Chapter 1

Best Practices

A firm foundation is vitally important for any application. Before you write a single line of code, you need to spec out the app's architecture. What features will your app have, and how will these be implemented? More importantly, how will these features work with one another; in other words, what is the app's ecosystem?

Answering these questions involves a combination of research, prototyping, and a firm grounding in best practices. While I can't help you research or prototype the specific components of your app, I can pass on the wisdom I've gained on best practices.

This chapter covers the fundamental engineering concept of loose coupling, then explains one method of achieving it: JavaScript MVCs and templating engines. Next you discover a variety of development tools, such as Weinre, version control and CSS preprocessing. Finally, you learn how to set up a project in Grunt to automate tasks such as concatenation and minification. Using Grunt, you establish a test driven development pattern that runs your app through a suite of tests whenever a file is modified.

Loose Coupling

If you take only one thing from this book, I hope that it's to avoid tight coupling in your app. Tight coupling is an old engineering term that refers to separate components being too interdependent. For instance, say that you buy a TV with a built-in BluRay player. But what happens if the TV breaks? The BluRay player may still work perfectly, but is rendered useless by the broken TV. From an engineering perspective, it's better to avoid tight coupling and get a separate, external BluRay player.

This pattern also applies to software development. Basically, you should design your app with isolated modules that each handle a single task. By decoupling these tasks, you minimize any dependencies between different modules. That way each module remains as "stupid" as possible, able to focus on an individual task without having to concern itself with the other code in your app.

But it can be challenging to determine what exactly should be grouped into a module. Unfortunately, there's no one-size-fits-all solution—too few modules leads to tight coupling, too many leads to unnecessary abstraction. The best approach is in the middle: designing your app to use a reasonable number of modules with high cohesion. Cohesive modules group highly related pieces of functionality to handle a single, well-defined task.

Problems with Tight Coupling

Examples of tight coupling are all around us. If you're like me, your phone has replaced your music player, video game console, and even your flashlight. There's a certain convenience to having all these features integrated in one simple device. In this case, tight coupling makes sense. But that means that when one thing breaks, a chain of failures can occur—listen to enough music on your phone and suddenly your flashlight is out of batteries.

In software development, tight coupling isn't necessarily a bad thing—poorly designed coupling is more the problem. Apps almost always have a certain number of dependencies; the trick is to avoid any unnecessary coupling between separate tasks. If you don't take efforts to isolate different modules, you'll wind up with a brittle app that can be completely broken by even small bugs. Sure, you want to be doing everything you can to avoid bugs in the first place, but you aren't doing yourself any favors if every bug takes down your entire app.

Furthermore, debugging a tightly coupled app is extremely difficult. When everything is broken, it's almost impossible to track down exactly where the bug happened in the first place. This issue results from what is commonly referred to as *spaghetti code*. Much like pieces of spaghetti, the lines of code are all interwoven and very difficult to pull apart.

Advantages of Loose Coupling

Even if you rarely encounter bugs, loose coupling still has some pronounced advantages. In fact, one of the main reasons to build loosely coupled apps boils down to another cornerstone of classic engineering: interchangeable parts. Over the course of production, it's often necessary to rebuild portions of your app. Has Google started charging for their translation API? Better patch something else in. Has a component scaled poorly and begun to run slowly under load? Better rebuild it.

If your app is too tightly coupled, a change in one module can cause a ripple effect, where you have to accommodate the change in all the dependent modules. Loose coupling avoids that extra development time, and keeps code changes contained in individual modules.

Furthermore, loose coupling encourages easier collaboration with other developers. When all the individual components are isolated, working on different pieces in parallel becomes much easier. Each developer can work on his or her task without fear of breaking something someone else is working on.

Finally, loose coupling makes testing easier. When each piece of your app handles a separate, specific task, you can easily set up unit tests to ensure these tasks are executing correctly under any number of circumstances. You'll find out more about unit testing later this chapter.

In an ideal world, you'd never have to refactor your codebase. But there will always be unforeseen issues, and even if you could account for every possible scenario, why would you bother? Trying to preemptively solve problems can lead to "premature optimization" and is sure to slow down development. In an agile world, you should only concern yourself with the problems you face right now, and deal with future issues in the future. Loose coupling streamlines the agile development process, and allows your codebase to evolve naturally as conditions change.

JavaScript MVCs and Templates

Continuing the theme of loose coupling, another design pattern emphasized in this book is the use of JavaScript model-view-controllers (MVCs) and templates. These provide a structure to decouple various aspects of your application.

MVCs

MVC is a design pattern that encourages loose coupling. It separates the data that drives an application from the visual interface that displays that data. Using an MVC framework allows you to change the front-end styling of an application, without having to modify the underlying data. That's because MVCs separate concerns into three related components: the model, view, and controller, as illustrated in Figure 1-1.

