loadingPage = 0;
function getNextPage() {
  if (loadingPage != 0) return;

  loadingPage++;
  $.getJSON(nextPageWithJSON(), {}, updateContent).
    complete(function() { loadingPage-- });
}

function updateContent(response) {
  loadData(response);
}
```

After the AJAX call has finished we hand off the response to the loadData() function we defined in code, on page 60. After loadData() adds the new content, we update the URL stored in the nextPage variable. This way we're all set up to make the *next* AJAX call.

With the function to request the next page in place, we need a way to determine whether or not the user is ready to load that page. Normally this is where the user would just click the "Next Page" link but instead we want a function that returns true when the bottom of the browser's screen is within a given distance from the bottom of the page.

Download endlesspagination/endless_pagination.js

```
function readyForNextPage() {
  if (!$('#next_page_spinner').is(':visible')) return;

  var threshold = 200;
  var bottomPosition = $(window).scrollTop() + $(window).height();
  var distanceFromBottom = $(document).height() - bottomPosition;

  return distanceFromBottom <= threshold;
}
```

Finally we apply a scroll event handler that calls the observeScroll() function. That way every time the user scrolls through the page, we call the newly created readyForNextPage() helper function. When the helper function returns true we'll call getNextPage() to make our AJAX request.

Download endlesspagination/endless_pagination.js

```
function observeScroll(event) {
  if (readyForNextPage()) getNextPage();
}
```

```
$(document).scroll(observeScroll)
```

We've taken care of the endless part but in reality there *will* be an actual end to our content. After the user has seen the last product, we want to hide the spinner since seeing it will only confuse them and make them think that either their internet connection has slowed or that our site is broken. To remove the spinner we add a final check to hide it when the server has returned an empty list.

Download endlesspagination/endless_pagination.js

```
function loadData(data) {
  $.tmpl("<div class='product'> \
      <a href='/products/${id}'>${name}</a> \
      <br /> \
      <span class='description'>${description}</span> \
    </div>", data).appendTo('#content');

  if (data.length == 0) $('#next_page_spinner').hide();
}
```

Further Exploration

This technique is excellent for displaying long lists of information and is a behavior users are going to come to expect. Since we've separated our functionality into separate functions, it will be easy to adapt this solution to other scenarios. We can change the code to load the content earlier or later by changing the threshold variable or to render an HTML or XML response instead of one from JSON by modifying the loadData() functin. And best of all we can rest easy knowing that our site will still be accessible even if jQuery somehow goes missing, which we can test by disabling JavaScript.

Also See

- Recipe 12, *State-Aware AJAX*, on page 64
- Recipe 10, *Building HTML With jQuery Templates*, on page 52

Recipe 12

State-Aware AJAX

Problem

One of the things that makes the Internet great is that we can easily share links with each other. If you're browsing the web and you see something informative, entertaining or funny, you can copy the URL and send it off to your friends and family. But with the advent of AJAX enabled sites, this was no longer the case by default; clicking an AJAX link no longer guaranteed that the browser's URL would be updated. Not only does this prevent the sharing of links, but it breaks the back button. These types of sites don't act like Good Net Citizens™ because once your session is over there's no way to immediately pick up where you last left off.

Unfortunately, the endless pagination we wrote in Recipe 11, *Displaying Information with Endless Pagination*, on page 58 isn't being a very Good Net Citizen. As we scroll through the pagination and request new pages via AJAX, the browser's URL never changes. Yet, we're in a different state and the site is displaying different information than when the page was loaded. For example, if we liked a product on page 5 and sent the link in an email to a friend saying "Check out this great price!" they wouldn't necessarily know what we were talking about since they wouldn't see the same list as us. To keep our web karma in alignment and prevent user frustration, the right thing to do is to make this list page state-aware.

That's not all. When a user clicks the browser's Back button on an all-AJAX site, they often end up at whatever page led them to our site instead of where they expected to go. Then they get frustrated, click the Forward button, and will have completely lost their place. Thankfully, we have a great solution to these interface problems.

Ingredients

- jQuery

Solution

We're going to go back and finish implementing Recipe 11, *Displaying Information with Endless Pagination*, on page 58. While the old method works, we can't easily share links with anyone. When we change the page that we're looking at, we also want a way to change the browser's URL. The HTML5 specification introduced a JavaScript function called `pushState()` which, in most browsers, lets us alter the URL without leaving the page. This is great news for us web developers! We can make an entire AJAX web application that never goes through the traditional request/reload lifecycle. At the same time we get the advantages that come with that workflow. This means there's no need to reload resources like the extraneous header and footer HTML or repeated requests for images, stylesheets or JavaScript files every time we move to the next screen. And users can quickly share the current URL with others or refresh the page and retain their spot in their workflow. Best of all, the back button can work as expected too.

Using the pushState Function

The details for `pushState()` are still being ironed out. Most old browser versions don't support `pushState()`, but there are fallback solutions that use the hash portion of the URL. The solution works, but it is ugly. It's not only the issue of having pretty looking URLs. The Internet has a very good long-term memory. It was not only built to send links of funny, talking kittens to your grandmother, but also to find pages you linked to years ago that may have moved to a different server (assuming the original content creators were also Good Net Citizens and set up the old URL to return the appropriate 301 HTTP status code). If we use the URL hash as a stopgap for important information, we could be stuck supporting those deprecated links until the end of time[12]. Since URL hashes are never sent to the server, our application would have to continue redirecting traffic after `pushState()` becomes standard.

With that said, let's see what it takes to make our endless products page state-aware.

Parameters to Track

Because we don't know which page a user will load on the first request, we will keep track of the starting page as well as the current page. If a user went directly to page 3, we want them to be able to get back to page 3 on subsequent visits. If they start scrolling down from page 3 and load multiple pages, for instance to page 7, we want to know that too. We need a way to

12. http://danwebb.net/2011/5/28/it-is-about-the-hashbangs

keep track of the start and end pages so that a hard refresh won't require the user to scroll through the site again.

Next, we need a way to send the start and end pages from the client. The most direct way would be to set these params in the URL during a get request. When a page is first loaded, we'll set the page parameter of the URL to be the current page and assume the user wants to see only that page. If the client also passes in a start_page parameter, we'll know that the user wants to see a range of pages, from start_page through page. So following our example from above, if we were on page 7, but started browsing from page 3, our URL would look like http://localhost:8080/products?start_page=3&page=7.

This set of parameters should be enough information for us to recreate a list of products from the server and subsequently show the user the same page they saw when they first visited this URL.

Download statefulpagination/stateful_pagination.js

```
function getParameterByName(name) {
  var match = RegExp('[?&]' + name + '=([^&]*)')
    .exec(window.location.search);

  return match && decodeURIComponent(match[1].replace(/\+/g, ' '));
}

var currentPage = 0;
var startPage = 0;

$(function() {
  startPage = parseInt(getParameterByName('start_page'));
  if (isNaN(startPage)) {
    startPage = parseInt(getParameterByName('page'));
  }
  if (isNaN(startPage)) {
    startPage = 1;
  }
  currentPage = startPage - 1;

  if (getParameterByName('page')) {
    endPage = parseInt(getParameterByName('page'));
    for (i = currentPage; i < endPage; i++) {
      getNextPage(true);
    }
  }

  observeScroll();
});
```

All we're doing here is figuring out the start_page and current_page, then requesting those pages from the server. We use mostly the same function from the last chapter, getNextPage(), but it's been slightly modified to allow multiple requests at a time. Unlike when the user is scrolling and we want to prevent multiple, overlapping requests, right now it's alright since we know exactly which pages should be requested.

Just as we tracked the currentPage in code, on page 60, we want to track the startPage. We'll grab this parameter from the URL so we can make the requests for the pages that haven't been loaded yet. This number will never change, but we do want to make sure that it gets added to the URL and stays there every time a new page is requested.

Updating the Browser's URL

To update the URL, let's write a function called updateBrowserUrl() that will call pushState() and set the parameters for the start_page and page. It's important to remember that not every browser supports pushState(), so we need to check that the it's defined before we call it. For those browsers, this solution simply will not work, but that shouldn't stop us from future-proofing our site.

Download statefulpagination/stateful_pagination.js

```
function updateBrowserUrl() {
  if (window.history.pushState == undefined) return;

  var newURL = '?start_page=' + startPage + '&page=' + currentPage;
  window.history.pushState({}, '', newURL);
}
```

The pushState() function takes three parameters. The first is a state object which is generally a JSON object. This argument could potentially be a storage point for that information since we get JSON back from the server as we scroll. But, since our data is relatively lightweight and easy to get from the server, this strategy is overkill. For now, we'll pass in an empty hash. The second argument is a string that will update the title of the browser. This feature isn't widely implemented yet, and for our purposes, even if it was implemented, we don't really have a reason to update the browser's title. We pass in a filler argument again, this time an empty string.

Finally we get to the meat, or if you're vegetarian, the tofu, of the pushState() function. The third parameter is how we want the URL to change. This method is flexible and can either be an absolute path or just the parameters to be updated at the end of the URL. For security reasons we can't change the domain of the URL, but we can change everything after the top level domain with relative ease. Since we're only worried about updating the pa-

rameters of the URL, we prepend the pushState()'s third parameter with a "?". Finally, we set the start_page and page parameters, and if they already exist, pushState() is smart enough to update them for us.

Download statefulpagination/stateful_pagination.js

```
function updateContent(response) {
  loadData(response);
  updateBrowserUrl();
}
```

Lastly, we add a call to updateBrowserUrl() from the updateContent() function in order to make our endless pagination code state-aware. With this added, our users can now use the back button to leave our page and return with the forward button without losing their spot. They can also hit the refresh button with impunity and get the same results. Most importantly, our links our now sharable across the web. We've been able to make our index page a Good Net Citizen with minimal effort thanks to the hard work of modern browser developers.

Further Exploration

As we add more JavaScript and AJAX to our pages, we have to be aware of how the interfaces behave. HTML5's pushState() method and the History API give us the tools we need to provide support for the regular controls in the browser that people already know how to use. Abstraction layers like History.js[13] make it even easier by providing graceful fallbacks for old browsers that don't yet support the History API.

The approaches we discussed here are also making their way into JavaScript frameworks like Backbone.js which means even better back-button support for the most complex single-page applications.

Also See

- Recipe 10, *Building HTML With jQuery Templates*, on page 52
- Recipe 12, *State-Aware AJAX*, on page 64
- Recipe 14, *Organizing Code with Backbone.JS*, on page 79

13. http://plugins.jquery.com/plugin-tags/pushstate

Recipe 13

Snappier Cient-Side Interfaces with Knockout.js

Problem

When developing modern web applications, we often try to update only part of the interface in response to user interaction instead of refreshing the entire page. Calls to the server are often expensive, and refreshing the entire page can cause people to lose their place.

Unfortunately the JavaScript code for this can very quickly become difficult to manage. We start out watching only a couple of events, but suddenly we have several callbacks updating several regions of the page, and it becomes a maintenance nightmare.

Knockout is a simple, yet powerful, framework that lets us bind objects to our interface and can automatically update one part of the interface when another part changes, without lots of nested event handlers.

Ingredients

- Knockout.js[14]
- jQuery
- jQuery Templates[15]

Solution

Knockout.js uses *view-models* which encapsulates much of the view logic associated with interface changes. We can then bind properties of these models to elements in our interface.

We want our customers to be able to modify the quantity of items in their shopping cart and see the updated total in real-time. We can use Knockout's view models, data binding, and its support for templates to build the update screen for our shopping cart. We'll have a line for each item, a field for the customer to update the quantity, and a button to remove the item from the

14. http://knockoutjs.com
15. http://api.jquery.com/category/plugins/templates/

cart. We'll update the subtotal for each line when the quantity changes, and we'll update the grand total whenever anything on the line changes. When we're done, we'll have an interface that looks like Figure 13, *Our cart interface*, on page 71.

Knockout Basics

Knockout's "view models" are simply regular JavaScript objects with properties and methods with a few special keywords. Here's a simple Person object with methods for first name, last name, and full name.

Download knockout/binding.html

```
var Person = function(){
  this.firstname = ko.observable("John");
  this.lastname = ko.observable("Smith");
  this.fullname = ko.dependentObservable(function(){
    return(
      this.firstname() + " " + this.lastname()
    );
  }, this);
};

ko.applyBindings( new Person);
```

We use HTML5's data- attributes to bind this object's methods and logic to elements on our interface.

Download knockout/binding.html

```
<p>First name: <input type="text" data-bind="value: firstname"></p>
<p>Last name: <input type="text" data-bind="value: lastname"></p>
<p>Full name:
   <span aria-live="polite" data-bind="text: fullname"></span>
</p>
```

When we update either the first name or the last name text boxes, the full name shows up on the page. Since the update happens dynamically, this can cause troubles for blind users with screen readers. To solve that issue, we use the aria-live attribute to give the screen readers a hint that this part changes dynamically.

That's a relatively trivial example, so let's dig into Knockout a little more by building a single line of our cart, getting the total to change when we update the quantity. Then we'll refactor it so we can build the entire shopping cart. We'll start with the data model.

We'll represent the line item using a simple JavaScript object called LineItem with properties for name and price. Create a new HTML page, and include

Product	Price	Quantity	Total	
Macbook Pro 15 inch	1699	1	1699	Remove
Mini Display Port to VGA Adapter	29	1	29	Remove
Magic Trackpad	69	1	69	Remove
Apple Wireless Keyboard	69	1	69	Remove
Total			1866	

Figure 13—Our cart interface

both the jQuery library and the Knockout.JS library in the page's <head> section:

Download knockout/item.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Update Quantities</title>
    <script type="text/javascript"
      src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js">
    </script>
    <script type="text/javascript" src="knockout-1.2.0.js"></script>
  </head>
  <body>
  </body>
</html>
```

Add a new <script> block at the bottom of the page, above the closing <body> tag and add this code:

Download knockout/item.html

```
var LineItem = function(product_name, product_price){
    this.name = product_name;
    this.price = product_price;
};
```

In JavaScript, functions are object constructors, so we can use a function to mimic a class. In this case, the class' constructor accepts the name and the price when we create a new LineItem instance.

Now we need to tell Knockout that we want to use this lineItem class as our View Model so its properties are visible to our HTML markup. We do that by adding this call to our script block:

Download knockout/item.html

```
var item = new LineItem("Macbook Pro 15", 1699.00);
```

```
ko.applyBindings(item);
```

We're creating a new instance of our LineItem to Knockout's applyBindings() method, and we're setting the product name and price. We will make this more dynamic later, but for now we'll hard-code these values.

With the object in place, we can build our interface and pull data from the object. We'll use an HTML table to mark up our cart, and we'll use <thead> and <tbody> tags to give us a little more structure.

Download knockout/item.html

```
<div role="application">
  <table>
    <thead>
      <tr>
        <th>Product</th>
        <th>Price</th>
        <th>Quantity</th>
        <th>Total</th>
      </tr>
    </thead>
    <tbody>
      <tr aria-live="polite">
        <td data-bind="text: name"></td>
        <td data-bind="text: price"></td>
      </tr>
    </tbody>
  </table>
</div>
```

Since our table row updates based on user input, we use the aria-live attribute on the table row so screen readers know to watch that row for changes. We also wrap the whole cart within a with the HTML5-ARIA role of application, which tells screen readers that this is an interactive application. You can learn about these in the HTML5 specification.[16].

Pay special attention to these two lines:

Download knockout/item.html

```
<td data-bind="text: name"></td>
<td data-bind="text: price"></td>
```

Our LineItem instance is now a global, visible object on our page, and its name and price properties are visible as well. So with these two lines, we're saying that we want the "text" of this element to get its value from the property we specify.

16. http://www.w3.org/TR/html5-author/wai-aria.html

When we load the page in our browser, we see the row of our table start to take shape, and the name and price are filled in!

Let's add a text field to the table so that the user can update the quantity.

Download knockout/item.html

```
<td><input type="text" name="quantity"
      data-bind='value: quantity, valueUpdate: "keyup"'>
</td>
```

In Knockout, we reference data fields within regular HTML elements with text, but HTML form elements like <input> have value attributes. This time we bind the "value" attribute to a quantity property in our view model, which we need to define next.

The quantity property isn't just for displaying data; it's going to set data as well, and when we set data, we need events to fire. We do that by using Knockout's ko.observable() function as the value of our quantity property in our class.

Download knockout/item.html

```
this.quantity = ko.observable(1);
```

We're passing a default value to ko.observable() so the text field has a value when we bring the page up for the first time.

Now we can enter the quantity, but we need to show the row's subtotal. Let's add a table column to print out the subtotal:

Download knockout/item.html

```
<td data-bind="text: subtotal "></td>
```

Just like our name and price columns, we set the text of the table cell to the value of our view model's subtotal property.

This brings us to one of the more powerful features of Knockout.JS, the dependentObservable() method. We defined our quantity property as observable, which means that other things notice when that field changes. We declare a dependentObservable() which executes code whenever our observed field changes, and we assign that dependentObservable() to a property on our object so it can be bound to our user interface.

Download knockout/item.html

```
this.subtotal = ko.dependentObservable(function() {
  return(
    this.price * parseInt("0"+this.quantity(), 10)
  ); //<label id="code.subtotal" />
}, this);
```

But how does the dependentObservable() know what fields to watch? It actually looks at the observable properties we access in the function we define! Since we're adding the price and quantity together, Knockout tracks them both and runs this code when either one changes.

The dependentObservable() takes a second parameter that specifies the context for the properties. This is due to the way JavaScript's functions and objects work, and you can read more on this in the Knockout.JS documentation.

And that's it for a single row. When we change the quantity, our price updates in real-time. Now let's take what we learned here and turn this into a multiple-line shopping cart with line totals and a grand total.

Refactoring With Templates

Binding objects to HTML is quite handy, but it's likely that we'll have more than one item in our cart, and duplicating all that code is going to get a little tedious, not to mention more difficult since we'll have more than one LineItem object to bind. We need to rethink the interface a bit.

Instead of working with a LineItem as the view model, let's create another object that represents the shopping cart. This Cart object will hold all of the LineItem objects. Using what we know about Knockout's dependentObservables, this new Cart object can have a property that computes the total when any of the items in the cart changes.

But what about the HTML for the line item? Well, we can reduce duplication by defining a *template*, and telling Knockout to render that template once for each item in our cart. Let's get started.

First, let's define an array of items we'll use to populate the cart.

Download knockout/update_cart.html

```
var products = [
  {name: "Macbook Pro 15 inch", price: 1699.00},
  {name: "Mini Display Port to VGA Adapter", price: 29.00},
  {name: "Magic Trackpad", price: 69.00},
  {name: "Apple Wireless Keyboard", price: 69.00}
];
```

In a real-world situation, we would get this data from a web service or AJAX call, or by generating this array server-side when we serve up the this page.

Now, let's create a Cart object that holds the items. We'll define it the same way we defined our LineItem.

Download knockout/update_cart.html

```
var Cart = function(items){
```

Joe asks:
What about Knockout and Accessibility?

Interfaces that rely heavily on JavaScript often raise a red flag when it comes to accessibility, but use of JavaScript alone doesn't make a site inaccessible to the disabled.

In this recipe, we made use of the HTML5 ARIA roles and attributes to help screen readers understand the application we're developing, but accessibility is about much more than screen readers; it's about making our applications usable by the widest audience possible.

Knockout is a JavaScript solution and will only work when JavaScript is enabled or available, so you need to take that under consideration. We recommend that you build applications to work without JavaScript and then use Knockout to *enhance* your application. Our example uses jQuery Templates to render the cart initially, but if we were using a server-side framework we could render the HTML for the cart and only use jQuery templates for additional behavior like when we add elements to the cart. The accessibility of a site depends much more on the implementation than on the library or technology used.

```
  this.items = ko.observableArray();

  for(var i in items){
    var item = new LineItem(items[i].name, items[i].price);
    this.items.push(item);
  }
}
```

and we need to change our binding from using the LineItem class to the Cart class.