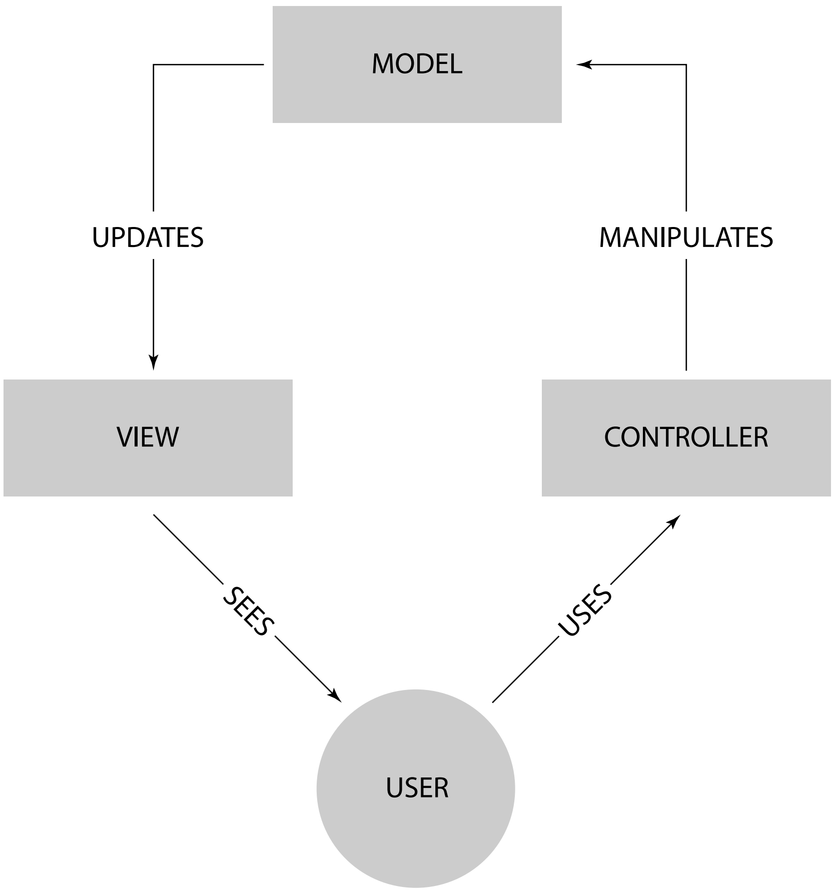


Figure 1-1 This diagram shows the relationship between the three components of an MVC.

Model

The model component of an MVC is the data that drives your application. You can think of the model layer as the domain logic of your application—it is all the data that your app will handle. On a simple brochure site, the model layer might only contain a few objects representing the content of the site: the text, image paths, and so forth. More complex apps, on the other hand, often use a large number of models to represent every individual piece of data the app needs. One example might be a model to represent an individual user (username, password, and so on).

The data in models is often saved in a database, or another data store like local storage. That way the data can persist across multiple sessions.

View

The view's sole duty is to present the user interface. In a web app, the end result of the view is markup, since it displays the content on the screen. But the view is more complicated than simple markup; it must reprocess the data from the model into a format that can be rendered as markup.

For example, dates may be stored as UNIX timestamps in the model, but you wouldn't want to display these to the user. The view takes these system time values, converts them to a readable value such as "August 4" or "Posted 5 minutes ago." Next, the view takes this cleaned-up timestamp and displays it as markup—for instance:

```
<div class="post-date">Posted 5 minutes ago</div>
```

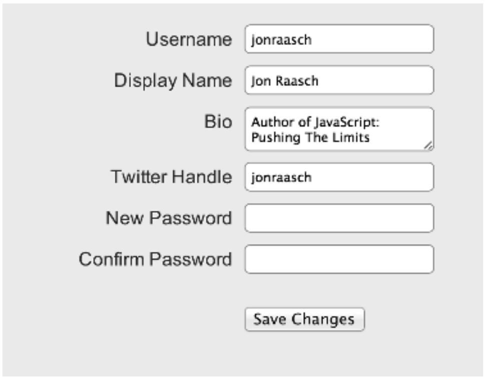
Combine this with all the other markup you need for the page, and you've got the view.

Controller

Last but not least, the controller relays data back and forth between the model and the view. It is the component that takes in user input to modify the model and is ultimately responsible for updating the view with modified data. One example of the controller in action is a form handler. After the user posts a form, the data he or she submitted is processed by the controller, which in turn changes the appropriate data in the model. Then the change in the model is relayed back through the controller to update the view.

Putting It All Together

Alone, each of these components cannot do very much, but when the three separate pieces come together, they build an application. For example, apps commonly have a form that allows users to manage their user data: username, password, and so on. This data is stored in the model, which is typically a database. The view then takes this data from the model and uses it to render the form with all the default fields filled out (old username, twitter handle, and so on), as shown in Figure 1-2.



Username	jonraasch
Display Name	Jon Raasch
Bio	Author of JavaScript: Pushing The Limits
Twitter Handle	jonraasch
New Password	
Confirm Password	

Figure 1-2 The view has displayed data from the model in a form.

The users then interact with this form, changing whichever fields they want. Once submitted, the controller handles the users' request to update the form and then modifies and optionally persists the data model. Then the cycle can repeat itself, as demonstrated earlier in Figure 1-1.

You can find more about the MVC design pattern in Chapter 3, where I cover Backbone.js. That chapter also covers when it's appropriate to use an MVC framework: they aren't for every project.

Templates

JavaScript templates are part of the "V" in MVC: they're tools to help build the view. You don't need to use templates to follow the MVC design pattern, and you can also use templates without an MVC. In fact, I encourage you to use them regardless of whether you're using an MVC framework.

You may already be familiar with using templates in PHP or another back-end language. Then you're in luck because JavaScript templates work essentially the same.

How to Use Templates

Here's an example of a basic JavaScript template file:

```
<hgroup>
  <h1><%=title %></h1>
  <h2><%=subtitle %></h2>
</hgroup>

<section class="content">
  <p>
  <%=content %>
  </p>
</section>
```

The template is basically HTML markup with some variables enclosed in `<% %>`. These variables can be passed to the template to render different content in different situations. Don't get caught up on the syntax here—different template frameworks use different syntaxes, and most allow you to change it as needed.

For an in-depth discussion of JavaScript templating engines, turn to Chapter 4.

Why Use Templates

Templates make it easy to represent models in the view. Their engines provide syntax for iterating though collections, say an array of products, and easy access to object properties that you want the view to render like a product's price.

Without templates, your JavaScript can get quite messy, with concatenated markup strings interspersed throughout scripting tasks. To make matters worse, if the markup ever has to change for styling or semantic reasons, you have to track down these changes throughout your JavaScript. Therefore, you should use templates anywhere you're touching the markup in your JavaScript, even if it's just one tag.

Development Tools

Part of being a good developer is using the best tools for the job. These tools speed up your development process, help you squash bugs, and improve the performance of your app. In this section you first learn about the WebKit Developer Tools. You're probably at least somewhat familiar with these tools, but we'll go more in depth and explore some of the more advanced features. Next you discover Weinre, a remote console tool you can use to get the WebKit Developer Tools on any platform, such as a mobile device or non-WebKit browser. Finally, I stress the importance of using version control and CSS preprocessing.

WebKit Developer Tools

Of all the developer toolkits available, my personal favorite is the WebKit Developer Tools. These are baked into WebKit-based browsers such as Chrome and Safari. These tools make it easier to debug issues in your JavaScript, track performance, and more.

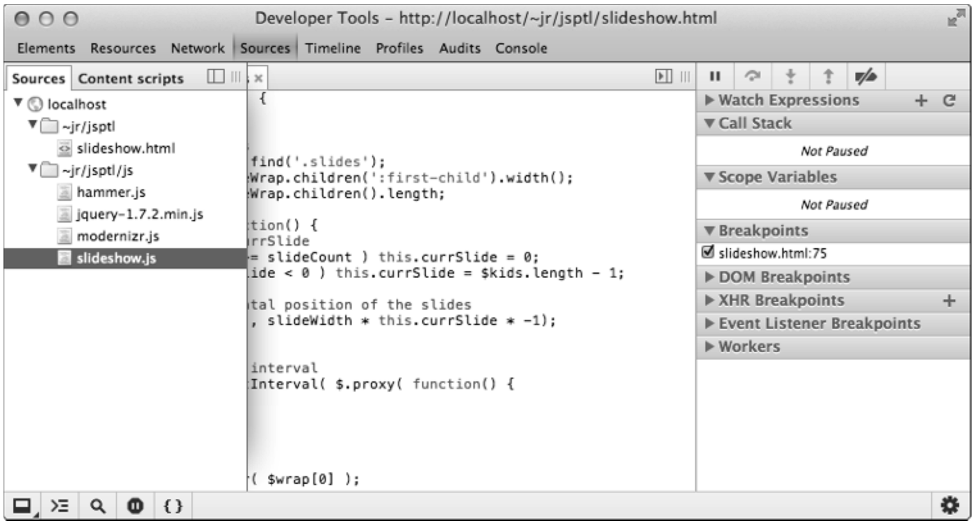
The Chrome Developer Tools are installed by default in Chrome. You can access them through the Chrome menu or by right-clicking any element on the page and selecting Inspect Element.

If you prefer to develop in Firefox, you can try Firebug or the baked-in Firefox Developer Tools.

Breakpoints

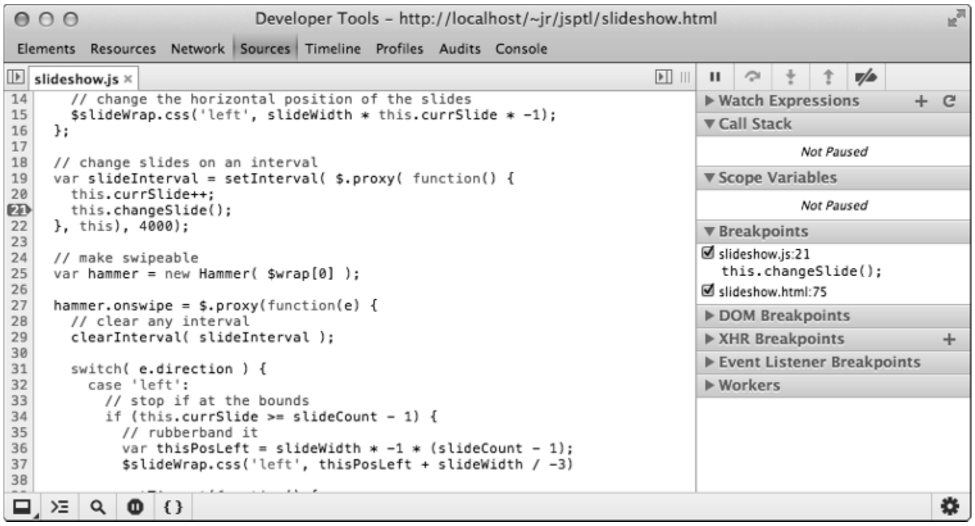
One extremely useful tool for JavaScript development is the Sources panel in the WebKit Developer Tools. With this tool you can set up arbitrary breakpoints in your scripts. At these points, scripting pauses and allows you to gather information about what is going on in the script. Open up the Sources panel and select a script from the flyout on the left, as shown in Figure 1-3.

Next set up a breakpoint by clicking the left margin at whatever line number you want to analyze. As shown in Figure 1-4, a blue flag displays next to the breakpoint.



```
{



find('.slides');
Wrap.children(':first-child').width();
Wrap.children().length;

tion() {
rrSlide
= slideCount ) this.currSlide = 0;
ide < 0 ) this.currSlide = $kids.length - 1;

tal position of the slides
, slideWidth * this.currSlide * -1);


interval
Interval( $.proxy( function() {





( $wrap[0] );
```

Figure 1-3 First select the script you want to analyze.