Download knockout/update_cart.html

```
var cartViewModel = new Cart(products);
ko.applyBindings(cartViewModel);
```

The items are stored in the cart using an observableArray() which works just like an observable() but has the properties of an array. When we created a new instance of our cart, we passed in the array of data. Our object iterates over the items of data, and creates new LineItem instances which get stored in the items array. Since this array is observable, our user interface will change whenever the array's contents changes. Of course, now that we're dealing with more than one item, we'll need to modify that user interface.

Knockout uses the jQuery.tmpl template library. That means we need to download that library[17] and link it in our <head> section.

Download knockout/update_cart.html

```
<script type="text/javascript" src="jquery.tmpl.min.js"></script>
```

Now, let's define a new <script> block to hold the template. To keep the browser from rendering it right away, we'll give it a type of text/html. We'll also give it an ID so we can uniquely identify it.

Download knockout/update_cart.html

```
<script type="text/html" id="rowTemplate">
  <tr aria-live="polite">
    <td>${ name }</td>
    <td>${ price }</td>
    <td><input type="text" name="quantity" data-bind='value: quantity'></td>
    <td><span data-bind="text: subtotal "></span></td>
  </tr>
</script>
```

Then we modify our HTML page by removing the table rows and calling the template by using a Knockout data-bind call on the <tbody> tag.

Download knockout/update_cart.html

```
<div role="application">
  <table>
    <thead>
      <tr>
        <th>Product</th>
        <th>Price</th>
        <th>Quantity</th>
        <th>Total</th>
      </tr>
    </thead>
    <tbody data-bind='template: {name: "rowTemplate", foreach: items}'>

    </tbody>
  </table>
</div>
```

We pass our collection of Items to the template, which renders our template for each item in the collection. We can use the template language's ${ } syntax to embed our values into the table cells.

At this point we have multiple lines displaying on the page, each subtotalling correctly. Now let's handle computing the grand total and removal of items.

17. There's a link to jQuery.tmpl on the Knockout JS website at http://knockoutjs.com/documentation/installation.html.

The Grand Total

We've already seen how Knockout's dependentObservable() method works when we used it to calculate the subtotal for each item. We can use the same approach to calculate the total for the entire cart by adding a dependentObservable() to the Cart itself.

Download knockout/update_cart.html

```
this.total = ko.dependentObservable(function(){
  var total = 0;
  for (item in this.items()){
    total += this.items()[item].subtotal();
  }
  return total;
}, this);
```

Any time any of the items in our array changes, this code will fire. To display the grand total on the form, we simply need to add the appropriate table row, right above the closing <tbody> tag. Since it's the total for the cart and not for a line item, it doesn't go in the template.

Download knockout/update_cart.html

```
<tr>
  <td colspan="4">Total</td>
  <td aria-live="polite" data-bind="text: total()"></td>
</tr>
```

When we refresh our page, we can change any quantity and update both the line total and the cart total simultaneously. Now, about that "remove" button...

Removing Items

To wrap this project up, we need to add a "Remove" button to the end of each row that removes the item from the row. Thanks to all the work we've done, this is a very simple task. First, we modify the template to add the "Remove" button:

Download knockout/update_cart.html

```
<td>
  <button
    data-bind="click: function() { cartViewModel.remove(this) }">Remove
  </button>
</td>
```

This time, instead of binding data to the interface, we bind an event and a function. In this case, we pass the item (this) to the remove() method on our cartViewModel instance. Since we haven't defined the remove() method yet, this button won't work. So let's fix that by adding this method to our Cart object:

Be Sure To Reconcile With The Server!

Building a shopping cart update screen entirely on the client side is becoming more popular. In some cases it's just not possible to send AJAX requests back and forth every time a user makes a change to the interface.

When you use an approach like this, you'll want to synchronize the data in the cart on the client side with data on the server. After all, you wouldn't want someone changing prices on you!

When the user checks out, submit the updated quantities to the server and recompute the totals server-side before checking out.

Download knockout/update_cart.html

```
this.remove = function(item){ this.items.remove(item); }
```

That's it! Since the items array is an observableArray, our entire interface gets updated. Even our grand total changes!

Further Exploration

Knockout is great for situations where we need to build a dynamic single-page interface, and because it's not tied to a specific web framework, we can use it anywhere.

More importantly, the view models Knockout uses are just ordinary JavaScript, which means we can use Knockout to implement many commonly-requested user interface features. For example, we could very easily implement an AJAX-based live-search, build in-place editing controls that persist the data back to the server, or even update the contents of one dropdown field based on the the selected value of another field.

Also See

- Recipe 10, *Building HTML With jQuery Templates*, on page 52
- Recipe 14, *Organizing Code with Backbone.JS*, on page 79

Recipe 14

Organizing Code with Backbone.JS

Problem

As users demand more robust and responsive client-side applications, developers respond with amazing JavaScript libraries. But as applications get more complex, the client-side code starts to look like your basic kitchen junk drawer, with libraries strewn about, all crammed together in a disorganized pile of event bindings, jQuery AJAX calls, and JSON parsing functions.

We need a way to develop our client-side applications using the same approach we've been using for years in our server-side code–a framework. With a robust JavaScript framework, we'll be able to keep things organized, reduce duplication, and standardize on something other developers understand.

Ingredients

- Backbone.js[18]
- Underscore.js[19]
- JSON2.js[20]
- jQuery Templates[21]
- jQuery
- QEDServer

Solution

There are a number of frameworks that we can use to make this work, but Backbone.js is one of the most popular due to its flexibility, robustness, and its code quality, despite being relatively new at the time of writing. We can use Backbone to do event binding and template rendering similar to what we did with Knockout in Recipe 13, *Snappier Cient-Side Interfaces with*

18. http://documentcloud.github.com/backbone
19. http://documentcloud.github.com/underscore/
20. https://github.com/douglascrockford/JSON-js
21. http://api.jquery.com/category/plugins/templates/

Knockout.js, on page 69, but with Backbone, we get models that interact with our server, and we get a request routing system that can monitor changes in the URL. Backbone also supports templating languages like jQuery Templates with little effort.

Let's use Backbone to improve the responsiveness of our online store's interface. Data from our logs and user studies shows that page refreshes are taking too long, and a lot of the stuff we're going to the server for could be done on the client. Our manager suggested that we take our product management interface and turn it into a single-page interface where we can add and delete products without page refreshes. Because Backbone is a complex library, this is a much longer recipe than the others in this book.

Before we get into building our interface, let's dig a little deeper into what Backbone is and how we can use it to solve our problem.

Backbone Basics

Backbone is a client-side implementation of the Model-View-Controller pattern, and it's heavily influenced by server-side frameworks like ASP.Net MVC and Ruby on Rails. Backbone has several components that help us keep things organized as we communicate with our server-side code.

Models represent the data and can interact with our backend via AJAX. Models are also a great place to do any business logic or data validations.

Views in Backbone are a little different from views in other frameworks. Instead of being the presentation layer, Backbone's views are more like "view controllers". We may have lots of events in a typical client-side interface, and the code these events trigger lives in these views. They can then render templates and modify our user interface.

Routers watch changes in the URL and can tie models and views together. When we want to show different "pages" or tabs on an interface, we can use Routers to handle requests and display different Views. In Backbone, they also provide support for the browser's Back button.

Finally, Backbone introduces Collections, which give us an easy way to fetch and work with multiple model instances. Figure 14, *Backbone's Components*, on page 81 shows how these components work together, and how we'll use them to build our product management interface.

By default, Backbone's models use jQuery's `ajax()` method to communicate with a RESTful server-side application using JSON. The backend needs to accept `GET`, `POST`, `PUT`, and `DELETE` requests and be able to look for JSON in

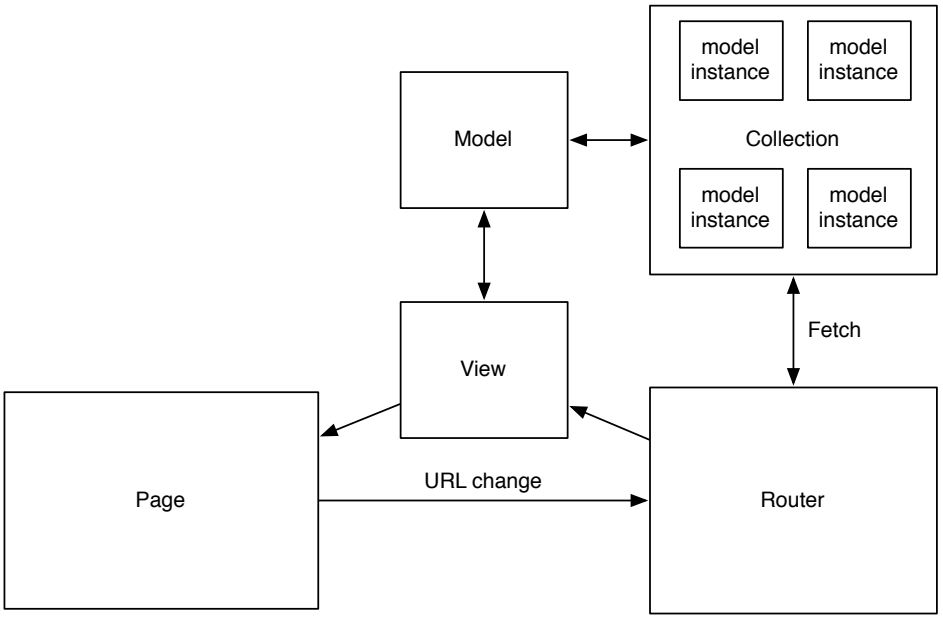


Figure 14—Backbone's Components

the body of the request. These are merely defaults though, and the Backbone documentation explains how to modify your client-side code to work with different kinds of backends.

The backend we'll be working with supports Backbone's default behavior, so we'll simply be able to call some methods on our Backbone models and Backbone will seamlessly serialize and deserialize our product information.

One last note— As we mentioned in *What about Knockout and Accessibility?*, on page 75, it's best to use frameworks like Backbone *on top of* an existing website, to provide an enhanced user experience. If your client-side code builds on a solid foundation, it's easier to provide a solution that works without JavaScript. In this recipe, we assume we're building an interface that already has a working non-JavaScript alternative.

Building Our Interface

We're going to build a simple, single page interface to manage products in our store, like the one in Figure 15, *Our Product interface*, on page 83. We'll have a form at the top of the page for adding products, and below that, we'll display a list of the products. We'll use Backbone to talk to our backend to retrieve or modify our product inventory, using it's REST-like interface:

- A GET() request to http://example.com/products.json retrieves the list of products
- A GET request to /products/1.json retrieves a JSON representation of the product with the ID of "1".
- A POST request to /products.json with a JSON representation of a product in the request body creates a new product.
- A PUT request to http://example.com/products/1.json with a JSON representation of a product in the request body updates the product with the ID of "1".
- A DELETE request to /products/1.json deletes the product with the ID of "1".

Because AJAX requests have to be done against the same domain, we'll be using QEDServer for our development server and using its product management API. We'll place all of our files in the public folder that QEDServer creates in our workspace so our development server will serve them properly.

To build our interface, we'll create a model to represent our product and a collection to hold multiple Product models. We'll use a Router to handle requests for displaying the product list and showing the form to add a new product. In addition, we'll have Views for the product list and the product form.

First, let's create a lib folder to hold the Backbone library and its dependencies.

```
$ mkdir javascripts
$ mkdir javascripts/lib
```

Next, we need to get Backbone.js and its components from the Backbone.js web site[22]. In this recipe, we're using Backbone 0.5.3. Backbone requires the Underscore.js library which provides some JavaScript functions Backbone uses behind the scenes so we can write less code. We also need the JSON2 library which provides broader support for parsing JSON across browsers. And since we're already familiar with jQuery Templates, we need that library as well so we can use it for our templating language[23]. Download these files and place them in the javascripts/lib folder.

Finally, let's create a single app.js file in the javascripts folder. This file will contain all of our Backbone components and custom code. While it might

22. http://documentcloud.github.com/backbone/
23. To save time, you can find all of these files in the book's source code.

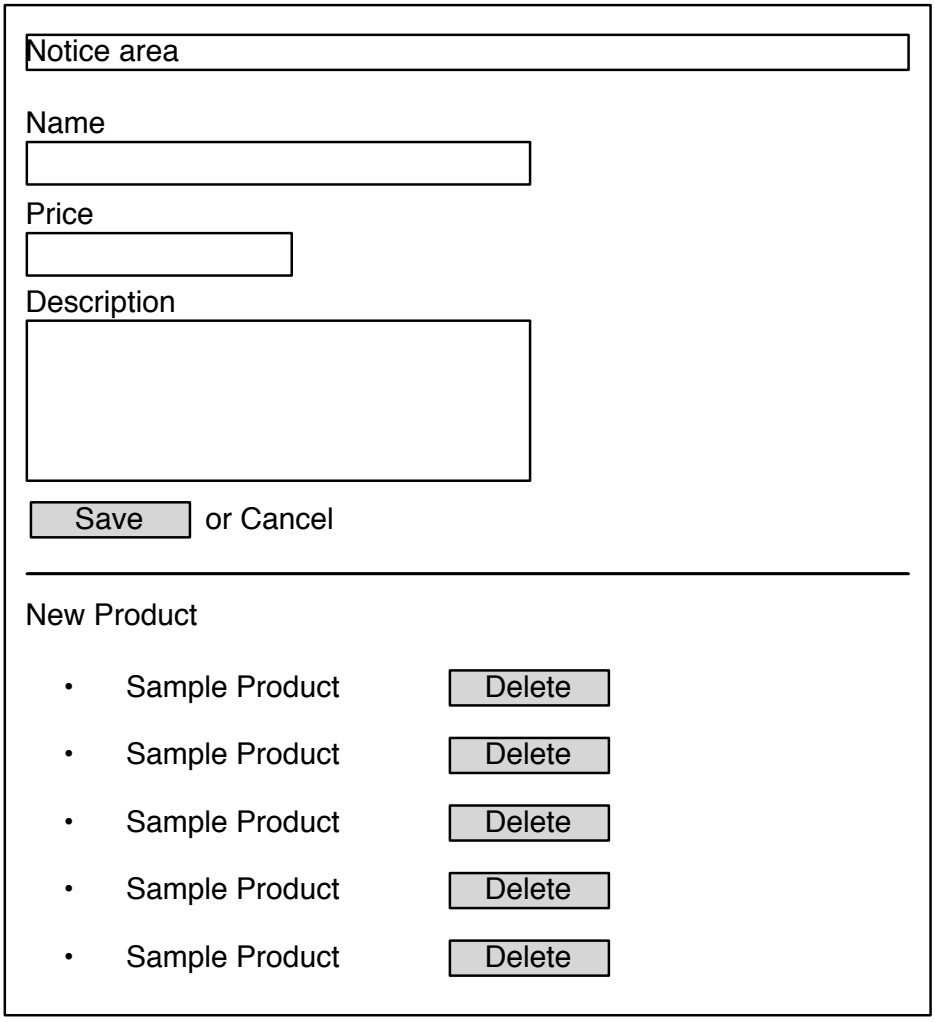


Figure 15—Our Product interface

make sense to split these into separate files, we'd end up with an additional call to the server for each file when we load the page.

Now that we have everything we need, let's create a very simple HTML skeleton in index.html to hold our user interface elements and include the rest of our files. First, we'll declare the usual boilerplate pieces and create empty s for our messages to the user, our form, and a <ul> for the product list.

Download backbone/public/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Product Management</title>
  </head>
  <body role="application">
    <h1>Products</h1>

    <div aria-live="polite" id="notice">
    </div>

    <div aria-live="polite" id="form">
    </div>

    <p><a href="#new">New Product</a></p>

    <ul aria-live="polite" id="list">
    </ul>
  </body>
</html>
```

We'll be updating these regions without refreshing the page, so we're adding in the HTML5 ARIA attributes to tell screen readers how to handle these events.[24]

Below those regions, we include jQuery, the Backbone library, and its prerequisites:

Download backbone/public/index.html

```
<script type="text/javascript"
  src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js">
</script>
<script type="text/javascript" src="javascripts/lib/json2.js"></script>
<script type="text/javascript" src="javascripts/lib/underscore-min.js"></script>
<script type="text/javascript" src="javascripts/lib/backbone-min.js"></script>
<script type="text/javascript" src="javascripts/lib/jquery.tmpl.min.js"></script>
<script type="text/javascript" src="javascripts/app.js"></script>
```

Now, let's get to work building the product list.

Listing Products

To list products, we'll fetch the products from our AJAX backend. To do this, we need a model and a collection. The model will represent a single product, and the collection will represent a group of products. When we create and delete a product, we'll be using the model directly, but when we

24. http://www.w3.org/TR/html5-author/wai-aria.html

want to grab a list of products from our server, we can use the collection to fetch records and give us a group of Backbone models we can work with.

First, let's create the model. In javascripts/app.js, we'll define our Product like this:

Download backbone/public/javascripts/app.js

```
var Product = Backbone.Model.extend({

  defaults: {
    name: "",
    description: "",
    price: ""
  },

  url : function() {
    return(this.isNew() ? "/products.json" : "/products/" + this.id + ".json");
  }
});
```

We're setting up some default values for situations where there's no data, like when we create a new instance. Next, we're telling the model where it should get its data. Backbone uses a model's url() method to figure this out, and we have to fill it in.

With a model defined, we can now create a collection, which we'll use to grab all of the products for our list page.

Download backbone/public/javascripts/app.js

```
var ProductsCollection = Backbone.Collection.extend({
  model: Product,
  url: '/products.json'
});
```

Like a model, a collection also has a url() method we have to implement, but since we're only interested in fetching all of the products, we can just hard-code the URL to /products.json.

We'll access this collection in several places in our application, so let's create an instance of our Products collection. At the very top of javascripts/app.js, we'll create the object:

Download backbone/public/javascripts/app.js

```
$(function(){
  window.products = new ProductsCollection();
```

We attach this product collection object to the window object. This will let us easily access the collection of products from multiple views later.

With our model and collection defined, we can turn our attention to the view.

The List Template and View

Backbone views encapsulate all of the logic associated with changing the interface in response to events. We'll use two views to render our list of products. We'll create one view to represent a single product, which will render a jQuery template and handle any events related to that product. We'll then use a second view that will iterate over our collection of products and then render the first view for each object, placing the results onto our page. This way, we'll have more fine-grained control over each component.

First, we create a simple jQuery template that iterates over a collection of products. We need to add this template to our index.html page, *above* the <scrip> tags that include our libraries:

Download backbone/public/index.html

```
<script type="text/html" id="product_template">
  <li>
    <h3>
      ${product.get("name")} - ${product.get("price")}
      <button class="delete">Delete</button>
    </h3>
    <p>${product.get("description")}</p>
  </li>
</script>
```

We display the product name, price, and description,as well as a button to delete the product.

Next, we create a new view called ProductView by extending Backbone's View class and defining a few key pieces:

Download backbone/public/javascripts/app.js

```
ProductView = Backbone.View.extend({
  template: $("#product_template"),
   initialize: function(){
     this.render();
   },
  render: function(){
  }
});
```

We define an initialize() function which will fire when we create a new instance of our ListView and we'll tell it to fire the view's render() function.

Every view has a default render() function, but we need to override it so it actually does something. We'll have it render our jQuery template, which

we pull off of the index page by its ID and store in a variable called template. This way we're not continuously pulling the jQuery template off of the page every time we want to render a product.

Download backbone/public/javascripts/app.js