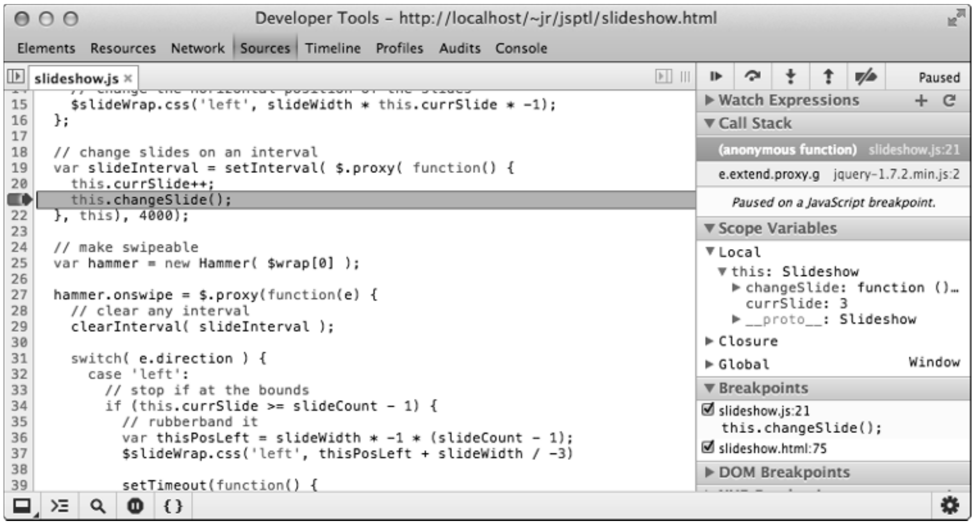


```
14     // change the horizontal position of the slides
15     $slideWrap.css('left', slideWidth * this.currSlide * -1);
16   };
17
18   // change slides on an interval
19   var slideInterval = setInterval( $.proxy( function() {
20     this.currSlide++;
21     this.changeSlide();
22   }, this), 4000);
23
24   // make swipeable
25   var hammer = new Hammer( $wrap[0] );
26
27   hammer.onswipe = $.proxy(function(e) {
28     // clear any interval
29     clearInterval( slideInterval );
30
31     switch( e.direction ) {
32       case 'left':
33         // stop if at the bounds
34         if (this.currSlide >= slideCount - 1) {
35           // rubberband it
36           var thisPosLeft = slideWidth * -1 * (slideCount - 1);
37           $slideWrap.css('left', thisPosLeft + slideWidth / -3)
38
```

Figure 1-4 A breakpoint has been set up on line 21.

Finally, let your script run. Once it gets to the breakpoint, it pauses all scripting on the page the text, and `Paused in debugger` displays across the top of the browser window. At this point you can gather any information you need. The current variables are output in the right column in the Scope Variables section. You can also click any object in the source to see its value as shown in Figure 1-5.

Once you are done analyzing the data, simply click the play button to allow the script to continue until it hits the next breakpoint.


Developer Tools – http://localhost/~jr/jsptl/slideshow.html
Elements Resources Network Sources Timeline Profiles Audits Console
slideshow.js
Paused
Watch Expressions
Call Stack
Scope Variables
Breakpoints
DOM Breakpoints

```
  $slideWrap.css('left', slideWidth * this.currSlide * -1);
};

// change slides on an interval
var slideInterval = setInterval( $.proxy( function() {
  this.currSlide++;
  this.changeSlide();
}, this), 4000);

// make swipeable
var hammer = new Hammer( $wrap[0] );

hammer.onswipe = $.proxy(function(e) {
  // clear any interval
  clearInterval( slideInterval );

  switch( e.direction ) {
    case 'left':
      // stop if at the bounds
      if (this.currSlide >= slideCount - 1) {
        // rubberband it
        var thisPosLeft = slideWidth * -1 * (slideCount - 1);
        $slideWrap.css('left', thisPosLeft + slideWidth / -3)

        setTimeout(function() {
```

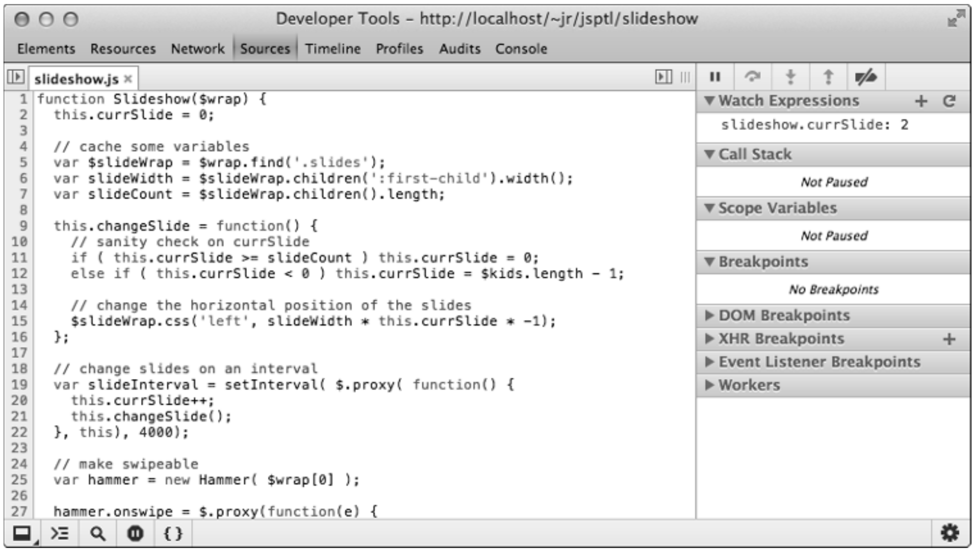
```
(anonymous function)  slideshow.js:21
e.extend.proxy.g  jquery-1.7.2.min.js:2
Paused on a JavaScript breakpoint.

▼Local
  ▼this: Slideshow
    ▶changeSlide: function ()…
     currSlide: 3
    ▶__proto__: Slideshow
▶Closure
▶Global                    Window

☑ slideshow.js:21
  this.changeSlide();
☑ slideshow.html:75
```

Figure 1-5 When the script reaches the breakpoint, you will be able to analyze the current value of any object.

Setting up breakpoints is absolutely invaluable for debugging since it allows you to pause the script whenever you want. It is especially useful on scripts that use timing functions or other features that can change before you have a chance to analyze them.

Watch Expressions

In addition to breakpoints, you can also use watch expressions from the Sources panel. Simply add an expression to the Watch Expressions column as shown in Figure 1-6.


Developer Tools – http://localhost/~jr/jsptl/slideshow
Elements Resources Network Sources Timeline Profiles Audits Console
slideshow.js
Watch Expressions
Call Stack
Scope Variables
Breakpoints
DOM Breakpoints
XHR Breakpoints
Event Listener Breakpoints
Workers

```
function Slideshow($wrap) {
  this.currSlide = 0;

  // cache some variables
  var $slideWrap = $wrap.find('.slides');
  var slideWidth = $slideWrap.children(':first-child').width();
  var slideCount = $slideWrap.children().length;

  this.changeSlide = function() {
    // sanity check on currSlide
    if ( this.currSlide >= slideCount ) this.currSlide = 0;
    else if ( this.currSlide < 0 ) this.currSlide = $kids.length - 1;

    // change the horizontal position of the slides
    $slideWrap.css('left', slideWidth * this.currSlide * -1);
  };

  // change slides on an interval
  var slideInterval = setInterval( $.proxy( function() {
    this.currSlide++;
    this.changeSlide();
  }, this), 4000);

  // make swipeable
  var hammer = new Hammer( $wrap[0] );

  hammer.onswipe = $.proxy(function(e) {
```

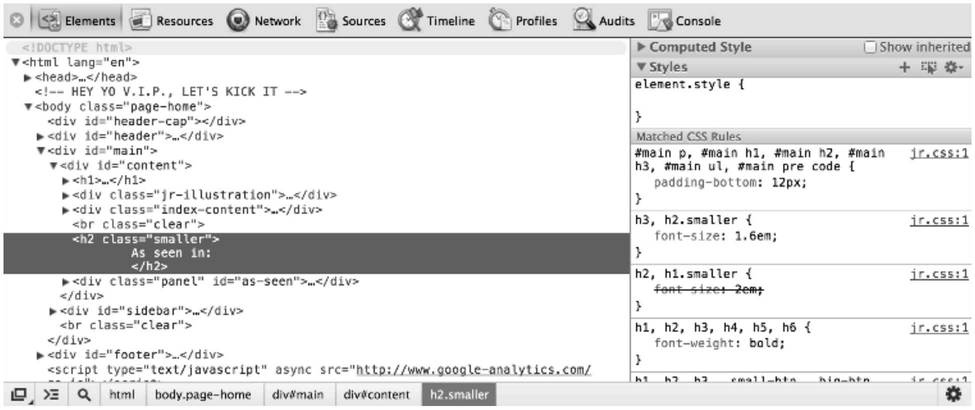
```
slideshow.currSlide: 2
Not Paused
Not Paused
No Breakpoints
```

Figure 1-6 Watching the value of an expression; in this case the sources panel tracks the value of `slideshow.currSlide`.

You can enter anything you want into the watch expression, a particular object or a custom expression. When working with watchpoints, click the reload icon to refresh the value, or use the pause button to stop scripting at any point.

DOM Inspector

The DOM Inspector allows you to click on any item on the page and bring up that element's location in the markup. It also shows any CSS styling that has been applied to that element, as shown in Figure 1-7.



```
<!DOCTYPE html>
▼<html lang="en">
  ▶<head>…</head>
    <!-- HEY YO V.I.P., LET'S KICK IT -->
  ▼<body class="page-home">
      <div id="header-cap"></div>
    ▶<div id="header">…</div>
    ▼<div id="main">
      ▼<div id="content">
        ▶<h1>…</h1>
        ▶<div class="jr-illustration">…</div>
        ▶<div class="index-content">…</div>
          <br class="clear">
          <h2 class="smaller">
                  As seen in:
                  </h2>
        ▶<div class="panel" id="as-seen">…</div>
        </div>
      ▶<div id="sidebar">…</div>
        <br class="clear">
      </div>
    ▶<div id="footer">…</div>
      <script type="text/javascript" async src="http://www.google-analytics.com/
```

```
element.style {

}
```

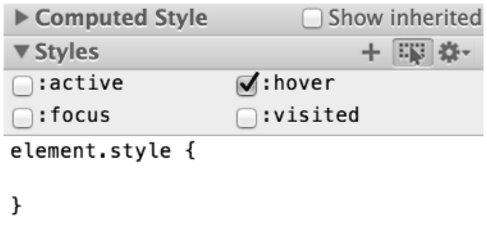
```
#main p, #main h1, #main h2, #main      jr.css:1
h3, #main ul, #main pre code {
   padding-bottom: 12px;
}
h3, h2.smaller {                        jr.css:1
   font-size: 1.6em;
}
h2, h1.smaller {                        jr.css:1
   ~~font-size: 2em;~~
}
h1, h2, h3, h4, h5, h6 {                jr.css:1
   font-weight: bold;
}
```

Figure 1-7 The DOM Inspector shows the element's location in the DOM and any styling.