```
render: function(){
  var html = this.template.tmpl( {product: this.model } );
  $(this.el).html(html);
  return this;
}
```

We're referencing this.model in this method, which will contain the products we want to list. When we create a new instance of our view, we can assign a model or a collection to the view so we can easily reference the model or collection in the view's methods without having to pass it around, just like we're doing with the jQuery template.

The render() method places the rendered HTML from the jQuery template into a property on the view called el and then returns this instance of the ProductView. When we call this method, we'll take the results out of that property and append it to the page.

To do that, we'll create a view called ListView which has nearly the same structure as our ProductView view, but instead of rendering a jQuery template, it's going to iterate over our products collection and render our Product view for each one.

Download backbone/public/javascripts/app.js

```
ListView = Backbone.View.extend({
  el: $("#list"),

  initialize: function() {
    this.render();
  },

  renderProduct: function(product){
    var productView = new ProductView({model: product});
    this.el.append(productView.render().el);
  },

  render: function() {
    if(this.collection.length > 0) {
       this.collection.each(this.renderProduct, this);
    } else {
      $("#notice").html("There are no products to display.");
    }
  }
});
```

We need to update the contents of the list region on our page with the list of products. We're storing a reference to this region in a property called el. This gives us convenient access to it from our render() method, similar to how we referenced our jQuery template in ProductView.

Backbone uses Underscore.js to give us some helpful functions that make working with collections very easy. In our render() method, we're iterating over the collection using the each() method and calling our renderProduct method. The each() method automatically passes the product. We pass this as a second parameter to specify that we want the view to be the scope for the renderProduct(). Without that, the each() method would look for our renderProduct() method on the collection, and it wouldn't work.

So far, we've declared a model, a collection, a couple of views, and we've added a template, but we still don't have anything to show for it. We need to tie this all together when we load our page in our browser. We'll do that with a Router.

Handling URL Changes with Routers

When we bring up our page, we'll want to fire some code to fetch our collection of products from the AJAX API. Then we'll need to pass the collection of products to a new instance of our ListView so we can display the products. Backbone's Routers let us respond to changes we make in the URL and respond by executing functions.

Let's create a new Router called ProductsRouter. Inside this file, we'll extend Backbone's Router and then define a *route* that maps the part of the URL that appears after the hash mark to a function in our Router. To handle the default case where there is no hash in the URL, we define a route that's empty, and map it to a function called index(). When we load the page index.html, this default route will fire.

Download backbone/public/javascripts/app.js

```
ProductsRouter = Backbone.Router.extend({
    routes: {
      "": "index"
    },
    index: function() {
    }
});
```

Inside of the index() action, we call the fetch() method of our Products collection to retrieve the data from our server.

Download backbone/public/javascripts/app.js

```
index: function() {
```

```
  window.products.fetch({
    success: function(){
      new ListView({ collection: window.products });
    },
    error: function(){
      $("notice") = "Could not load the products.";
    }
  });
}
```

The fetch() method takes in success and error callbacks. When we get an error from our backend, we display a notice to the users in the notice region of the page. When the backend returns data to the collection, the success() callback fires, and we create a new instance of our view. Since our list view automatically renders thanks to the code we placed in the initialize() method of the view, we don't have anything else to do except create a new instance of the Router to kick everything off.

In javascripts/app.js, we create the Router instance right below our definition for window.productCollection. Then we have to tell Backbone to start tracking URL changes.

Download backbone/public/javascripts/app.js

```
window.products = new ProductsCollection();
// START_HIGHLIGHTING
$.ajaxSetup({ cache: false });
window.router = new ProductsRouter();
Backbone.history.start();
// END_HIGHLIGHTING
```

The Backbone.history.start(); line makes Backbone start watching changes in the URL. If we forget this, the Router won't work and we won't see anything happen.

This line:

```
$.ajaxSetup({ cache: false });
```

prevents some browsers from caching AJAX responses from our server.

When we visit http://localhost:8080/index.html, we finally see a list of our products.

To review our progress so far, we've got a Router that looks at the URL and fires a method which uses our Collection to fetch some Models from our web service. This Collection is then passed to a View which renders a Template and outputs the template onto our user interface. You can see a diagram of these interactions in Figure 16, *Listing Products with Backbone*, on

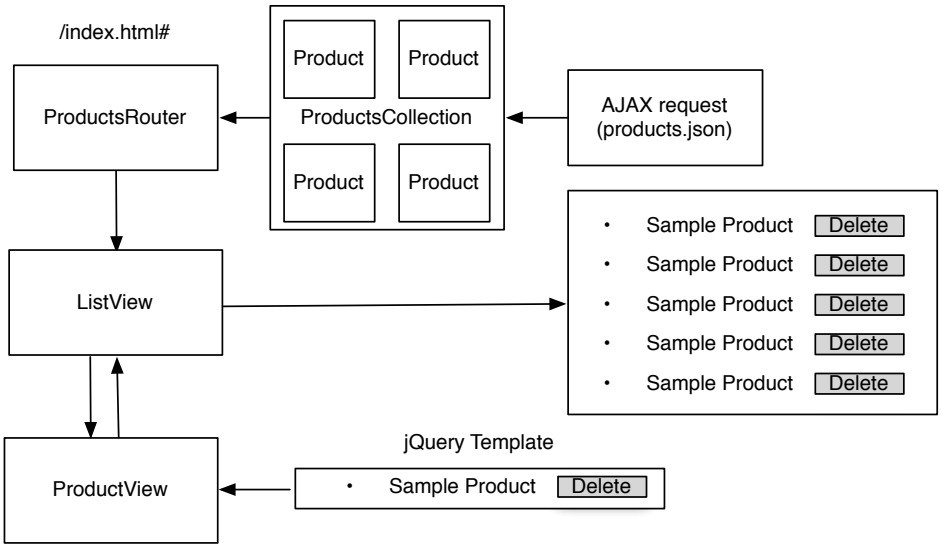


Figure 16—Listing Products with Backbone

page 90. That may seem like lot of steps and a lot of code for something that's fairly trivial, but it will add up to huge time savings as we evolve our code. We've laid the groundwork for adding, updating, and deleting products, and we won't have to struggle with where these pieces go. Let's go a little further by adding the ability to add products.

Creating a New Product

To create a product, we'll need to add a form to the page when the user clicks the "New Product" link. When the user fills out the form, we'll take the form data, submit it to our backend, and then redisplay the list.

First, let's add a jQuery template for the form to our index.html page, *right below* the product template, but still *above* the <scrip> tags that include our libraries:

Download backbone/public/index.html

```
<script type="text/html" id="product_form_template">
  <form>
    <div class="row">
      <label>Name<br>
        <input id="product_name" type="text" name="name"
               value="${product.get('name')}">
      </label>
    </div>
    <div class="row">
```

```
      <label>Description<br>
        <textarea id="product_description"
                  name="description">${product.get('description')}</textarea>
      </label>
    </div>
    <div class="row">
      <label>Price<br>
        <input id="product_price" type="text" name="price"
               value="${product.get('price')}">
      </label>
    </div>
    <button>Save</button>
  </form>
  <p><a id="cancel" href="#">Cancel</a></p>
</script>
```

The jQuery template tags will pull values out of the model into the form fields. This is why we set default values in our Backbone model.

Now we need a view to render this template from a model. Let's create a new view called FormView in javascripts/app.js similar to the one we created for our list. This time, we'll set the el variable to the form region of our page, and we have the render() function grab our form template, rendering the result into that region.

Download backbone/public/javascripts/app.js

```
FormView = Backbone.View.extend({
  el: $("#form"),
  template: $("#product_form_template"),
  initialize: function(){
    this.render();
  },
  render: function(){
    var html = this.template.tmpl( {product: this.model });
    this.el.html(html);
  }
});
```

When our user clicks on the "New Product" link, we want this view to render the form onto the page. Since the link changes the URL by adding "#new", we can use the Router to respond to that change. First, we need to modify the routes section so we have a route for #new which is where our "New Products" link points.

Download backbone/public/javascripts/app.js

```
routes: {
  "new": "newProduct",
  "": "index"
},
```

And then we need to define the function that grabs a new model and passes it to a new instance of a Form view, so that view can render on the page. We'll place this method above our index() method, and since these method declarations are actually defined using a JavaScript hash, we need to ensure we place a comma between each of these declarations.

Download backbone/public/javascripts/app.js

```
newProduct: function() {
  new FormView( {model: new Product()});
},
```

When we reload our page and click the "New Product" link, our form displays. With Backbone's History tracking, we can press the "Back" button in our browser and the URL will change. But we can't save new records yet. We need to add the logic for that next.

Responding To Events In The View

We've used our Router to display the form, but Routers can only respond to changes in the URL. We need to respond to click events on the "Save" button and the "Cancel" link. We'll do that in the Form view we created.

First, let's define events for the view to watch. We add this to our view, above the initialize() function:

Download backbone/public/javascripts/app.js

```
events: {
   "click .delete": "destroy"
 },
events: {
    "click #cancel": "close",
    "submit form": "save",
},
```

The syntax here is a little different than typical JavaScript event monitoring; The key of the hash defines the event we're watching followed by the CSS selector for the element we want to watch. The value specifies the function on the view we want to invoke. In our case, we're watching the click event on our cancel button and the submit event on our form.

The code for the "close" link is easy—we simply remove the contents of the HTML element that contains this view:

Download backbone/public/javascripts/app.js

```
close: function(){
  this.el.unbind();
  this.el.empty();
},
```

The save() method is a little more complex. We first prevent the form from submitting, and then we grab the values from each form field, placing those values into a new array. Then we set the model's attributes and call the model's save() method.

Download backbone/public/javascripts/app.js

```
save: function(e){
  e.preventDefault();

  data = {
    name: $("#product_name").val(),
    description: $("#product_description").val(),
    price: $("#product_price").val()
  };

  var self = this;

  this.model.save(data, {
    success: function(model, resp) {
      $("#notice").html("Product saved.");
      window.products.add(self.model);
      window.router.navigate("#");
      self.close();
    },
    error: function(model, resp){
      $("#notice").html("Errors prevented the product from being created.");
    }
  });

},
```

The save() method expects us to use the same approach we used with the fetch() method on collections, in which we define the behavior for success and for errors. Since those callbacks have a different scope, we create a temporary variable called self that we assign the current scope to so we can reference that scope in the success callback. Unlike the each() method we used when we rendered the list of products, Backbone doesn't support passing the scope to the callbacks.[25]

When the save is successful, we add this new model to our collection and we use the Router to alter the URL. This doesn't actually fire the associated function in the Router though, which means we won't see our new product in the list. But that's an easy fix thanks to Backbone's event binding.

When we add a model to a collection, the collection fires off an add event which we can watch. Remember the renderProduct() method in our List view? We can have our List view execute that method any time we add a model to

25. At least, not at the time this was written.

our collection. All we have to do is add this line to the initialize() method of our ListView:

Download backbone/public/javascripts/app.js

```
this.collection.bind("add", this.renderProduct, this);
```

The bind() method lets us bind to specific events, specifying the event, the function, and the scope. We pass this as the third parameter to specify that we want the *view* to be the scope, and not the collection, just like we did in the List view's render() method with collection.each.

Since we reused the renderProduct() when we added a new record, the new record is appended to the bottom of the list. To make it appear at the top of the list, we could instead make a new addProduct() which would use jQuery's prepend() method instead, but we'll leave that up to you to try.

Now that we can create products and see the list of products update all on the same page without refreshing, let's tun our attention to removing products. That's where a lot of this up-front work and code organization really pays off.

Deleting A Product

To delete a product, we use what we learned from working in the FormView and implement a destroy() function in our ProductView that fires when we press the Delete button.

First, we define the event to watch for clicks on the buttons with the class of "delete"

Download backbone/public/javascripts/app.js

```
events: {
   "click .delete": "destroy"
 },
events: {
    "click #cancel": "close",
    "submit form": "save",
},
```

Then we define the destroy() method that the event will call. Inside this method we call the destroy() method on our model that's bound to this view. This uses the same success and error callback strategy we've used previously. We use the self trick we used when creating records to get around the scope issues, just like we did when we saved records in the Form view.

Download backbone/public/javascripts/app.js

```
destroy: function(){
  var self = this;
```

```
  this.model.destroy({
  success: function(){
    self.remove();
  },
  error: function(){
    $("#notice").html("There was a problem deleting the product.");
  }
  });
},
```

When the model is successfully deleted by the server, our success callback fires, calling the remove() method of this view, making the record disappear from the screen. If something goes wrong, we display a message on the screen insteadd.

And that's it! We now have a simple, well-organized prototype that we can show off, or continue to evolve.

Further Exploration

This application is a good start, but there are a few things you can explore on your own.

First, we're using jQuery to update the notice in several places, like this:

```
$("#notice").html("Product saved.");
```

You could create a wrapper function for that to decouple it from the markup, or even use another Backbone view and jQuery template to display those messages.

When we save records, we're explicitly pulling the values off the form with jQuery selectors. You could instead place the data right into the model instance by using onchange events on the form fields.

We built support for adding and removing records in this recipe, but you could go one step further and add in support for editing records. You could use the Router to display the form, and even reuse the same form view we used for creating products.

Backbone provides a great system for working with backend data, but that's just the beginning. You're not required to use Backbone with an AJAX backend; you could just as easily use it to persist data to HTML5's client-side storage mechanism.

For more robust integration with server-side applications, Backbone supports HTML5's History pushState(), which means we can use real URLs instead of hash-based ones. We can then have graceful fallbacks that serve pages from

our server when JavaScript is disabled, but work with Backbone when javaScript is available.

With its numerous options and excellent support for AJAX backends, Backbone is an incredibly flexible framework that works well for those client-side situations where we need an organized structure.

Also See

- Recipe 10, *Building HTML With jQuery Templates*, on page 52
- Recipe 13, *Snappier Cient-Side Interfaces with Knockout.js*, on page 69

CHAPTER 3

Data Recipes

Web developers work with data in many forms. Sometimes we're pulling in a widget from another service, and other times we're taking data from our users. In these recipes, we spend some time consuming, manipulating, and presenting data.

Recipe 15

Adding an Inline Google Map

Problem

Users want simple methods to locate their destination, and they want that information quickly in an easy and accessible manner. While addresses and written directions work, the simplest way is to glance at a map, memorize the street number, and grab your keys and go. By including a map on our site, we immediately give users a sense of where things are located and how they can get there.

Ingredients

- The Google Maps API

Solution

Using the Google Maps API, we can bring the power and functionality of Google Maps into our own application. We can render maps of two types: static and interactive. The static map is an image that we can insert into our page, whereas the interactive map allows for zooming and panning. The Google Maps API supports any programming language that can make a request to Google's servers. The documentation includes a lot of JavaScript examples, which is perfect for our needs. [1]

We can use the API to accomplish any task that a user could accomplish in the full application. We can render maps of two types: static and interactive. The static map is an image that we can insert into our page, whereas the interactive map allows for zooming and panning.

Along with rendering maps, the JavaScript API lets us insert other elements on the maps. We can place markers and bind mouse events to the markers. We can also create popout dialogs that show information directly within the map. We can show street views, Geolocate the user, create routes and direc-

1. http://code.google.com/apis/maps/documentation/javascript/reference.html

tions, and draw custom models on the map. The sky is in fact the limit until Google launches their space program and has taken over NASA.[2]

We're working with a local university to develop a map for their web page for new visitors. The Admissions office wants to show these visitors where they can find places to eat as well as where to park. We'll create an interactive map that contains markers and information by using the JavaScript Google Maps API.

Let's start off by creating a basic HTML document. We will declare the <DOCTYPE> as HTML5 as a recommendation from Google; however, if you can't use <DOCTYPE html> in your own application, you are not explicitly required to do so.

Download googlemaps/map_example.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Freshman Landing Page</title>

    <style>
    </style>

    <script type="text/javascript">
    </script>
  </head>

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

Next, we'll include the Google Maps JavaScript API in our document. To make this request, we need to define whether or not our application is using a sensor to determine our user's location. Since this is not within the scope of the tutorial, we will set it to false.

Download googlemaps/map_example.html

```
<script type="text/javascript"
  src="http://maps.google.com/maps/api/js?sensor=false">
</script>
```

The API requires a to act as a container for the map, so we'll add that to our page.

Download googlemaps/map_example.html

```
<div id="map_canvas"></div>
```

2. http://www.google.com/space

The map will scale to the size of this container, so let's set dimensions on this with CSS, by adding it to a new <style> section in our page's <head> region like this:

Download googlemaps/map_example.html

```
#map_canvas {
  width: 600px;
  height: 400px;
}
```

This container is now ready to hold a map that is 600x400 pixels. Let's go fetch some data.

Loading the Map With JavaScript

At the bottom of our <head> region, let's add a <script> block to hold the code that will initialize our map. We'll create a function called loadMap() to load the map based on our latitude and longitude, and we'll make this happen when the browser window loads. If you're using a framework such as jQuery in your project, you can do the loading of the map inside of your DOM-ready call, but we'll do this with vanilla JavaScript for our example.

Download googlemaps/map_example.html

```
window.onload = loadMap;
```

Next, we'll create the loadMap() function. Since we're not using a sensor, we'll hard-code our latitude and longitude. These coordinates define the center point of the map. To find these values, we have a few options. We could navigate to Google Maps, find what we want to center our map on, right click on a pin, and select "What's here?", the values for latitude and longitude appear in the search box. Alternatively, we can use Google Maps Lat/Long Popup. [3] This application allows us to click on a location to find our values.

Download googlemaps/map_example.html

```
function loadMap() {
  var latLong = new google.maps.LatLng(44.798609, -91.504912);

  var mapOptions = {
    zoom: 15,
    mapTypeId: google.maps.MapTypeId.ROADMAP,
    center: latLong
  };

  var map = new google.maps.Map(document.getElementById("map_canvas"),
        mapOptions);
}
```

3. http://www.gorissen.info/Pierre/maps/googleMapLocationv3.php

Within this function, we create an object to hold some options for our map. We can define the type of map we want, a zoom value, and more. The zoom requires some experimentation; the higher the number, the further in it zooms. A value of 15 works well for street level maps.

We can change how the map appears by setting a different `mapTypeId`. Note that zoom values along with maximum ranges for zoom change when changing map type. You can find a reference for map types in the Google Maps API documentation. [4]

Finally, we create the map. The Map constructor requires that we pass the DOM element that will hold the map along with our object containing the options. When we load this page in our browser, as shown in Figure 17, *The initial map*, on page 102, we see a map centered on our desired location.

Creating Marker Points

To show incoming freshman where they can go to get a bite to eat or otherwise be social, we will create markers on the map. A marker in Google Maps is one of many overlays that that we can add. Overlays respond to a click event, and we will use this to show an info window when the marker is clicked.

Since we already have our map, creating the marker is as simple as invoking the constructor and passing some options.

Download googlemaps/map_example.html

```
mogiesLatLong = new google.maps.LatLng(44.802293, -91.509376);
var marker = new google.maps.Marker({
  position: mogiesLatLong,
  map: map,
  title: "Mogies Pub"
});
```

To define a marker, we pass the latitude and logitude coordinates, the map that will hold the marker, and a title that appears when we hover over the marker.

Next, let's create the info window that appears when this marker is clicked. To create an info window, invoke the constructor.

Download googlemaps/map_example.html

```
var mogiesDescription = "<h4>Mogies Pub</h4>" +
  "<p>Excellent local restaurant with top " +
  "of the line burgers and sandwiches.</p>";
```

4. http://code.google.com/apis/maps/documentation/javascript/reference.html#MapTypeId

Figure 17—The initial map

```
var infoPopup = new google.maps.InfoWindow({
  content: mogiesDescription
});
```

Finally, we need to add an event handler to the marker. Using the Google Maps event object, we add a listener to open the info window.

Download googlemaps/map_example.html

```
google.maps.event.addListener(marker, "click", function() {
  infoPopup.open(map,marker);
});
```

When we click the marker, a new window shows up that gives us information about the location. as you can see in Figure 18, *A clicked marker*, on page 103,.

We can add any amount of HTML that we wish to the window. This gives us the freedom to show large amounts of information. From here, we can gather the coordinates of other points of interest and build the rest of the map.

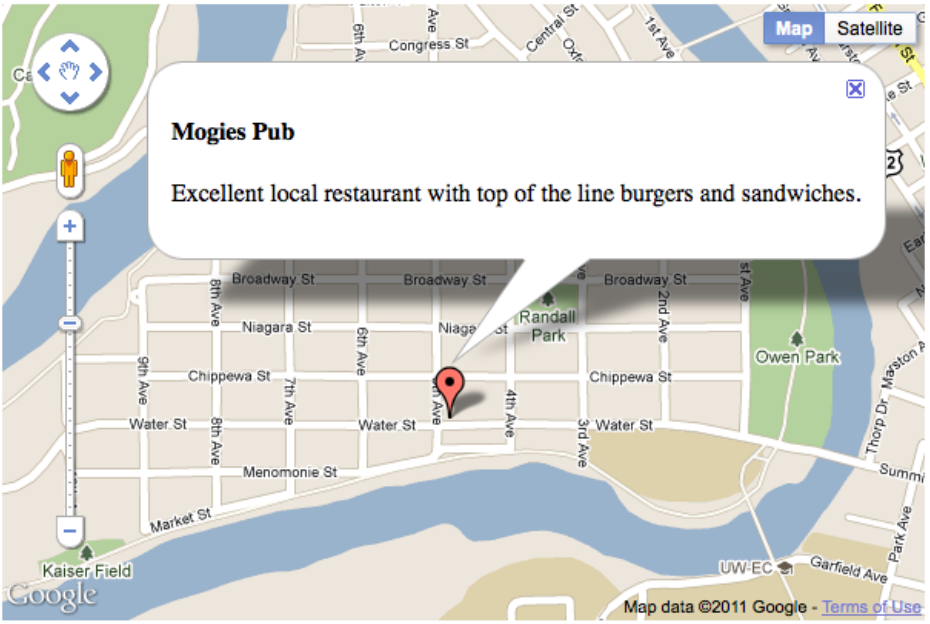


Figure 18—A clicked marker

Further Exploration

We have only scratched the surface of what can be accomplished with the Google Maps API. Along with markers, there are other layers of interaction that make the map more usable for your customers. You can create directions, map routes, use Geolocation, and even add street views. Each of these features is well explained in the GoogleMaps API documentation[5], and there are a number of working examples to follow along with.

Google Maps is just one component of the Google APIs. To see a full list of Google APIs, have a look at the Google APIs and Developer Products Page.[6]

Also See

- Recipe 17, *Building a Simple Contact Form*, on page 105
- Recipe 18, *Accessing Cross-site Data with JSONP*, on page 106
- Recipe 19, *Creating a Widget to Embed on Other Sites*, on page 110

5. http://code.google.com/apis/maps/documentation/javascript/reference.html
6. http://code.google.com/more/table

Recipe 16

Charts and Graphs with Highcharts

Coming soon...

Recipe 17

Building a Simple Contact Form

Coming soon...

Recipe 18

Accessing Cross-site Data with JSONP

Problem

We need to access data from a site on another domain, but are unable to do it using a server-side language, either due to restrictions on our web server, or because we want to push the load on to the user's browser. Regular API calls to external sites are not an option because of the same origin policy[7] which prevents client-side programming languages like JavaScript from accessing pages on different domains.

Ingredients

- jQuery
- A remote server returning JSONP
- Flickr API Key [8]

Solution

We can use JSONP to load remote data from a server at another domain. JSONP, or JSON with Padding, returns data in the JSON format, but wraps it in a call to a function. When the browser loads the script from the remote server it tries to run the function if it exists on the page, with the JSON data passed in as a variable. All we have to do is write the function that will be called, tell it how to process the JSON, and we will be able to work with data from a remote site.

We'll use the Flickr API to load the 12 most interesting photos of the moment. Some APIs let you set the function name that wraps the content when you load the page on their server, but the Flickr API always returns data wrapped in a call to `jsonFlickrApi()`. This is the function we'll need to write on our page once we have the data loaded from Flickr.

7. https://developer.mozilla.org/en/Same_origin_policy_for_JavaScript
8. http://www.flickr.com/services/api/keys/

We'll start with a blank page with no content in the <body> - everything that ends up being displayed on the page will be loaded dynamically.

First we'll create a function to load photos from Flickr. In loadPhotos() we set our API key, the method from Flickr we want to use, and the number of photos we want Flickr to return to us. Other methods available from Flickr are in the API documentation.[9]

Download jsonp/index.html

```
function loadPhotos(){
  var apiKey = '98956b44cd9ee04132c7f3595b2fa59e';
  var flickrMethod = 'flickr.interestingness.getList';
  var photoCount = '12';
  var extras = 'url_s';
  $.ajax({
    url:'http://www.flickr.com/services/rest/?method='+flickrMethod+
     '&format=json&api_key='+apiKey+
     '&extras='+extras+
     '&per_page='+photoCount,
    dataType: "jsonp"
  });
}
```

We define a few variables to make it easier to change what we're requesting from Flickr without having to dig through the URL. We also add an extra attribute, url_s, to the request so that the data we get back contains the URL of a small version of the photos. Next, jQuery's ajax() function makes a call to Flickr. We set the dataType to "jsonp" so that jQuery knows this request will be across domains.

Now we'll create the function that loads the data returned to us from Flickr's API. The data returned from Flickr contains several things, including how many other pages are available if we want to get more photos, but this time we'll just use the 12 photos we requested.

Download jsonp/flickr_response.html

```
jsonFlickrApi({
  "photos": {
    "page": 1,
    "pages": 250,
    "perpage": 2,
    "total": 500,
    "photo": [
      {
        "id": "5889925003",
        "owner": "12386438@N04",
```

9. http://www.flickr.com/services/api/

```
        "secret": "51c74e7c3e",
        "server": "6034",
        "farm": 7,
        "title": "",
        "ispublic": 1,
        "isfriend": 0,
        "isfamily": 0,
        "url_s": "http:\/\/farm7.static.flickr.com\/1\/image_m.jpg",
        "height_s": "160",
        "width_s": "240"
      },
      ...
    ]
  },
  "stat": "ok"
})
```

The data we get from Flickr includes the photos in an array appropriately called 'photos', so we'll want to loop over each of those and build out the image tags to add to the page.

Download jsonp/index.html

```
function jsonFlickrApi(data){
  $.each(data.photos.photo, function(i,photo){
    var imageTag = $('<img>');
    imageTag.attr('src', photo.url_s);
    $('body').append(imageTag);
  });
}
```

We call `$.each(data.photos.photo, function(i,photo){...}` to go over the array of photos. Inside our loop we'll work with each photo to build an `<img>` tag and set its `src` attribute to the URL of the small photo that we requested with `url_s` in the `extras` param of the querystring. Now that the `<img>` is built, append it to the body of the page, and we have our own gallery of the 12 most interesting pictures on Flickr at this moment.

With all the pieces in place we just need to call `loadPhotos()` when the DOM is ready and then we'll have a page full of photos.

Download jsonp/index.html

```
$(function(){
  loadPhotos();
});
```

JSONP gives us a way to load dynamic content from external sites without needing to resort to server-side languages. It's a pretty easy way to pull content in to our pages,

Further Exploration

What if we were relying on an external site to provide functionality for our site and they made the current status of their system available via JSONP? We could refresh the current status at a regular interval, like every 60 seconds, and update the page when there is an update.

Since this all happens client-side we don't have to worry about any additional load on our server, but it could be something that our user doesn't want to happen. To avoid making unwanted requests we could add a checkbox to the page that, when checked, activates the timer and the updater.

See Also

- Recipe 19, *Creating a Widget to Embed on Other Sites*, on page 110
- Recipe 14, *Organizing Code with Backbone.JS*, on page 79

Recipe 19

Creating a Widget to Embed on Other Sites

Problem

Widgets are a combination of HTML, Javascript, and CSS that allow owners of other sites to embed code in their site that will display content from another site. From general information about our site to tailored content around a user's activities, widgets let us expand the reach of our site and allow users to share that they use our site. It's a simple concept, but developing a widget requires us to do a few things that may be unfamiliar, like ensuring our JavaScript doesn't conflict with existing JavaScript on our user's site, and loading data from a remote site. Properly encapsulating our code will ensure that the functions we introduce don't inadvertently overwrite existing code or other widgets, which could break a page that was working fine before our widget was added to the page.

Ingredients

- jQuery
- JSONP

Solution

Widgets are small chunks of code that users can add to their own web pages that will load content from another site. Using JavaScript and CSS we can load content from our server and insert it in to the page, and all the user has to do is load a JavaScript file from our server. Additionally, because none of the actual code is on the user's server, we can make adjustments and add new features, and the end users will see those changes as we make them available.

We'll create a widget that lets users include the commit logs from the official Ruby on Rails repository[10] on their website. We'll use JavaScript to create an anonymous function to avoid conflicting with any JavaScript that is already on the page. Next we'll check to see if jQuery is already loaded so that

10. https://github.com/rails/rails

we have access to its shortcuts and helper methods. If it's not, or if it's not the right version, we'll load our own copy. Then we'll execute and create our actual widget by loading data remotely with JSONP, which lets us access information from a remote server via Javascript, without issues due to getting that data from a different domain. After loading the that content with JavaScript, we will generate HTML and insert it on the page as shown in Figure 19, *Our widget on a simple page*, on page 112.

A widget should be simple to add, so we'll design our widget so our user only has to add two lines of code on their site - a link to the JavaScript and a where the content will be inserted once it has loaded.

Download widget/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Swapping Examples</title>
  </head>
  <body>
    <div style="width:350px; float:left;">
      <h2>AwesomeCo</h2>
      <p>
        Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam
        nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat
        volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation
        ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.
        Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse
        molestie consequat, vel illum dolore eu feugiat nulla facilisis at
        vero eros et accumsan et iusto odio dignissim qui blandit praesent
        luptatum zzril delenit augue duis dolore te feugait nulla facilisi.
      </p>
    </div>
    <script src="widget.js"></script>
    <div id="widget"></div>
  </body>
</html>
```

First, we create an anonymous function that keeps our code from affecting the user's existing code. This is a common and critical practice that isolates our code from other JavaScript code. When we give our users some code to place on their site, we want to make sure we don't cause their existing code to stop working, and we need to ensure their code doesn't break our widget. This function will automatically run once the script has been loaded on the page, which will then populate our widget.

```
(function() {...})();
```

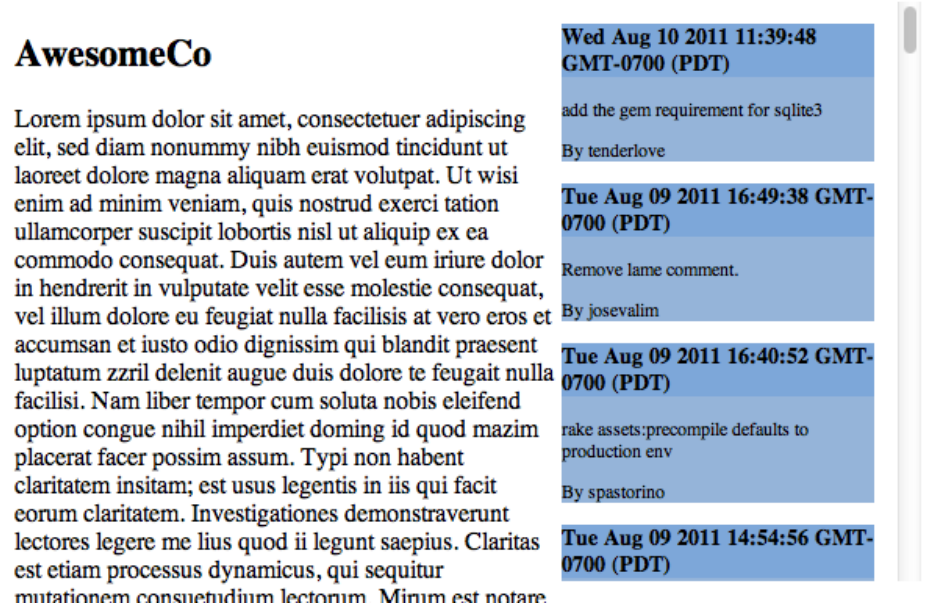


Figure 19—Our widget on a simple page

Since our widget is going use jQuery we want to make sure that it is scoped to only run within the widget, again ensuring that our widget stays completely isolated from any other client-side code.

Download widget/widget.js

```
var jQuery;

if (window.jQuery === undefined || window.jQuery.fn.jquery !== '1.6.2') {
    var jquery_script = document.createElement('script');
    jquery_script.setAttribute("src",
        "https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js");
    jquery_script.setAttribute("type","text/javascript");
    jquery_script.onload = loadjQuery; // All browser loading, except IE
    jquery_script.onreadystatechange = function () { // IE loading
        if (this.readyState == 'complete' || this.readyState == 'loaded') {
            loadjQuery();
        }
    };
    // Insert jQuery to the head of the page or to the documentElement
    (document.getElementsByTagName("head")[0] ||
      document.documentElement).appendChild(jquery_script);
} else {
    // The jQuery version on the window is the one we want to use
    jQuery = window.jQuery;
    widget();
```

```
}

function loadjQuery() {
    // load jQuery in noConflict mode to avoid issues with other libraries
    jQuery = window.jQuery.noConflict(true);
    widget();
}
```

When we load jQuery we assign it to a variable that is scoped to our function using var. By using var for all of our variables, we ensure that they are scoped to only our function, again ensuring that we don't affect the user's existing code. If the jQuery version we want is already loaded we'll use the existing library, otherwise we build a script tag and insert it in to the document. We also call jQuery's noConflict() method when we load our local instance of jQuery to avoid conflicts with other versions of jQuery or other libraries like Prototype that use $() as a top-level function name.

Now that we've gotten jQuery in place, we can load our widget's data using JSONP and insert it in to the page. We'll use GitHub's API to load the latest commits to Rails.

Download widget/widget.js

```
function widget() {
  jQuery(document).ready(function($) {
    // Load Data
    var account = 'rails';
    var project = 'rails';
    var branch = 'master';

    $.ajax({
      url: 'http://github.com/api/v2/json/commits/list/'+
        account+
        '/'+project+
        '/'+branch,
      dataType: "jsonp",
      success: function(data){
        $.each(data.commits, function(i,commit){
          var commit_div = document.createElement('div');
          commit_div.setAttribute("class", "commit");
          commit_div.setAttribute("id","commit_"+commit.id);
          $('#widget').append(commit_div);
          $('#commit_'+commit.id).append("<h3>"+
            new Date(commit.committed_date)+
            "</h3><p>"+commit.message+"</p>"+
            "<p>By "+commit.committer.login+"</p>");
        });
      }
    });
```

```
    var css = $("<link>", {
      rel: "stylesheet",
      type: "text/css",
      href: "widget.css"
    });
    css.appendTo('head');
  });
}
```

In widget() we first load our data using JSONP and get ready to display it. We use jQuery's ajax() function to request the data, then use the success call to create a new for each commit that contains the date of the commit, its author, and its message. As we create each div we append it to the #widget div that we had the user add to their page alongside the <script> tag.

After we've loaded the data, we can build the HTML to display the data in the widget and insert it into the widget div that we had the user add alongside the <script> tag. We also load a stylesheet and apply it to the widget.

Download widget/widget.css

```
#widget {
  width:230px;
  display:block;
  font-size: 12px;
  height: 370px;
  overflow-y: scroll;
}

.commit {
  background-color: #95B4D9;
  width:200px;
}

.commit h3 {
  display:block;
  background-color: #7DA7D9;
}
```

The stylesheet we load sets a height and width for the element and sets its overflow-y attribute to scroll. This lets us include large amounts of data without worrying about overwhelming the page that our widget is embedded on.

Now we have a simple chunk of code we can give to anyone who wants to include information from our site on their own. Whether it's information tailored to their specific account or general news about what's happening on our site, widgets make it easy to extend the reach of our content and potentially increase our users' interaction with our site.

Further Exploration

The widget we created loads content only once, when the page it is embedded on loads, but doesn't offer any information specific to one user or their account. If we wanted our widget to include information to identify a user so that the remote server could return more relevant data, how might we do that? This could be done with a variable in the URL of the <script> tag that we use to dynamically generate the JavaScript on the server. You could also use a different JavaScript file for each user's content.

Widgets can also offer much more interaction and go beyond just displaying content from JSON or XML. You could use jQuery to create a widget that users can click through multiple records, rather than having to scroll as they do in our example. You could load this data when the page loads, or make a request to the remote server every time a new record is requested. Or you could have the widget that automatically refreshes itself every 60 seconds with the latest content.

You could also create an interactive widget that requests data from visitors on our user's site and allows them to submit information to us, whether via email or submitting to our site.

There are many possibilities for widgets. Any time there is information that users want to share, or when you want to make it easy for a user to collect data for your site, giving them a widget to use is a great option.

Also See

- Recipe 18, *Accessing Cross-site Data with JSONP*, on page 106
- Recipe 29, *Cleaner JavaScript with CoffeeScript*, on page 135

Recipe 20

Building a Status Site with JavaScript and CouchDB

Problem

Database-driven applications can be somewhat complex. A typical database-driven application usually consists of a mix of HTML, JavaScript, SQL queries, and a server-side programming language, as well as a database server. Developers need to know enough about each of these components to make them work together. We need an alternative that's simple and lets us leverage some of the web development skills we already have, while still giving us the flexibility we need to get more complex as our needs change.

Ingredients

- CouchDB[11]
- A Cloudant.com account[12]
- CouchApp[13]
- jQuery

Solution

CouchDB is a document database and web server combined into one small, but powerful package. We can build database-driven applications using only HTML and JavaScript, and upload them right to the CouchDB server so it can serve them to our end users directly. We'll even use JavaScript to query our data, so we don't need to learn yet another language. And it just so happens we have a good excuse to play with CouchDB.

Despite our best efforts, we've been experiencing some network problems with our web servers recently. It's important to communicate this downtime to our end users, so we can keep some of the angry support calls at bay. We'll use CouchDB to develop and host a very simple site that will alert our

11. http://couchdb.apache.org/
12. http://cloudant.com
13. http://couchapp.org

end users to issues with our network. Since we could be experiencing trouble with our network, we need to host the status site on a separate network, so we'll use a service called Cloudant instead of setting up our own CouchDB server. Cloudant is a CouchDB hosting provider that gives us a small free CouchDB instance we can use for testing.

To speed up the process, we'll use CouchApp, a framework for building and deploying HTML and JavaScript applications for CouchDB. CouchApp gives us tools to create projects and push files up to our CouchDB database. But before we start hacking on our status site, let's dig into how CouchDB works.

Understanding CouchDB

CouchDB is a "document database". Instead of storing "rows" in "tables", we store "documents" in "collections". This is different from relational databases like MySQL and Oracle. Relational databases use a *relational model*, where we divide the data up into multiple entities and relate things together to reduce data duplication. We then use queries to pull this data together into something we can use. In a relattional model, a person and her address would be in separate tables. This is a fine, trusted solution, but it's not always a good fit.

In a document database, we're more concerned with storing the data as a document so we can reuse it later, and we're not all that interested in how one document relates to another. While some folks like to pit traditional relational databases and document database against each other, you'll often find that they serve completely different needs, or can actually complement each other.