To access the DOM Inspector, simply right-click any element on the page and click Inspect Element.

Additionally, you can use the DOM Inspector to adjust styling on the fly, changing any style rules that have been applied and adding new ones. The DOM Inspector can be instrumental both in tracking down pesky styling issues and testing new styling directly in the browser.

In more recent versions of the WebKit Developer Tools, you can also activate pseudo-classes such as `:hover` and `:focus`. Select the element you want to study, and then expand the pseudo-class menu in the right column's Styles area (by clicking the dotted rectangle icon shown in Figure 1-8).



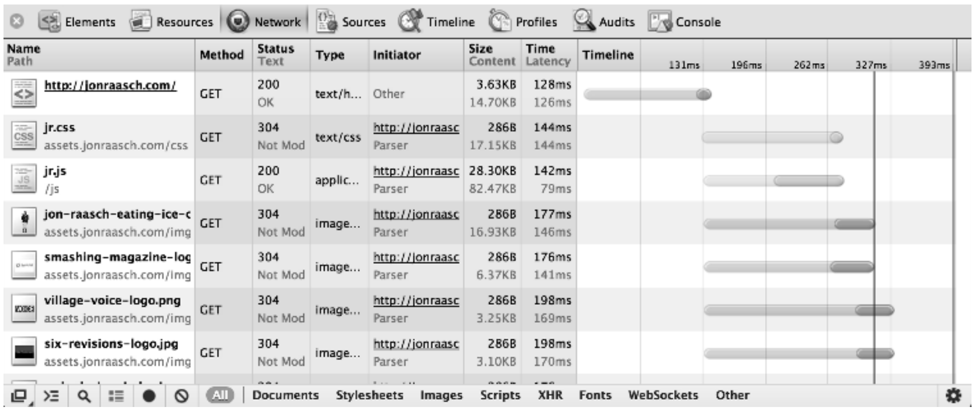
```
element.style {

}
```

Figure 1-8 You can also modify pseudo-classes with the DOM Inspector.

If you're using the DOM Inspector to style content as you go, be sure to save your work in a stylesheet often. Otherwise, you'll lose all your work if you accidentally reload the page or your browser crashes.

Network Panel

The Network panel allows you to track the time it takes each requested resource to be delivered to the client. It gives you a breakdown of server latency, download speed, and instantiation time for your various resources, as you can see in Figure 1-9.



Name / Path	Method	Status / Text	Type	Initiator	Size / Content	Time / Latency	Timeline
http://jonraasch.com/	GET	200 OK	text/h...	Other	3.63KB 14.70KB	128ms 126ms	
jr.css assets.jonraasch.com/css	GET	304 Not Mod	text/css	http://jonraasc Parser	286B 17.15KB	144ms 144ms	
jr.js /js	GET	200 OK	applic...	http://jonraasc Parser	28.30KB 82.47KB	142ms 79ms	
jon-raasch-eating-ice-c assets.jonraasch.com/img	GET	304 Not Mod	image...	http://jonraasc Parser	286B 16.93KB	177ms 146ms	
smashing-magazine-log assets.jonraasch.com/img	GET	304 Not Mod	image...	http://jonraasc Parser	286B 6.37KB	176ms 141ms	
village-voice-logo.png assets.jonraasch.com/img	GET	304 Not Mod	image...	http://jonraasc Parser	286B 3.25KB	198ms 169ms	
six-revisions-logo.jpg assets.jonraasch.com/img	GET	304 Not Mod	image...	http://jonraasc Parser	286B 3.10KB	198ms 170ms	

Figure 1-9 The Network panel shows how long it takes to download various resources.

The Network panel provides information for resources such as HTML, CSS, JavaScript, and images—and even allows you to filter by the type of request. This data is priceless when performance-tuning your app. Problems with slow responses are easy to pinpoint as you have access to the full request and response including the header information that was sent.

When you're developing locally, the network panel isn't useful for determining response times. That's because the browser doesn't have to download any of the resources from an external site. Upload your work to a staging server before testing.

Keyboard Shortcuts

If you use the Developer Tools as much as I do, you'll want to become skilled at using some of the keyboard shortcuts to access them quickly.

- To open or close the Developer Tools in Mac, press Cmd-Option-I, or Ctrl+Shift+I on PC.
- To open the console, press Cmd-Option-J on Mac, or Ctrl+Shift+J on PC.
- To toggle Inspect Element mode, press Cmd-Shift-C on Mac, or Ctrl+Shift+C on PC.

For more shortcuts, visit: `https://developers.google.com/chrome-developer-tools/docs/shortcuts`.

Weinre

The WebKit Developer Tools are great for working on issues in Chrome and Safari. However, often you need access to tools in other browsers, whether you're debugging an issue that affects only a specific browser or testing performance on mobile.

You could install a different set of developer tools in each browser. But these will all work slightly differently, and may be insubstantial in some cases (such as the limited tools in mobile browsers). That's where Weinre comes in. Weinre is a remote console, which allows you to access the WebKit Developer Tools in any browser or platform. It's especially ideal for mobile, since it provides a complete, robust testing suite on any device.