For our status update system, each status update will be a CouchDB document, and we'll create a simple interface that displays these documents. Let's start by defining our database and our Status document.

Creating the Database

We'll use the web interface Cloudant provides to create a new database. When we log into our Cloudant account for the first time, we'll be prompted to create our first database. We'll call our database `statuses`.

We can also use the Cloudant interface to create a couple of status documents. Once we've selected our database, we'll see a list of documents in the database. The "New Document" button gives us a simple interface to add status messages.

Documents are just a collection of keys and values represented as JSON data. Each of our status notifications needs a title and a description, and so a JSON representation looks like this:

```
{
  "title": "Unplanned Downtime",
  "description": "Someone tripped over the power cord!"
}
```

We can either add each field to the document using the wizard, or we can choose the "View Source" button and insert the JSON directly. We could also use cURL, as discussed in *Manipulating CouchDB with cURL*, on page 119.

Let's use the GUI to add a couple of documents so we'll have something to display. We first create a new document and set a title and description for a status message. We can leave the _id field blank, as shown in Figure 20, *Adding a New Document With The Cloudant Wizard*, on page 119,

Now that we have some data in our database, let's build an interface to display it.

Creating A Simple CouchApp

CouchApps are applications that we can host from CouchDB itself. The CouchApp command-line application gives us some tools to create and manage these applications. With CouchApp, we can even push our files directly to our remove database.

CouchApp is written in Python, but there are installers for Windows and OSX we can use that don't require us to have Python on our system. Visit the installation page, get the package for your system, and install it.[14]

With CouchApp installed, we can create our first application from our shell like this:

```
$ couchapp generate app statuses
```

This creates a new folder called statuses which contains a new CouchApp. There are several subfolders within this app, each with a different purpose.

The _attachments folder is where we'll put our HTML and JavaScript code for our interface. When we push our CouchApp to our CouchDB server, the contents of this folder will be uploaded as a design document.

14. http://couchapp.org/page/installing

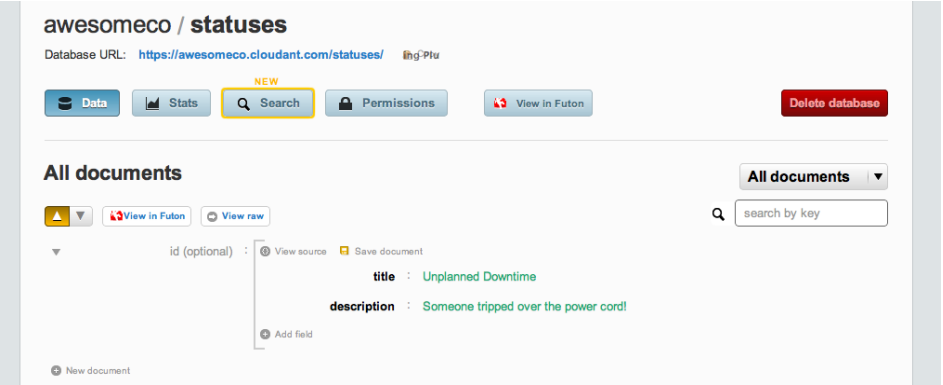


Figure 20—Adding a New Document With The Cloudant Wizard

Manipulating CouchDB with cURL

Since CouchDB uses a RESTful JSON API, we can create databases, update documents, and run queries from the command line instead of a GUI tool. We can use cURL, a command line tool for making HTTP requests to do just that. The cURL program is available on most operating systems, and might even be installed for you if you're on OSX or Linux.

For example, instead of creating our `statuses` database with the GUI, we can use cURL to send a `PUT` request like this:

```
curl -X PUT http://awesomeco:****@awesomeco.cloudant.com/statuses
```

and we can push some data like this:

```
curl -X POST http://awesomeco:****@awesomeco.cloudant.com/statuses \
  -H "Content-Type: application/json" \
  -d '{"title":"Unplanned Downtime","description":"Someone tripped over the cord."}'
```

The `-H` flag sets the content type, and the `-d` flag lets us pass a string of data to send.

With cURL, we can set up and seed our database in much less time that we could by going through a web console. We could even script it so we can do it over and over again.

The `views` folder holds CouchDB "views", which are different representations of our documents. For example, a document may contain thirty fields, but we can use views to only show the two or three fields we're interested in for a particular purpose. Views are a common component in many types of databases, even relational ones.

We can then push this app right to our CouchDB database from the command line

```
$ couchapp push statuses \
http://awesomeco:****@awesomeco.cloudant.com/statuses
```

```
2011-07-20 14:24:28 [INFO] Visit your CouchApp here:
http://awesomeco.cloudant.com/statuses/_design/statuses/index.html
```

We're pushing the statuses folder, which contains our entire app, into the statuses database, where it will be stored as a "Design document". We can look at our app in the browser at http://awesomeco.cloudant.com/statuses/_design/statuses/index.html, although it will simply show us a boilerplate "Welcome" page. Let's get to work building our actual status application now that we know how to push our files to the server.

Creating a View

We use views in CouchDB to optimize the results we want to return, rather than just querying our documents directly. When we access a view, CouchDB executes a JavaScript function we define to pare down the results and manipulate them into a data structure that works for us.

The couchapp command can create the files for our view. we'll create a messages view, like this:

```
$ couchapp generate view statuses messages
```

Download couchapps/statuses/views/messages/map.js

```
function(doc) {
  emit( "messages", {
    title: doc.title,
    description: doc.description
  } )
}
```

Each status message has a title and description, but they also contain a unique identifier and a revision number. For our status page, we only need the title and description.

We can verify that our view works by pushing the application to our database, and looking at http://awesomeco.cloudant.com/statuses/_design/statuses/_view/messages in our browser. We should see something that looks like this:

```
{"total_rows":2,"offset":0,"rows":[
  {"id":"02abeecc98362b3a26f85ea047bfaf5d","key":"messages","value":
    {"title":"Unscheduled Downtime",
    "description":"Someone tripped over the power cord!"}
```

```
  }
]}
```

With the view in place, let's whip up some HTML and jQuery code to display the status messages on our site.

Displaying The Messages

To build our simple interface, we can replace most of what's in the default page in _attachments/index.html with this:

Download couchapps/statuses/_attachments/index.html

```
<body>
  <h1>AwesomeCo Status updates</h1>

  <div id="statuses">
    <p>Waiting...</p>
  </div>

  <script src="vendor/couchapp/loader.js"></script>
</body>
</html>
```

We'll then update the contents of the statuses region with the data we pull from our database.

As we learned in Recipe 10, *Building HTML With jQuery Templates*, on page 52, we can use jQuery Templates when we're going to be building up HTML we want to add to the page. Our page includes a JavaScript file called loader.js loads up several JavaScript libraries we need to make a basic CouchApp run, including jQuery and the jQuery Couch library. We simply copy the jquery.tmpl.min.js file into the vendor/couchapps/_attachments and add it to the list of scripts, like this:

Download couchapps/statuses/vendor/couchapp/_attachments/loader.js

```
couchapp_load([
  "/_utils/script/sha1.js",
  "/_utils/script/json2.js",
  "/_utils/script/jquery.js",
➤ "vendor/couchapp/jquery.tmpl.min.js",
  "/_utils/script/jquery.couch.js",
  "vendor/couchapp/jquery.couch.app.js",
  "vendor/couchapp/jquery.couch.app.util.js",
  "vendor/couchapp/jquery.mustache.js",
  "vendor/couchapp/jquery.evently.js"
]);
```

With that in place, we can now add a simple jQuery template to our index.html page that represents the status message. The jQuery CouchDB plugin will

return an array of value objects which we'll pass on to the template, so we'll need to reference this object in our template, like this:

Download couchapps/statuses/_attachments/index.html

```
<script type="text/html" id="template">
  <div class="status">
    <h2>${ value.title }</h2>
    <p>${ value.description }</p>
  </div>
</script>
```

Next, we need to make a connection to CouchDB and fetch our status messages, which we'll feed into our jQuery template. We'll define this as a function inside of a new <script> block on our index.html page.

Download couchapps/statuses/_attachments/index.html

```
$db = $.couch.db("statuses");
var loadStatusMessages = function(){
  $db.view("statuses/messages",{
    success: function( data ) {
      $("statuses").empty();
      var template = $("#template").tmpl( data.rows );
      $("#statuses").html(template);
    }
  });
}
```

We're using the same "success" callback pattern we've used in Recipe 14, *Organizing Code with Backbone.JS*, on page 79. There's an error callback you could define yourself, but the CouchDB plugin throws up an error message for us by default.

Finally, we just need to call this function when our page loads, like this:

Download couchapps/statuses/_attachments/index.html

```
$(function(){
  loadStatusMessages();
});
```

All that's left is to push the CouchApp to our database one last time. When we visit our page in the browser again, we see our status messages, nicely rendered, just like in Figure 21, *Our Status Site*, on page 123. From here, we can continue to build this application out, making changes to the code and pushing it up to the server.

Further Exploration

We've built a very trivial, but functional, web application using only HTML and JavaScript, all hosted with a CouchDB database, but there's more we

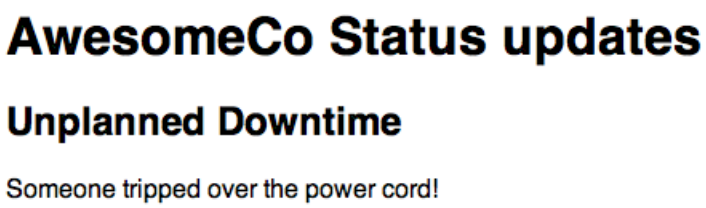


Figure 21—Our Status Site

could do. We could use JavaScript frameworks like Backbone to organize our code as things get more complex. CouchApp actually includes a framework called Evently that simplifies some of the event delegation stuff you might find in a more complex user interface.[15] While we didn't need it in our simple example, you might find that it works for you.

The URL for our application is quite long and ugly, but CouchDB has its own URL-rewriting features so we can shorten http://awesomeco.cloudant.com/statuses/_design/statuses/index.html to something less clunky, like http://status.awesomeco.com.

CouchDB isn't just a client-side data store, though. We could also integrate CouchDB into server-side applications. It's a good, solid document store that's easy to use and extend. While it may not fit every need, it certainly has its place, especially when working with data that isn't necessarily relational.

Also See

- Recipe 10, *Building HTML With jQuery Templates*, on page 52
- Recipe 13, *Snappier Cient-Side Interfaces with Knockout.js*, on page 69
- Recipe 14, *Organizing Code with Backbone.JS*, on page 79

15. http://couchapp.org/page/evently

CHAPTER 4

Mobile Recipes

More and more people access web sites and applications from mobile devices, and we need to develop with these users in mind. Limited bandwidth, smaller screens, and new user interface interactions create interesting problems for us to solve. With these recipes, you'll learn how to save bandwidth with CSS sprites, work with multitouch interfaces, and build a mobile interface with transitions.

Recipe 21

Targeting Mobile Devices

Coming soon...

Recipe 22

Touch-Responsive Dropdown Menus

Coming soon...

Recipe 23

Mobile Drag and Drop

Coming soon...

Recipe 24

Using jQuery Mobile

Coming soon...

Recipe 25

Using Sprites with CSS

Coming soon...

CHAPTER 5

Workflow Recipes

The tools and processes we use ultimately make or break our productivity. As developers, we're used to looking at better ways to make our clients happy, but we should also look at ways to improve our own workflow. This collection of recipes explores different workflows for working with layouts, content, CSS, and JavaScript, as well as our code.

Recipe 26

Rapid Responsive Design with Grid Systems

Coming soon...

Recipe 27

Creating A Simple Blog with Jekyll

Coming soon...

Recipe 28

Building Modular Stylesheets With Sass

Coming soon...

Recipe 29

Cleaner JavaScript with CoffeeScript

Problem

JavaScript is the programming language of the Web, but it's often misunderstood, which leads to poorly written and terribly performing code. Its rules and syntax can lead to developer confusion and frustration which can slow down productivity. Since JavaScript is everywhere, we can't simply remove it or replace it with a language with a more comfortable syntax, but we can use other languages to *generate* good, standard, and well-performing JavaScript.

Ingredients

- CoffeeScript[1]
- Guard[2] and the CoffeeScript addon[3]
- QEDServer

Solution

CoffeeScript lets us write JavaScript in a more concise format, similar to languages like Ruby or Python. We then run our code through an interpreter which emits regular JavaScript that we can use on our pages. While this interpretation adds an additional step to our development process, the productivity gains are often worth the tradeoff. For example, we won't accidentally forget a semicolon or miss a closing curly brace, and we won't forget to declare variables in the proper scope. CoffeeScript takes care of those issues and more so we can focus on the problem we're solving.

We'll test-drive CoffeeScript by using it with jQuery to fetch the products from our store's API. We'll use our test server, just like we did in Recipe 14, *Organizing Code with Backbone.JS*, on page 79.

1. http://coffeescript.org/
2. https://github.com/guard/guard
3. https://github.com/netzpirat/guard-coffeescript

CoffeeScript is a language of its own, which means we need to learn a new syntax for declaring things like variables and functions. The CoffeeScript web site and Trevor Burnham's *CoffeeScript: Accelerated JavaScript Development* [Bur11] explain a lot of these fundamentals in excellent detail, but let's take a look at a couple of basic CoffeeScript concepts we'll need to understand to move forward.

CoffeeScript Basics

CoffeeScript's syntax is designed to be similar to JavaScript, but with much less noise. For example, instead of declaring a function like this in JavaScript:

```
var hello = function(){
  alert("Hello World");
}
```

We can express it with CoffeeScript as:

```
hello = -> alert "Hello World"
```

We don't need to use the var keyword to declare our variables. CoffeeScript will figure out which variables we've declared and add the var statement in the appropriate place for us.

Second, we use the -> symbol instead of the function keyword to define functions in CoffeeScript. Function arguments come before the -> symbol, and the function body comes immediately after, with no curly braces. If the function body goes for more than one line, we indent it, like this:

```
hello = (name) ->
  alert "Hello " + name
```

While there are many more powerful expressive features of CoffeeScript, those two make it possible to turn something like this:

```
$(function() {
  var url;
  url = "/products.json";
  $.ajax(url, {
    dataType: "json",
    success: function(data, status, XHR) {
      alert("It worked!");
    }
  });
});
```

into this:

```
$ ->
```

```
url = "/products.json"
$.ajax url,
  dataType: "json"
  success: (data, status, XHR) ->
    alert "It worked!"
```

The CoffeeScript version of the code is a little easier on the eyes and it takes less time to write. If we made syntax errors, we'll find out as soon as we try to convert our CoffeeScript to JavaScript, which means we won't be spending time hunting these things down in the web browser.

Installing CoffeeScript

There are numerous ways to get CoffeeScript running, but the absolute simplest way to test it out is through the browser. This way we don't have to install anything on our machines to try a quick demo. We download the CoffeeScript interpreter[4] and include it on our web page like this:

Download coffeescript/browser/index.html

```
<script
  src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js">
</script>
<script src="coffee-script.js"></script>
```

Then we can place our CoffeeScript code in a <script> block, like this:

Download coffeescript/browser/index.html

```
<script type="text/coffeescript">
$ ->
  url = "/products.json"
  $.ajax url,
    dataType: "json"
    success: (data, status, XHR) ->
      alert "It worked!"
</script>
```

This in-browser approach is great for experimenting but it's not something you'll ever want to roll out in production because the CoffeeScript interpreter is a very large file, and interpreting the CoffeeScript on the client machine is going to be much slower. We want to convert our CoffeeScript files ahead of time and only serve the resulting JavaScript files from our website. For that, we'll need to install a CoffeeScript interpreter and we'll need a good workflow to go along with that.

4. http://jashkenas.github.com/coffee-script/extras/coffee-script.js

People usually install the CoffeeScript interpreter with Node.JS and NPM, the Node Package Manager.[5], but we can also use it with Ruby. Since we've used Ruby for other recipes, we'll go that route. Assuming you've installed Ruby following the instructions in Appendix 1, *Installing Ruby*, on page 187, you can type this on the command line:

```
$ gem install coffee-script guard guard-coffeescript
```

This installs the CoffeeScript interpreter and Guard, which we'll use to automatically convert CoffeeScript files to JavaScript files whenever we make changes to them. The guard-coffeescript gem does this automatic conversion for us.

Let's set up our project and get a demo going.

Working With CoffeeScript

We'll use QEDServer and its product management API as our development server. We'll place all of our files in the public folder in our workspace so our development server will serve them properly, and our AJAX requests will work without issue.

Since we're going to turn CoffeeScript files into JavaScript files, let's create folders for each of those types of files.

```
$ mkdir coffeescripts
$ mkdir javascripts
```

Now, let's create a very simple web page that loads up jQuery, the jQuery.tmpl library which we learned about in Recipe 10, *Building HTML With jQuery Templates*, on page 52, and app.js, which will contain the code that fetches our data and displays it on the page:

Download coffeescript/guard/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <script
      src="https://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js">
    </script>
    <script src="javascripts/jquery.tmpl.min.js"></script>
    <script src="javascripts/app.js"></script>
  </head>
  <body>
  </body>
</html>
```

5. http://npmjs.org/

We'll need to place jquery.tmpl.min.js in the javascripts folder, but app.js will be generated from CoffeeScript when we're done.

Now we'll add a simple jQuery template to the page which we'll use to display each recipe.

Download coffeescript/guard/index.html

```
<script id="product_template" type="text/html">
  <div class="product">
    <h3>${ name }</h3>
    <p>${ description }</p>
  </div>
</script>
```

Next, we'll create the file coffeescripts/app.coffee.

Download coffeescript/guard/coffeescripts/app.coffee

```
$ ->
  $.ajax '/products.json',
    type: 'GET'
    dataType: 'json'
    success: (data, status, XHR) ->
      $('#product_template').tmpl(data).appendTo("body")
    error: (XHR, status, errorThrown) ->
      $('body').append "AJAX Error: #{status}"
```

Instead of the unfriendly jQuery shortcut $(function(){}), we simply use $ -> to define the code that runs when the document is ready. We define a variable for our URL and we call jQuery's AJAX() method, rendering the jQuery template if we get a response, or displaying the failure message when we don't. The logic and flow is identical to a pure JavaScript implementation, but it's several lines fewer. Of course, this code won't work yet because our page is requesting a JavaScript file that we still need to generate. We'll do that with Guard.

Using Guard to Convert CoffeeScript

Guard is a command-line tool we can configure to watch files for changes and then perform tasks in response to those changes. The guard-coffeescript plugin gives Guard the ability to convert our CoffeeScript files.

We need to tell Guard to watch files within the coffeescript folder for changes and convert them to JavaScript files, placing those in the javascripts folder. We do that by creating a file called Guardfile in the root of our project, which tells Guard how to handle our CoffeeScript files. We can create this file by hand, or by running:

```
$ guard init coffeescript
```

Then we open the newly generated Guardfile and change the input and output folders so they point to our folders:

Download coffeescript/guard/Guardfile

```
# A sample Guardfile
# More info at https://github.com/guard/guard#readme

guard 'coffeescript', :input => 'coffeescripts', :output => "javascripts"
```

Now we can start up Guard from our shell, and it will start watching our coffeescripts folder for changes:

```
$ guard
Guard is now watching at '/home/webdev/coffeescript/public/'
```

When we save the coffeescripts/app.coffee file, Guard will notice and do the conversion from CoffeeScript to JavaScript:

```
Compile coffeescripts/app.coffee
Successfully generated javascripts/app.js
```

When we view the page at http://localhost:8080/index.html, everything works! If we inspect the generated app.js file, we'll see that all of the required curly braces, parentheses, and semicolons are where they should be. We now have a workflow we can use to write better JavaScript, so we can continue making changes to our application. When we're doe, we can deploy the javascripts folder and leave the coffeescripts folder in our source code repository.

Further Exploration

More and more JavaScript projects are moving to CoffeeScript as a development platform to develop their projects due to its ease of use, and because it provides some of the niceties of languages like Ruby, including implicit returns and string interpolations. CoffeeScript itself is actually *written* in CoffeeScript. After all, CoffeeScript's output is regular, standard JavaScript that works anywhere JavaScript works. You could continue experimenting with CoffeeScript by attempting to implement some of the recipes in this book in CoffeeScript.

In addition to CoffeeScript, Guard also has support for Sass, which we talk about in Recipe 28, *Building Modular Stylesheets With Sass*, on page 134 via the guard-sass gem. Using Guard, Sass, and CoffeeScript together gives you an incredibly powerful workflow for managing your sites. And if you really want to integrate CoffeeScript into your web development workflow, you could use a tool like MiddleMan which makes building static sites with Sass

and CoffeeScript a breeze.[6]. Combining that with an automated deployment strategy like the one we talk about in Recipe 42, *Automate Static Site Deployment with Guard and Rake*, on page 185 can create an efficient and enjoyable development experience.

Also See

- *CoffeeScript: Accelerated JavaScript Development* [Bur11]
- Recipe 28, *Building Modular Stylesheets With Sass*, on page 134
- Recipe 14, *Organizing Code with Backbone.JS*, on page 79
- Recipe 42, *Automate Static Site Deployment with Guard and Rake*, on page 185

6. http://middlemanapp.com/

Recipe 30

Managing Files Using Git

Problem

As web developers we often find ourselves in situations where we're asked to juggle multiple versions of our code. Sometimes we need to experiment with the latest and greatest plugin. Then there are the times where we're in the zone, cranking away on a new feature, but then get sidetracked because we need to fix a critical bug. We all use some form of version control, even if it is just keeping multiple copies of a file. But that multiple-file situation breaks down pretty fast because it's all on our machine and isn't easy to manage. We need something that's fast, robust and modern; something that we can use to manage our code, but also something we can use to collaborate with others.

Ingredients

- Git[7]

Solution

Today we have many options for version control. Git is very popular among developers, because it's local and fast, faster than making local copies. Git also allows us to work on multiple versions in parallel. We can save changes often, giving us many restore points. All of these features make it the choice VCS for many of today's open source projects.

During our morning meeting, our boss turned to us and said "I need you to take those two mocks you presented last week and develop actual versions of the site using those templates. Oh, and while you're working on that, we also need a few bugs fixed in the existing site."

Now we have three versions of our site to maintain. Let's use Git to keep our files organized and in sync.

7. http://git-scm.com/

Setting Up Git

Let's get started by installing Git. Head over to Git's website[8] and download the appropriate packages for your operating system. If you're running Windows, you should use MsysGit,[9] and you'll want to choose the option to use Git Bash, since you'll need to use that instead of the normal Command Prompt to follow along with this recipe.

Git tracks the person who made the change based on their configured Git username. This makes it easy to see who made what changes and when. Let's configure Git by specifying our name and email address. Open a new shell and type the following:

```
$ git config --global user.name "Firstname Lastname"
$ git config --global user.emphail "your_emphail@youremphail.com"
```

Now that we have Git installed and configured, let's get comfortable with the basics.

Git Basics

Let's start by turning our project into a Git repository. Let's create a folder for this web project called git_site, and initialize it as a Git repository. From the command line (or from Git Bash if you're on Windows), type:

```
$ mkdir git_site
$ cd git_site
$ git init
```

After we initialize the directory we will get a confirmation message:

```
Initialized empty Git repository in /Users/webdev/Sites/git_site/.git/
```

This creates a hidden folder called .git in the root of our directory. All of the history and other details about our repository will all go in this folder. Git will track changes to our folder and store "snapshots" of our code, but first we have to tell Git what files we'd like to track.

Let's copy our web site files into our new git_site folder. You can find these files in the git folder of the book's source code.

With the files in place, let's add them all to the repository so we can get back to where we started if something goes wrong. To add all the files simply issue the command:

```
$ git add .
```

8. http://git-scm.com/
9. http://code.google.com/p/msysgit/

The add command doesn't show anything; we need to use the a git status. We can use a Git status command at anytime to see the current status of our Git repository.

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "Git rm --cached <file>..." to unstage)
#
#       new file:   index.html
#       new file:   javascripts/application.js
#       new file:   styles/site.css
#
```

This is called "staging our files". This way we can see what is ready to be committed and can have one more chance to change our minds before we commit the files to the repository. Staging files just means that Git is ready to look for changes. Everything looks good so let's actually commit these files.

```
$ git commit -a -m "initial commit of files"
```

The two flags we passed in were a and m. The -a tells Git that we want to add all the changes to the index before committing, and the -m specifies a commit message. Unlike other version control systems, Git requires that every commit must have a commit message. This helps greatly when tracking commits, so don't take commit messages lightly! After our commit finishes we'll get confirmation of what it did as shown in the code snippet below.

```
                            [master (root-commit) 94c75a2] Initial Commit
1 files changed, 17 insertions(+), 0 deletions(-)
create mode 100644 index.html
create mode 100644 javascripts/application.js
create mode 100644 styles/site.css
```

We can verify that the files were committed with git status When we run that, we see that everything is up to date:

```
# On branch master
nothing to commit (working directory clean)
```

We now have a "snapshot" of our code, which means we can start making and tracking changes.

Working with Branches

Branching allows us work on multiple features of our website. We can effectively develop a new feature while maintaining our current deployed code. Unlike other VCS branching is an easy and very commonly used feature of Git.

Our boss wanted us to start work on implementing two site layouts, which we'll call *layout_a* and *layout_b*. Let's create a branch for *layout_a*.

```
$ git checkout -b layout_a
Switched to a new branch 'layout_a'
```

Now when we run git status we see that our current branch is *layout_a*. Let's go into the index.html file and change the text in the <h1> tag to say "Layout A" and save the file. Now when we do Git status we see:

```
# On branch layout_a
# Changed but not updated:
#   (use "Git add <file>..." to update what will be committed)
#   (use "Git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   index.html
#
no changes added to commit (use "Git add" and/or "Git commit -a")
```

Let's commit the changes to the *layout_a* branch.

```
$ git commit -a -m "changed heading to Layout A"
```

While we were working on our branch, our boss sent us an email that says "On the home page, it says that we offer one day shipping. We no longer offer that shipping promotion. We need to update it to two-day shipping and we have to do it right now before anyone else holds us to that option!." Let's switch back to our *master* branch and get that change made.

```
$ git checkout master
```

Now when we open the index.html we won't see the text we changed in the layout_a branch. The changes we made are in another branch, and instead of us moving files around, we simply let Git alter the file's contents when we change branches. Now we can make the changes to the home page that our boss wanted us to and then we can commit back to the *master* branch.

```
$ git commit -a -m "fixed shipping promotion from one day to two-day"
  [master d00d2de] fixed shipping promotion from one day to two-day
  1 files changed, 1 insertions(+), 1 deletions(-)
```

Joe asks:
Why are we committing changes so often?

Think of commits as snapshots, or restore points, for your project. The more commits you make, the more powerful and flexible Git becomes. If we keep our commits small and focused on a particular feature, we can use Git's "cherry-pick", which lets us take a commit from one branch and apply it to other branches. And if the idea of lots of small commits seems messy to you, you can always squash commits together using rebase when you've completed a feature..

We made this change on the *master* branch, but if we change back to our other branches, we won't see the change, and it would be a Really bad Thing if we lost the work we just did by accident, so let's get these changes into our *layout_a* branch so we can get back to working on that layout.

```
$ git checkout layout_a
$ git merge master
```

This takes anything that wasn't changed in *layout_a* but was changed in *master* and applies it to the *layout_a* branch.

Next let's create a branch for our *layout_b* option. We want this to start off based on our current production site, not our *layout_a* version, so we need to switch back to the *master* branch and then create a branch for *layout_b*.

```
$ git checkout master
$ git checkout -b layout_b
```

This time we'll change the text inside of the <h1> tag to say "Layout B". Let's save and commit this change.

```
$ git commit -a -m "Changed heading to Layout B"
```

This version of our layout requires us to add a products.html file, and a about_us.html. Let's create those files and then stage those files for check-in.

```
$ touch products.html
$ touch about_us.html
$ git add .
```

Now if we do a git status we'll see that we have two new files that have been staged.

```
# On branch layout_b
# Changes to be committed:
#   (use "Git reset HEAD <file>..." to unstage)
#
#       new file:   about_us.html
```

```
#       new file:   products.html
#
```

Let's commit those files.

```
$ git commit -a -m "added products and about_us, no content"
```

Now, let's add an <h1> to the products.html with the text of "Current Products" to comply with our design.

While we were doing that, we just got another email from our boss that says "We need to change the shipping time on the home page back to one day. We struck a deal with a major shipping company. Get these changes made ASAP!!!". We need to make these changes and get them pushed out right away. However we are not ready to commit the changes we just made.

Git's stash command is meant for situations like this. We can use stash to store our changes so we can switch branches. Stashes are a great way to store something you are working on without actually having to commit themph.

```
$ git stash
```

Now if we do a Git status we will see that there are no changes that need to be committed. Let's switch over to our *master* branch:

```
$ git checkout master
```

Now we can make our changes to the shipping information index.html and commit the changes.

```
$ git commit -a -m"updated shipping times"
```

Let's switch back over to our *layout_b* branch with Git checkout layout_b and explore what we can do with stashes. Let's see what stashes are available by using the Git stash list command:

```
$ git stash list
stash@{0}: WIP on layout_b: f8747f4 added products and about_us, no content
```

When we open up our products.html, we see that it's empty. Let's get the changes we made to that file back. We do that with this command:

```
$ git stash pop
```

Now when we look at our products.html file, we'll have our <h1> tag that we added before we got sidetracked.

After several more tweaks to both layouts (and several other "important" distractions, our boss decided that the *layout_b* option was the best and

wants to roll that out into production. Let's merge this work into our *master* branch.

```
$ git checkout master
$ git merge layout_b
$ git commit -a -m "merged in layout_b"
```

In traditional version controls, it's common to leave branches in a repository indefinitely. Git differs from this in that both branches and tags refer to a commit. With Git, when we delete a branch, Git does not remove any of the commits; it only removes the reference. Since we have merged our changes back into *master*, we can delete our branches that we used for development. First let's look at the branches we currently have by using the `git branch` command. It shows us that we are on *master* and lists *layout_a* and *layout_b*. Let's delete those branches, like this:

```
$ git branch -d layout_a
$ git branch -d layout_b
```

Git also will tell us if the branch has not been merged into the current branch. We can override this by using `-D`, which will force delete the branch.

Working with remote repositories

So far we have only worked with a local repository. While it is great to keep our local code under version control, having a remote repository allows us to collaborate with others and keep our code in two places.

Let's setup a remote git server on our development vm we created in Recipe 37, *Setting up a Virtual Machine*, on page 173. We can save ourselves the extra step of having to type our password whenever we log in or transfer files by creating SSH keys. Creating an SSH key and placing it on the server will allow us to authenticate quickly and without a password every time we want to push to our remote repository.

SSH keys consist of two components: a private key which we keep to ourselves, and a public key we give to another server. When we log in to that server, it checks to see if our key is authorized, and then our local system proves we are who we say we are by matching the public key with the private key. With Git, this handshaking process is all done transparently during the login process.

Before we continue, you should check to see if you have any SSH keys on your system. Try and change directories into `~/.ssh`. If you get a message saying the directory doesn't exist, then you'll need to generate keys. If you

see files like id_rsa and id_rsa.pub, then you already have keys and you can skip the next step.

Let's generate a new SSH Key with the ssh-keygen command. We'll pass our email address which will be placed into the key as a comment.

```
$ ssh-keygen -t rsa -C "webdev@awesomeco.com"
```

The comment will help us or other server administrators quickly identify who owns the key when it is uploaded to a server.

The ssh-keygen program will ask you for a place to store the SSH Key; you can simply hit the Enter key to save it in the default location. It will also ask you to enter a passphrase. This adds an additional layer of security to the key, but we'll just leave it blank for now. Simply press the Enter key again.

Now that we have our keys generated, let's add them to our VM. We can pipe our local public key into the file authorized_keys on the server. This lets the VM know that our machine can have access.

```
$ cat ~/.ssh/id_rsa.pub | ssh webdev@192.168.1.100 \
"mkdir ~/.ssh; cat >> ~/.ssh/authorized_keys"
```

After executing this command our server will ask for our password to make sure this is a legitimate request. After the command finishes. We can test our key by trying to SSH into the VM:

```
$ ssh webdev@192.168.1.100
```

and this time it will not ask us for our password.

Now that we're logged in to our VM, we use Ubuntu's package manager to install Git on the server:

```
$ sudo apt-get install git-core
```

And now we can create a "bare" repository on the VM, which is nothing more than a directory, usually with a .git extension, since this makes it easier for us to identify these. Then, inside the directory, we use the git command to initialize the folder, using the --bare switch.

```
$ mkdir website.git
$ cd website.git
$ git init --bare
```

With the repository created on the remote machine, we can log out of the VM by typing exit.

Back on our local machine let's add the location of our remphote repository and push up our *master* branch.

```
$ git remote add origin ssh://webdev@192.168.1.100/~/website.git
$ git push origin master
```

Let's say we wanted to work on a new feature with another developer. We can create a branch for this new feature called *new_feature*, then work on our design implementation. Once our design work is done, we can push the branch to the remote repository.

```
$ git checkout -b new_feature
$ git push origin new_feature
```

Now that we've pushed our branch, let's see what branches are out on the remote repository.

```
$ git branch -r
```

We'll end up with a list of branches. We won't see *layout_a* and *layout_b* because we deleted them locally and we never pushed them out.

```
origin/HEAD -> origin/master
origin/new_feature
origin/master
```

To get our developer access to our Git repository we can have him clone the full project. After he clones the whole project we can have him checkout the *new_feature* branch. Lastly he can make sure he is up to-date on the project by pulling the remphote branch from the server into his local branch.

```
$ git clone ssh://webdev@192.168.1.100/~/website.git
$ git checkout -b new_feature
$ git pull origin new_feature
```

With the branch on the developer machine also the cycle begins again. Git gives us the power to work side by side on the same code and merge the changes with ease, like we did locally earlier in the chapter.

Further Exploration

Now that you have explored the basics of Git, you might start seeing other uses for it. In this recipe, we only worked with text files, but Git supports any type of file. You could use Git to version control your Photoshop documents, so you can easily maintain multiple versions as you build out designs. You can explore how to pull out previous versions of files, so you can recover that change your boss didn't like last week but wants to look at one more time.

You can also use Git to collaborate on open source projects with others. For example, you can go out to GitHub[10] and find an open source project like jQuery or one of the other libraries you've learned about in this book, and clone it, which pulls it down to your computer as a Git repository. You can then use techniques like branching to develop new features for that project which you could then submit these new features back to the original maintainers to help the community grow.

Also See

- Recipe 36, *Using Dropbox to host a static site*, on page 168
- Recipe 37, *Setting up a Virtual Machine*, on page 173
- *Pragmatic Version Control Using Git* [Swi08]

10. http://www.github.com

CHAPTER 6

Testing Recipes

We need to ship, but we have to ship code that *works*. We often ensure our apps do what we want them to do by testing them in the browser manually. Sometimes we'll get other people to test things for us. In these recipes, we'll explore how to debug our code as we build it, and also how to create repeatable acceptance tests that we can run whenever we make changes to our code so we can see if things still work the way they did before.

Recipe 31

Debugging JavaScript

Coming soon...

Recipe 32

Tracking User Activity with Heatmaps

Problem

When running a promotion or redesigning a site, it's helpful to know what works and what doesn't so we know where to spend our time. We need to quickly identify the most used regions of our page.

Ingredients

- A server running PHP
- ClickHeat[1]

Solution

We can track where our users click on the page and displaying the results in a graphical overlay called a *heatmap*, giving us an at-a-glance idea of the most-used parts of our page. While there are several commercial products that can create heatmaps based on user activity, we'll use the open-source ClickHeat script because setting it up on modern web hosts is almost as easy as using a commercial solution.

Let's use ClickHeat to solve an internal dispute. One of our clients is launching a new product, and the two partners are at odds on whether the "Sign Up" or "Learn More" buttons are actually useful. These buttons are placed right next to each other on the interface. We can easily add some tracking to this page to see which one is getting pressed more.

Setting Up ClickHeat

ClickHeat needs PHP to work, but we can use it against any web site we want. We just need to download ClickHeat from the project's web page and place ClickHeat's scripts in a PHP-enabled folder on our server. For this recipe, we'll use a virtual machine running on our own network at http://192.168.1.100. Check out Recipe 37, *Setting up a Virtual Machine*, on page 173 to learn how to build your own virtual machine for testing.

1. http://www.labsmedia.com/clickheat/index.html

When we unzip the ClickHeat archive, we'll find a clickheat folder. We'll upload this folder into /var/www, the folder on our virtual machine that contains our existing web pages. Since our virtual machine has SSH enabled, we can copy the files up with a single command by using scp:

```
scp -R clickheat webdev@192.168.1.100:/var/www/clickheat
```

Or we can transfer them over to the server's /var/www folder with an SFTP client like FileZilla.

Once we've copied the code out to the server, we need to modify the permissions on a few folders within the clickheat folder structure so that we can write the logs and modify permissions. We'll log into our server and use the chmod command to make the config, tmp, and logs folders writeable:

```
$ ssh webdev@192.168.1.100
$ cd /var/www/clickheat
$ chmod -R 777 config logs cache
$ exit
```

With the files in place, we can complete the configuration by browsing to http://192.168.1.100/clickheat/index.php. ClickHeat will verify that it can write to the configuration folder, and we'll be able to follow the link to configure the rest of the settings.

We can leave the values alone for this case, but we'll enter values for the Administrator username and password. Once we press the "Check Configuration" button and we see no errors, we're done configuring ClickHeat. Now let's attach it to our web page so we can capture some data.

Tracking Clicks And Viewing Results

To begin tracking clicks, we simply need to add a few lines of JavaScript to our home page, right above the closing <body> tag:

Download heatmaps/index.html

```
<script type="text/javascript"
  src="http://192.168.1.100/clickheat/js/clickheat.js"></script>
<script type="text/javascript">
  clickHeatSite = 'AwesomeCo';
  clickHeatGroup = 'buttons';
  clickHeatServer = 'http://192.168.1.100/clickheat/click.php';
  initClickHeat();
</script>
```

We're defining a "Site" and a "Group" for this heatmap. This lets us track multiple sites.

When we redeploy our page to our server, clicks from our users will be recorded to ClickHeat's logs. After a few hours, we can visit http://192.168.1.100/clickheat/index.php to see the results of our test, which look similar to Figure 22, *Our Heatmap*, on page 158.

It looks like more people are clicking the "Sign Up!" button!

Further Exploration

ClickHeat is relatively low-maintenance once it's running, but there are a lot of options we can adjust, such as the number of times we'll record a click from the same user. We can also configure ClickHeat to record its results to the Apache logs and then parse them out with a script, which is a great approach for servers where PHP might be too slow to invoke on each request. Finally, ClickHeat can be set up on its own server, so it can collect data from more than one site or domain. Check out the documentation at the ClickHeat web site for more options, or just explore its interface.

If you'd like something with a little more power, you might want to investigate hosted commercial solutions like CrazyEgg[2] which has similar functionality.

Finally, when you're looking at heatmaps of your own sites, you might get a little unexpected guidance from your users. If you notice a bunch of click activity on part of your page that doesn't have a link, consider making that region active. Heatmaps can often times show you things you never saw before.