Setting Up Weinre

First, you need to install a Node.js server on your local machine. You can learn more about installing Node from the Node website (`http://nodejs.org/`) or turn to Chapter 6 and follow the instructions there.

Next, install Weinre using NPM. From the command line, type the following:

```
sudo npm -g install weinre
```

After the installation completes, start the Weinre server from the command line:

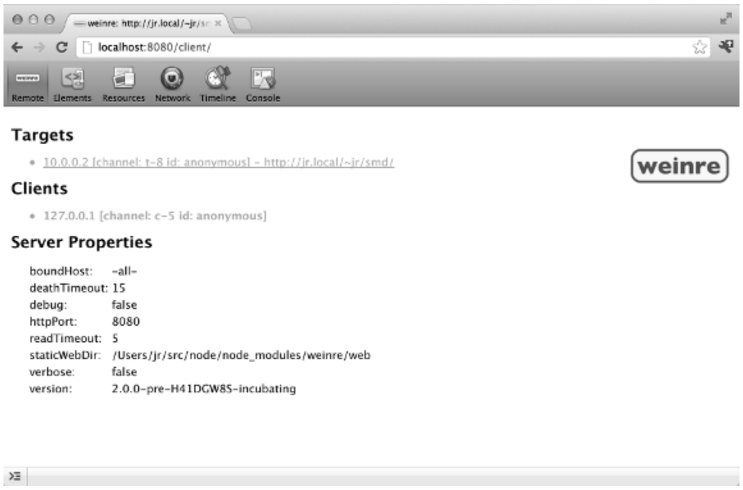
```
weinre --boundHost -all-
```

Finally, you need to add a script tag to your test page, to set it up as a debug target:

```
<script src="http://1.2.3: 8080/target/target-script-min.js"></script>
```

Here, replace `http://1.2.3:8080/` with the location of your local Weinre server (which is running off your desktop machine).

With Weinre set up, you can simply open the page on the mobile device you want to test (or whichever desktop browser you want to test). Then open Chrome on your desktop machine and visit `http://localhost:8080/client`. If all goes well, you should see two green IP addresses, one for the remote target (the mobile device) and one for the remote client (the desktop machine). See Figure 1-10.



Targets

- 10.0.0.2 [channel: t-8 id: anonymous] - http://jr.local/~jr/smd/

Clients

- 127.0.0.1 [channel: c-5 id: anonymous]

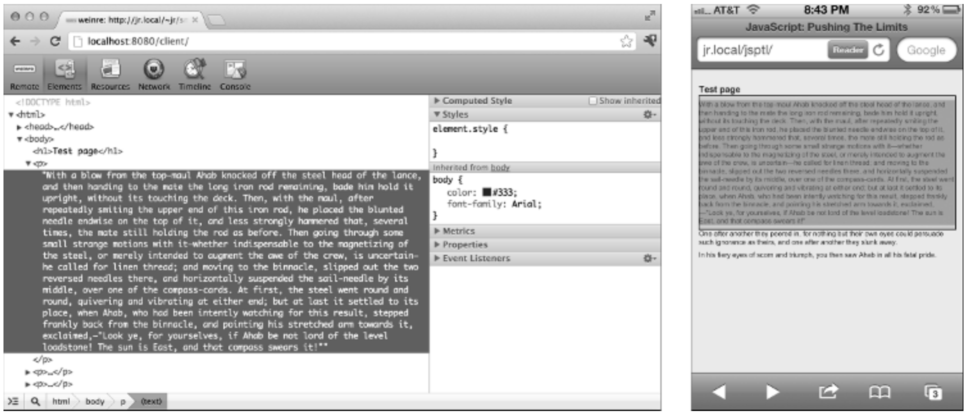
Server Properties

```
boundHost:     -all-
deathTimeout:  15
debug:         false
httpPort:      8080
readTimeout:   5
staticWebDir:  /Users/jr/src/node/node_modules/weinre/web
verbose:       false
version:       2.0.0-pre-H41DGW85-incubating
```

Figure 1-10 Weinre is connected when the target and client IPs are both green.

Using Weinre

After you've installed Weinre, using it is a piece of cake. Simply use the Weinre console in Chrome, and you have access to all the WebKit Developer Tools you read about earlier this chapter. You can use the JavaScript console, DOM Inspector, Sources panel, Network panel, and more, as shown in Figure 1-11.



```
<!DOCTYPE html>
▼<html>
  ▶<head>…</head>
  ▼<body>
     <h1>Test page</h1>
    ▼<p>
       "With a blow from the top-maul Ahab knocked off the steel head of the lance,
       and then handing to the mate the long iron rod remaining, bade him hold it
       upright, without its touching the deck. Then, with the maul, after
       repeatedly smiting the upper end of this iron rod, he placed the blunted
       needle endwise on the top of it, and less strongly hammered that, several
       times, the mate still holding the rod as before. Then going through some
       small strange motions with it-whether indispensable to the magnetizing of
       the steel, or merely intended to augment the awe of the crew, is uncertain-
       he called for linen thread; and moving to the binnacle, slipped out the two
       reversed needles there, and horizontally suspended the sail-needle by its
       middle, over one of the compass-cards. At first, the steel went round and
       round, quivering and vibrating at either end; but at last it settled to its
       place, when Ahab, who had been intently watching for this result, stepped
       frankly back from the binnacle, and pointing his stretched arm towards it,
       exclaimed,-"Look ye, for yourselves, if Ahab be not lord of the level
       loadstone! The sun is East, and that compass swears it!""
     </p>
    ▶<p>…</p>
    ▶<p>…</p>
```

```
element.style {

}
```

```
body {
    color: ■#333;
    font-family: Arial;
}
```

Figure 1-11 Here, I'm using the Web Inspector in Chrome to inspect elements on an iPhone. Notice how it is even highlighting DOM items on the iPhone.