Also See

- Recipe 37, *Setting up a Virtual Machine*, on page 173

2. http://www.crazyegg.com/

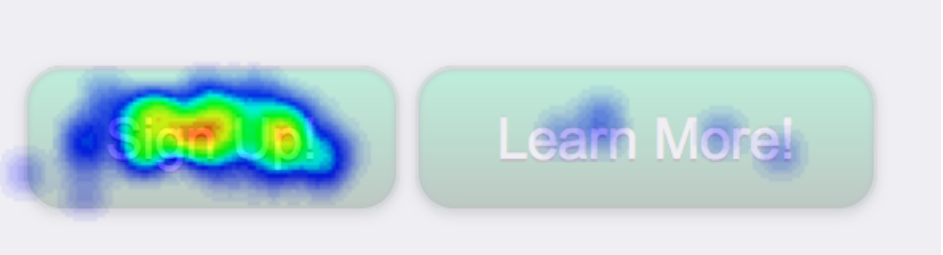


Figure 22—Our Heatmap

Recipe 33

Browser Testing With Selenium

Problem

Testing is a hard and tedious process. As websites become more complex, it becomes more important to have tests that are repeatable and consistent. Without automated testing, our only chance at having a consistent working website was to have a top-notch QA person that worked long hours and had very long checklists. That process could be painfully slow. We need to speed up the testing process and create tests we can run on-demand so that we can verify things work the way we want today, as well as several months from now when we start adding new features.

Ingredients

- Firefox[3]
- Selenium IDE[4]
- QEDServer (for our test server)*QEDServer*, on page iv

3. http://getfirefox.com
4. http://seleniumhq.org/download/

Solution

We can use automated tools to test our web projects in addition to manual testing. The Selenium IDE plugin for Firefox lets us build tests in a graphical environment by recording our actions as we use a website. As we move through a site, we can create "assertions", or little tests that ensure that certain things exist on the pages. We can then play them back any time we wish, creating a set of automated, repeatable tests.

Our development team has built a product management website, and our boss wants some safeguards in place to make sure this will always work. The development team has added some unit testing to their business logic underneath but we're tasked with building some automated tests for the user interface. Automated testing will give both the development team and us peace of mind if we make changes to the UI down the road.

Setting Up Our Test Environment

First, we need to install the Firefox web browser. Go to the Firefox web site and follow the instructions for your operating system.

Once we have Firefox working, we need to get the Selenium IDE installed. Open up Firefox, visit the Selenium web site [5] and download the latest version. [6]

With the tools installed, let's write our first test.

Creating Our First Test

We'll create our test by recording our movements with the Selenium IDE against our test server, which we'll run on our own machine using QEDServer. Start QEDServer and then launch Firefox. Go to http://localhost:8080 to bring up the test server, where you'll see an interface like the one in Figure 23, *Our Product Management Screen*, on page 160.

Since this is a product management application, we'll start off with a test to make sure we always have the "Manage products" link on the homepage, and that the link goes where we expect it to go.

Open up the Selenium IDE by selecting it from the "Tools" menu in Firefox. To start recording, we need to make sure the "Record" button is active. Then, in the browser, we click on the "Manage products" link. As we click the link,

5. http://seleniumhq.org/download/
6. If you have Firefox 4, you may need to install the Add-On Compatibility Reporter Extension version 0.8.2

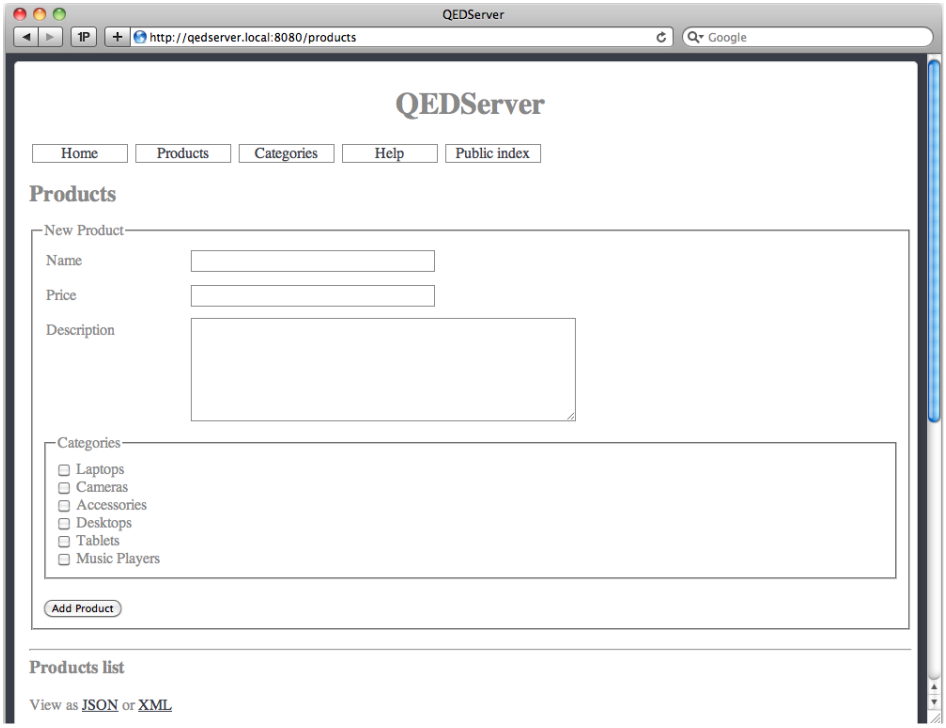


Figure 23—Our Product Management Screen

we'll see some items begin to show up in Selenium IDE, just like in Figure 24, *Our First Test With Selenium IDE*, on page 161. At the top, the Base URL is now set to http://localhost:8080, and then we see one of the most useful commands in the Selenium IDE: the clickAndWait() method. When we use web applications, we spend a lot of time clicking on links or buttons and waiting for pages to load. That is exactly what this command does. Every time we click a link, the Selenium IDE adds this method to our test along with some text that identifies the link. When we play the test back, it uses this method and the associated link text to drive the browser.

The Selenium IDE shows us the three parts of a Selenium test action. The first is Command, which is the action that Selenium is performing. The second is Target, which is the item that Selenium is performing the action on. Third is Value, which we'll use to set a value for fields that take inputs, such as when we're filling out a text box or selecting a radio button.

A powerful part of Selenium is its locator functions. We can use these to find an element on the page not only by its id, but also via the DOM, an

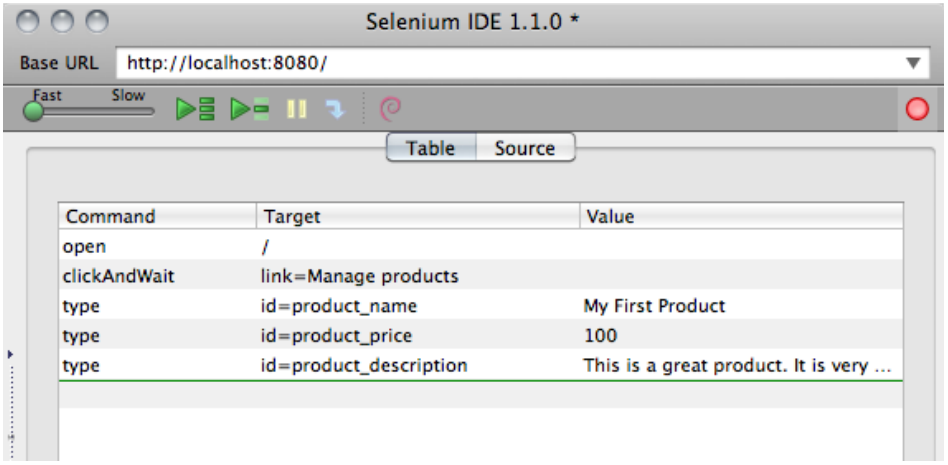


Figure 24—Our First Test With Selenium IDE

XPath query, a CSS selector, or even plain text. When we clicked on "Manage products", the target we used is link= Manage products. The link= is the selector that allows us to choose a block of text to perform an action on. One thing we should keep in mind is that locators default to looking for an ID first followed by that string of text. Identifying elements for testing with IDs is a great way to speed up your tests and improve accuracy, but it can make tests harder to read.

Now that we have an understanding of locators, let's look at what the commands do. Commands are the actions that Selenium performs when we run a test. Selenium can do anything a human would do, with one small exception - it can't upload a file without some significant modifications. The ability to manipulate a browser the way a human would allows us to simulate human interaction, giving our tests the ability to flex our code realistically.

Let's test that once we click on "Manage products" we are taken to a page where "Products" is present. First we need to click on the "Manage products" link. Next we want to make sure the word "Products" is on the screen. To add a test for that, locate "Products" on the web page and right-click on it. Choose the command verifyTextPresent from the context menu. We could also do this by using the Selenium IDE and clicking in the white space just below the clickAndWait() command and using the form fields to choose our command, target and value, but the Selenium IDE adds some test helpers to the context menu, which makes the process much faster.

We can save this test by choosing Save Test Case from the Selenium IDE's File menu. We can then run the test by clicking the play button below the Base URL window. As the test runs, the browser moves through our pages and the background color for each step changes to green as it passes. If a step fails, it will turn red, and will also show some bold red text in the log window below that with descriptions of what went wrong. This visual cue lets us know something went wrong so we can address it.

Creating an Advanced Test

We want to make sure that our product management application functions, and we can create a new product and delete a product. We also want to make sure we can view the details of a product. This is a multi-step process; let's use Selenium to automate it.

Let's go back to the home page at http://localhost:8080 and start up the Selenium IDE. Click on the "Manage products" link and wait for the page to load. Then select the "New Product" text, right-click on it and select "verifyTextPresent New Product". Next, leave the form blank and click the "Add Product" button. Our application requires that we fill in at least some of the product details, so we now have a form that did not submit, along with an error message on the screen.

Let's make this part of our test. Right-click on the "The product was not saved" error message and use the "verifyTextPresent" command to add an assertion that verifies that the error message show up on the Products page.

Now that we've shown that our error message works, we can now fill out all of the information in the form and submit it. The Selenium IDE adds a row to our test for each field we fill out. It also shows the value we typed in.

When we submit the form this time, it takes us back to the products page where we'll see the message "Created". We can use use the verifyTextPresent() command again to make sure this text is displayed.

Now we have a feature-rich example that we can save and run later. If anyone changes the site, we'll know what's broken, simply by replaying the test.

Further Exploration

Now that we have test coverage we can take this to the next level by automating our entire test suite. We currently have to run each test individually by loading it into the Selenium IDE, and this breaks down when we have a lot of tests. You'll want to investigate Selenium Remote Control and Selenium

Grid[7], which let you build automated test suites that run against multiple browsers.

And while Selenium IDE is primarily a testing tool, you could use it as an automation tool as well. For example, if you have a process that has a less-than-friendly user interface, such as a time-tracking system or a repetitive and clunky management console, you might try using Selenium IDE to save you some keystrokes and mouse clicks.

Also See

- Recipe 34, *Cucumber-driven Selenium Testing*, on page 164
- Recipe 35, *Testing JavaScript with Jasmine*, on page 165

7. http://selenium-grid.seleniumhq.org/

Recipe 34

Cucumber-driven Selenium Testing

Coming soon...

Recipe 35

Testing JavaScript with Jasmine

Coming soon...

CHAPTER 7

Hosting and Deployment Recipes

We want to get our work out there for others to see, but that's only the beginning. Once our sites are live, we have to make sure they're secure. In this collection of recipes, you'll learn how to deploy your work, and how to work with the Apache web server to redirect requests, secure content, and host secure sites.

Recipe 36

Using Dropbox to host a static site

Problem

We need to collaborate on a website with another person who is working remotely. The remote person does not have VPN access to our server farm, and our firewall only allows deployment from within our network. It would also be nice to have a publicly accessible URL so we can share our work with others.

Ingredients

- Dropbox[1]

Solution

We can use Dropbox to collaborate on static HTML files and share them with external users. With Dropbox we don't need to worry about firewalls, ftp severs or emailing files. Because Dropbox is cross platform we don't have to waste time with different applications for each OS, making Dropbox a productivity win.

Our company and our partner company, AwesomeCableCo, are sponsoring "Youth Technology Days". AwesomeCableCo have their own designer, Rob. We need a way to work with Rob on this site, and show our bosses the progress we are making.

Let's walk through the installation process so we can document it and send it off to Rob. Let's head to Dropbox's website and get the installer.

After the installer finishes we can go to the Dropbox folder on our local computer. Dropbox automatically creates a "Public" folder we can use to distribute files with anyone in the world. Let's make a "youth_tech_days" folder inside of that public folder.

Now that we have a workspace, we need to invite Rob to collaborate on files in this folder with us. When we right-click on the folder we created, we'll

1. http://www.dropbox.com

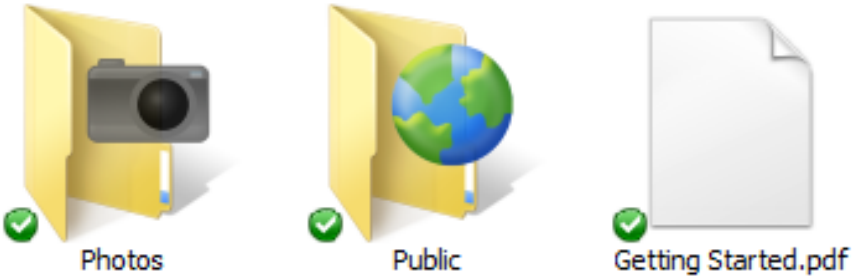


Figure 25—The folders Dropbox creates

see a context menu that gives us the option to share this folder, like the one in Figure 26, *Sharing context menu*, on page 170.

When we choose "Share This Folder", we get taken to the Dropbox website to finish the sharing process, as shown in Figure 27, *Dropbox online sharing form*, on page 170. We simply fill out the information to share this folder with Rob.

Now we can move the files for the web site into the `youth_tech_day` folder, which you can find in the book's source code in the `dropbox` folder. Now we will have a directory that looks like Figure 28, *Youth Technology Days website files*, on page 171

Whenever we drop files into this folder, they'll show up on Rob's computer as well. When anyone with access to the files makes changes to the files, everyone else's copy will stay in sync. We'll want to communicate with Rob anytime we are going to be working on a file so that we don't overwrite his work. Dropbox tries to keep us from stomping on each other's toes by identifying items that may be conflicting by appending a message to the file name. Ideally, if we are doing heavy active collaboration we would also be using Git, as mentioned in Recipe 30, *Managing Files Using Git*, on page 142
.

Now we just need to show our bosses what we've done. Since we put the files in the `public` folder they are available on the web to anyone who knows the URL. To find the address of our index file we simply need to right-click on it and choose "Copy public link" which will put the url on our clipboard. We can test this url by opening it from a browser. We will have a url similar to http://dl.dropbox.com/u/33441336/youth_tech_days/index.html.

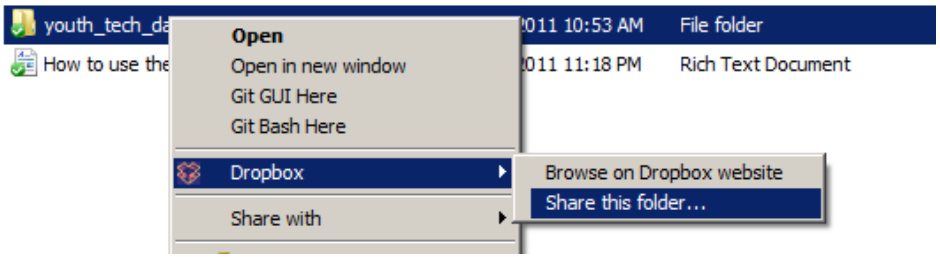


Figure 26—Sharing context menu

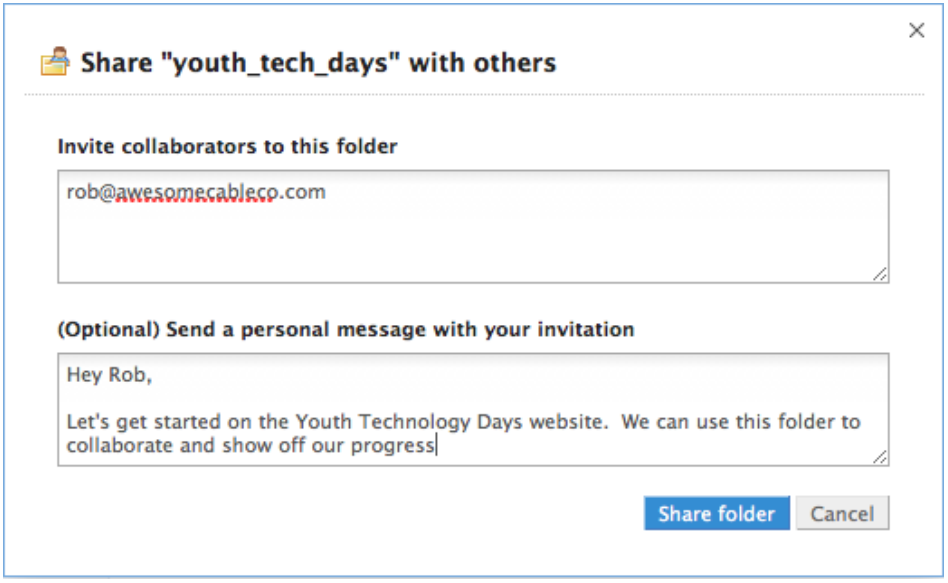


Figure 27—Dropbox online sharing form

This is a great, simple way to collaborate with people outside our company and easily show progress without the need for an FTP server, Web server or VPN connection. We can add other contributors to our project and share the URL with anyone that is interested in the progress we are making.

Further Exploration

We can further explore by sharing non-public folders with co-workers and friends. We can also use this tool to back up files, and share them among several of our own computers. We can also use the public folders to send Mom an Internet Explorer patch she just can't seem to find, or provide our

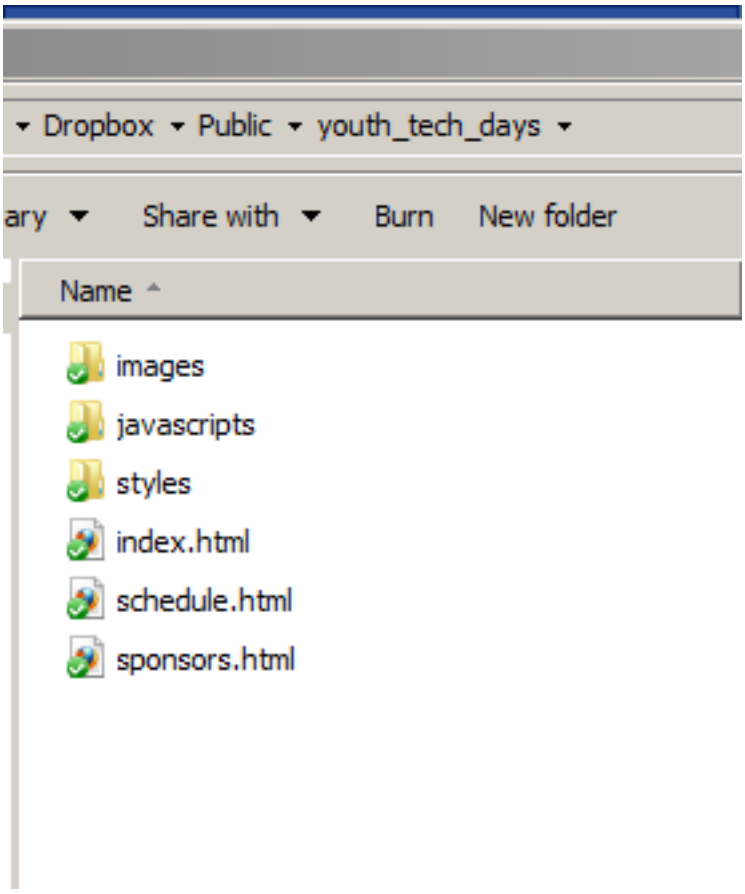


Figure 28—Youth Technology Days website files

clients with a place to send us photos or other assets they'd like us to post on their sites. Other uses include:

- Hosting files you want to share on a blog post
- Sharing a folder with each of your clients for easy collaboration
- Forwarding a vanity domain to a public site
- Creating a blog with Jekyll and hosting it from Dropbox

If your registrar or DNS provider supports redirection, you could set up a URL that's easier for people to remember when they want to check out your pages on Dropbox.

Also See

- Recipe 30, *Managing Files Using Git*, on page 142
- Recipe 27, *Creating A Simple Blog with Jekyll*, on page 133

Recipe 37

Setting up a Virtual Machine

Problem

We need to set up a local server that looks like our production server so that we have a place to test PHP scripts and configurations in an environment in which we can safely experiment.

Ingredients

- VirtualBox[2]
- Ubuntu 10.04 Server image[3]

Solution

We can use virtualization and open-source tools to create a server playground that runs right on our laptop or workstation. We'll use the free VirtualBox software and the Ubuntu Server Linux distribution to build this environment, and we'll then set up the Apache web server with PHP so we can use this environment to test some PHP web projects.

Creating our Virtual Machine

We need to grab two pieces of software: the Ubuntu server operating system, and VirtualBox, an open-source virtualization program. Virtualbox lets us create virtual workstations or servers that run on top of our operating system, giving us a sandbox that we can play in without modifying our actual operating system.

First, we need to visit the Ubuntu download page [4] and grab the 32-bit *server* version of Ubuntu 10.04 LTS instead of the most recent release. LTS stands for "Long-term Support" which means that we can get updates for a much longer period without having to do a complete OS upgrade. The LTS

2. http://www.virtualbox.org/
3. http://www.ubuntu.com/download/ubuntu/download
4. http://www.ubuntu.com/download/server/download

releases don't always have the most up-to-date features, but they're perfect for servers.

While that's downloading, we can go to the VirtualBox web page[5] and download the latest edition of VirtualBox. You'll want to get the one for your platform. Once it's downloaded, install it using the defaults and then launch the VirtualBox program.

Once VirtualBox is running, we need to click the New button to bring up the Virtual Machine Wizard. We need to give our virtual machine a name, such as "My Web Server". Then we'll choose "Linux"for the operating system, and "Ubuntu" for the version. We also need to decide how much memory and disk space to give to the virtual machine, and we can use the defaults which are pretty sensible. We'll end up with a virtual machine that has 512 MB of memory and an 8 GB hard disk.

With our virtual machine created, we can click on the Settings button to configure additional options. We need to change our network type from NAT to Bridged, as shown in Figure 29, *Setting The Network Type to Bridged*, on page 175, so that we can access our servers from our host machine.

Now we can click on the Start button to fire up the new virtual machine. VirtualBox detects that we are running it for the first time and will walk us through the steps to get the Ubuntu operating system running. We'll need to choose media for our operating system, and we can use a CD, but we can also use the ISO image we downloaded. Once we select our installation media, the virtual server starts and the installation of Ubuntu is underway.

For our purposes, we can accept all of the default settings in the Ubuntu installation process. You'll be asked for a hostname, and you can enter whatever you like, but the default will work just fine. You'll also be asked about disk partitioning, and you should accept the defaults, and answer yes whenever you're prompted to write changes to disk. Since this is a virtual machine, you're not going to erase data on your computer's actual hard drive.

Towards the end of the process, we'll be asked to create a user account. This is the user we'll use to log in to our server and do our web server configuration, so let's call it "webdev". We can use that value for both the full name and the username. We'll also need a password, which you can create on your own. Just don't forget it!

5. http://www.virtualbox.org/

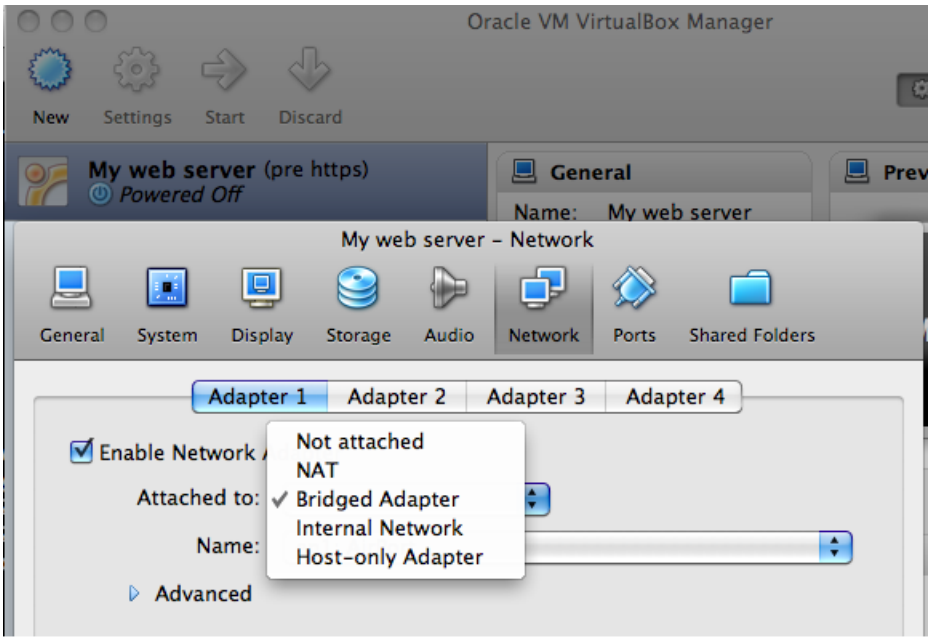


Figure 29—Setting The Network Type to Bridged

When asked if you'd like to install any predefined software, simply choose Continue. We'll install things ourselves at the end of the process.

When the installation finally ends, the virtual machine will restart and we'll be prompted to log in with the username and password we created. Let's do that and get our web server running.

Configuring Apache and PHP

Thanks to Ubuntu's package manager, we can quickly get the Apache web server running with PHP by logging into our server and typing

```
$ sudo apt-get install apache2 libapache2-mod-php5
$ sudo service apache2 restart
```

The first command installs the Apache web server, the PHP5 programming language, and sets up Apache to serve PHP pages. The second reloads Apache's configuration files to ensure the new PHP settings are enabled. Now let's set up our VPS so we can copy files into our web server's directory.

Getting Files to Our Virtual Server

In order to really work with our virtual server, we need to set up services so we can copy our files there.

Apache is serving all of the web files out of the /var/www folder, and the only user who can put files into that folder is the root user. Let's change that by taking ownership of that folder and all its contents with this command:

```
$ sudo chown -R webdev:webdev /var/www
```

Now, let's set up OpenSSH so we can use an SFTP client to copy files, just like we would if we were using a hosting company.

```
$ sudo apt-get install openssh-server
```

And now we can log in using any SFTP client. We just use the IP address of our virtual machine, which we can find by typing

```
$ ifconfig eth0
```

Our IP address is the one that looks like this:

```
inet addr: 192.168.1.100
```

We can now use an SFTP client to connect to that address with the username and password we set when we built the virtual machine. From a Windows machine, we could use FileZilla, and from a Mac we can use Cyberduck or even the command line, using the scp command to transfer a file. For example if we had a simple HTML file in our home directory, we could transfer it to our server like this:

```
scp index.html webdev@192.168.1.100:/var/www/index.html
```

We specify the source filename, followed by the destination path, which is the username we want to connect with, followed by the @ sign and the IP address of the server, a colon, and then by the full path where we'll place the file.

With our virtual machine in place, we can now start using it as a testing playground. When it comes time to deploy our code to our production environment, we'll have had enough practice.

Further Exploration

Virtual machines give us a playground where we can test, experiment, and break things, but we can do more than that. VirtualBox's "snapshot" feature lets us create restore points that we can revert to if we goof something up. This is perfect for those times when we're interested in playing with a new

piece of technology. In addition, we can create “appliances”, or specific virtual machines with preloaded packages. We could create a PHP appliance, which has PHP, MySQL, and Apache already configured, and then share that virtual machine with others so they can get started quickly.

Virtual machines are extremely useful for deploying actual applications. For example, we can snapshots in production to restore our machine after a failed upgrade or a security exploit, and we can clone virtual machines to scale things out. Closed-source products like VMWare provide enterprise-level solutions for hosting multiple virtual machines on a single physical server[6]. VMWare even provides some tools for taking a physical machine and converting it to a virtual one.[7]

Also See

- Recipe 38, *Securing Apache With SSL And HTTPS*, on page 178

6. http://www.vmware.com/virtualization/
7. http://www.vmware.com/products/converter/

Recipe 38

Securing Apache With SSL And HTTPS

Problem

When our applications and web sites deal with people's information, we owe it to them to safeguard it. We want to make sure our servers and databases safely store that information, but we also need to protect that data during its trips from their computer to our servers and back. We need to configure our web server so it connects to web browsers using SSL.

Ingredients

- A virtual machine running Ubuntu for testing
- The Apache web server with SSL support

Solution

In order to set up a secure web server, we need to set up SSL certificates. Production web sites use signed SSL certificates which are verified by a third party authority. This verification gives customers a sense of security.

Signed SSL certificates cost money, and we don't want to pay for certificates for our development environments. For testing purposes, we can create "self-signed" certificates, ones we verify ourselves.

We'll use the virtual machine we created in Recipe 37, *Setting up a Virtual Machine*, on page 173 so we can get some practice. That way, when we have to set up our production machine, we'll know exactly what to do. We'll do all of the commands in this recipe from our virtual machine's console, *not* on your local machine.

Creating a self-signed certificate for development

The process for getting an SSL certificate is the same whether we're getting a verified one or a self-signed one. We start by creating a "certificate request". This request usually gets sent to a certificate authority along with a payment, and they then send back a verified SSL certificate we can install on our

server. In our case, we'll be acting as both the certificate requester and the certificate authority.

To create the request, we fire up our virtual machine, log into the console, and type:

```
$ openssl req -new -out awesomeco.csr
```

This creates both a certificate request and a private signing key that requires a passphrase.

We'll need to provide a passphrase for this new key, and we'll be asked for our company name and other details. You'll want to fill these out with real data, especially if you plan to use this to request a key from a certificate authority!

The key we created requires that we enter a passphrase every time we use it. If we request a certificate with this key, we'll have to enter that passphrase every time we restart our web server. This is secure, but pretty inconvenient . It's also not very manageable in a production environment. Let's create a key we can use that doesn't require a password.

```
$ sudo openssl rsa -in privkey.pem -out awesomeco.key
```

Now that we have our request, we can sign it ourselves by passing both our request and our key.

```
$ openssl x509 -req -days 364 -in awesomeco.csr \
-signkey awesomeco.key -out awesomeco.crt
```

The certificate we created will be good for one year.

Finally, we need to copy our certificate and our keyfile to the appropriate locations.

```
$sudo cp awesomeco.key /etc/ssl/private
$ sudo cp awesomeco.crt /etc/ssl/certs
```

Now let's modify the default Apache web site to use SSL.

Configuring Apache for SSL support

We need to enable the apache module for SSL support on our server. To do that, we can either manually edit the list of installed modules, or we can type

```
$ sudo a2enmod ssl
```

which will do the modification for us.

Now we need to tell Apache to serve web pages using SSL.

Let's create a separate configuration file for our SSL site. Create the file /etc/apache2/sites-available/ssl_example and add the following configuration to the file:

```
  <VirtualHost *:443>
  ServerAdmin webmaster@localhost
  DocumentRoot /var/www
  <Directory /var/www/>
    Options FollowSymLinks
    AllowOverride None
  </Directory>
  SSLEngine on
  SSLOptions +StrictRequire
  SSLCertificateFile /etc/ssl/certs/server.crt
SSLCertificateKeyFile /etc/ssl/private/server.key
</VirtualHost>
```

We're creating a new virtual host on port 443, listening on all addresses. The document root specifies where our web pages are, and the directory section sets up some basic permissions.

The last few lines set up the actual SSL connections, turning on SSL support, ensuring it's strictly enforced, and that it knows where our self-signed certificate and key are located.

With this new configuration file saved, we need to enable it and tell Apache to reload its configuration.

```
$ sudo a2ensite ssl_example
$ sudo /etc/init.d/apache2 restart
```

Now we can visit our web site's URL over SSL. We'll get some warnings from the browser though because a certificate we created by ourselves isn't considered safe for the average user. And that makes sense. If anyone could create a certificate that was automatically trusted by every browser, it really would not be that secure. We need to get a third party involved to get a trusted certificate. That's where a Certificate Provider comes in.

Working with a certificate provider

We don't want our users thinking we're trying to steal their credit card information or do other evil things with their data, so we need to get a trusted certificate. To do that, we generate a certificate request and a key in the same fashion we did for our self-signed certificate. We then send the certificate request to the certificate authority along with our payment, and they'll send back a certificate we can install along with other instructions.

Some certificate authorities do more than just take your money in exchange for a certificate that removes the error message. Some will actually verify that your entity is a legitimate business. When your users review the details of the certificate in their browser, they can see this information, which adds an additional layer of trust. It also adds extra costs for you, but depending on your industry, it may be worth it.

There are many certificate authorities out there. Thawte[8] and Verisign[9] are well-known and trusted certificate authorities, but you'll need to research some on your own to find ones that meet your needs. If you're working with a hosting provider, you can often work with them to get a signed certificate for your site.

Further Exploration

There are actually several types of SSL certificates we can use. We can get certificates that cover a single server, or we can get a "wildcard" certificate that we could apply to all servers within our domain. Wildcard certificates are much more expensive than single server certificates.

Finally, *Server Name Indication*, or SNI certificates are a much cheaper option, but they only work with the most modern browsers and operating systems. SNI certificates are great for internal organizations where you have control over the browsers your clients use, but you'll want to rely on more traditional host or IP-based certificates for the general public.

Also See

- Recipe 37, *Setting up a Virtual Machine*, on page 173

8. http://www.thawte.com
9. https://www.verisign.com/

Recipe 39

Managing Configuration Files on the Server

Coming soon...

Recipe 40

Securing Your Content

Coming soon....

Recipe 41

Rewriting URLs to Preserve Links

Coming soon...

Recipe 42

Automate Static Site Deployment with Guard and Rake

Coming soon...

APPENDIX 1

Installing Ruby

Several of the recipes in this book use the Ruby programming language or its interpreter. Ruby is a powerful cross-platform interpreted language that's most well known for the Ruby on Rails web development framework. Many Ruby developers work on the web, and they've used Ruby to create some amazing tools like Cucumber[1] and SassOS X[2] to speed up the web development process. To use these tools, we'll need the Ruby interpreter and its package management system, RubyGems, installed. In this appendix, we cover how to do just that on Windows, OS X, and Ubuntu.

A1.1 Windows

Windows installation is a two-part process. First, download the Ruby Installer for Windows. [3] Grab the one for Ruby 1.9.2. The installation will give you the option to add Ruby executables to your PATH, which you should do. This way you'll be able to use Ruby and other libraries you install from your Command Prompt no matter what folder you're currently in.

Once that's done, download the Development Kit, which you can find on the same download page. While we won't be writing our own Ruby programs in these recipes, some of the components we want to install need to be compiled, as they're actually written in the C programming language. The Development Kit contains these compilers.

The Development Kit is a self-extracting archive, which you should extract to `c:\ruby\devkit`. Then, open a command prompt and run the following commands:

1. http://cukes.info/
2. http://sass-lang.com
3. http://rubyinstaller.org/downloads/

```
c:\> cd \devkit
c:\> ruby dk.rb init
c:\> ruby dk.rb install
```

To test out your setup, install the Cucumber gem by opening your Command Prompt and typing

```
$ gem install cucumber
```

That's all there is to it. Now you can use libraries like Cucumber and SASS in your development projects.

A1.2 Mac OS X and Linux with RVM

OS X and many Linux distributions have pre-built Ruby interpreters available; OS X even has it installed by default. We'll use RVM, or Ruby Version Manager, to install and manage our Ruby installations.[4] While RVM works exactly the same on every platform it supports, each platform has its own unique setup process. We'll cover setting up RVM for OS X and Ubuntu.

Setting up RVM on OS X

To use RVM on OS X, you need to install XCode.[5] We won't be using XCode in this book, but it's the easiest way to get the C compilers we need. You can find XCode on your OS X installation DVD, or through the Mac App Store. You'll also need to install Git for OS X, which we discuss in Recipe 30, *Managing Files Using Git*, on page 142, as we'll need that to fetch RVM.[6]

Next, execute this command from your Terminal to install RVM:

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

This fetches RVM and installs it to your home directory.

Next, run this command so that RVM and its files are available every time you start a new Terminal session:

```
$ echo '[[ -s "$HOME/.rvm/scripts/rvm" ]] && \
source  "$HOME/.rvm/scripts/rvm"' >> ~/.bashrc
```

Finally, close and restart your Terminal session to ensure that RVM is now available, and then continue on to *Installing Ruby with RVM*, on page 189.

4. http://rvm.beginrescueend.com
5. http://developer.apple.com/xcode/
6. http://code.google.com/p/git-osx-installer/

Setting up RVM on Ubuntu

RVM has several dependencies for Ubuntu. This command will work nicely, as it installs the compilers, prerequisites, and Git, which we discuss in Recipe 30, *Managing Files Using Git*, on page 142:

```
$ sudo apt-get install build-essential bison openssl \
libreadline6 libreadline6-dev curl git-core zlib1g \
zlib1g-dev libssl-dev libyaml-dev libsqlite3-0 \
libsqlite3-dev sqlite3 libxml2-dev libxslt-dev autoconf
```

When those libraries are finished installing, execute this command from your Terminal to install RVM:

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

This fetches RVM and installs it to your home directory.

Next, run this command so that RVM and its files are available every time you start a new Terminal session:

```
$ echo '[[ -s "$HOME/.rvm/scripts/rvm" ]] && \
source "$HOME/.rvm/scripts/rvm"' >> ~/.bashrc
```

Finally, close and restart your Terminal session to ensure that RVM is now available.

Installing Ruby with RVM

With RVM installed, you can now install Ruby 1.9.2 with this command:

```
$ rvm install 1.9.2
```

Then, to use this version of Ruby, you type:

```
$ rvm use 1.9.2
```

For the purposes of this book, you may want to set this to the default version of Ruby

```
$ rvm --default use 1.9.2
```

Now, test things out by installing the Cucumber library which we use in Recipe 34, *Cucumber-driven Selenium Testing*, on page 164. From the Terminal, type this command:

```
$ gem install cucumber
```

That's it! Ruby and its prerequisites are now installed, and you can now use tools like Cucumber and Sass in your web development projects.

Also See

- Recipe 34, *Cucumber-driven Selenium Testing*, on page 164
- Recipe 27, *Creating A Simple Blog with Jekyll*, on page 133
- Recipe 30, *Managing Files Using Git*, on page 142
- Recipe 29, *Cleaner JavaScript with CoffeeScript*, on page 135
- Recipe 28, *Building Modular Stylesheets With Sass*, on page 134
- Recipe 42, *Automate Static Site Deployment with Guard and Rake*, on page 185

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
http://pragprog.com/titles/wbdev
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *http://pragprog.com/titles/wbdev*

Contact Us

Online Orders:	*http://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://pragprog.com/write-for-us*
Or Call:	+1 800-699-7764