Bear in mind that some data will be a little bit skewed. For instance, performance data is a little off because the device is running Weinre. Additionally, Weinre data is being relayed over the network, so network data will be off as well. There's really no way around this, and it's good to bear in mind that even native tools suffer from some of these issues.

Version Control

If you're not using version control in your development process, I can't stress its importance strongly enough. I use it on every project, even on tiny projects where I work completely alone. Version control tracks the changes in a codebase over time. Periodically, you set commit points, which you can revert to at any point, or even fold certain changes from one commit into another. It's absolutely essential for collaborating with other developers. Instead of having to figure out who changed what file, version control handles all that for you.

Even if you and another developer worked on the same file, version control can merge those changes. Occasionally, conflicts will occur—for instance, if you both edited the same line of code. In those cases, the version control system will provide a way to merge the changes manually. But even if you're working alone, I strongly encourage that you use version control. Before I used version control, I was constantly commenting out sections I wasn't using anymore, for fear that I might use them later. Now I can just revert changes whenever I want, and leave the codebase a lot cleaner. And if a client changes his or her mind on a bit of functionality, I can revert the codebase with a single command.

Of the variety of version control systems on the market, my personal favorite is Git. Git is very widely used, which is important when selecting software designed for collaboration. Most importantly, Git is distributed version control as opposed to centralized. This type of version control has a number of advantages. First it allows for each user to have their own repo, which can later be merged with the main repo (or other repos), thereby providing extra layers of versioning. Additionally, it is easier to use, because you can set distributed version control up locally without a server (and connect it to a server at a later date). You can find more about how to use Git in the free e-book, *Pro Git*: `http://git-scm.com/book`.

CSS Preprocessing

I'm not going to talk too much about CSS in this book, but your app will undoubtedly involve a certain amount of styling, and for that you should really be using a CSS preprocessor. CSS preprocessors such as SASS and LESS allow you be smarter in the way you write CSS. They provide a wide variety of scripting operations, which are all compiled to a static CSS file. That means you get advantages of dynamic scripting language, while still generating completely valid CSS files that can be read by any browser.

It's a good idea to use a preprocessor in any project with a significant amount of styling. To learn more about CSS preprocessing, turn to Appendix A, where I discuss LESS in detail.

Since CSS preprocessors generate standard CSS, you can always go back to static CSS at any time.

Testing

In order to ensure the quality of the app you're building, it's vitally important to test all the functionality thoroughly. But you shouldn't wait until the app is built in order to set up a testing framework. It's a much better idea to establish a test driven development pattern (TDD) using unit tests.

Unit tests break code into the individual tasks (units) and then ensure that the logic is working as planned. The idea is that once you set up unit tests, you can run them in a variety of different environments, browsers, and so on, to make sure that the app is working exactly as it should. This approach is the best way to weed out edge cases that may otherwise not come up until well after you've launched a product.

Yes, unit testing is more work than just coding. But if you set up a TDD approach, it can actually save you time Q&Aing and debugging pesky issues that crop up in only the rarest of cases. More importantly, having a full suite of tests will improve your confidence in your deliverables and ensure regression issues will be picked up later on in development.

In this section, you learn how to use Grunt to create a build process for your app. You use it to automatically concatenate and minify your JavaScripts, and run these files through linting to test syntax and ensure that coding conventions are being enforced. Next, you set up unit tests using QUnit. Thanks to Grunt, these unit tests will run whenever a file in your app is modified, ensuring high quality code as you write it.

Using Grunt

Grunt is a task runner that provides a number of utilities you can use when building your app. It automates many of the repetitive tasks typically associated with generating a production version of a script. In this section you use Grunt to concatenate and minify your JavaScripts, then run them through unit testing and linting. You can either run these tasks manually, or set them up to run automatically whenever a file is modified.

To use Grunt, you first have to install Node.js by following the instructions on the Node website (`http://nodejs.org/`). Grunt is a command line tool, so open up a terminal window and type the following to globally install Grunt's command line tools:

```
npm install -g grunt-cli
```

Once the command line tools are installed, open up your project's directory and install Grunt:

```
npm install grunt
```

If you want to use the command line tools, you have to install Grunt locally for every project. Alternatively, you can install Grunt globally and add it to your bash profile.

Next, set up three directories for your project:

```
dist
src
test
```

These will contain the distribution and source files for the app, as well as the unit tests. Next you need to create two configuration files for Grunt: a `package.json` and a `Gruntfile.js` (both at the root of your project).

Building package.json and Installing Grunt Plugins

`package.json` stores some basic info about your app, as you can see here:

```
{
  "name": "my-project-name",
  "version": "0.1.0",
  "devDependencies": {}
}
```

This JSON simply contains the name and current version number of your app (which you should fill in), and then defines a number of dev dependencies. These dependencies are the Grunt plugins you want to use in the task runner. But rather than enter these manually, you can automatically populate this list when you install the plugins with NPM:

```
npm install grunt --save-dev
```

The `--save-dev` flag automatically enters the dependency along with its version number in your `package.json` file. Next, you install the rest of the Grunt plugins used in this example:

```
npm install grunt-contrib-concat --save-dev
npm install grunt-contrib-uglify --save-dev
npm install grunt-contrib-qunit --save-dev
npm install grunt-contrib-jshint --save-dev
npm install grunt-contrib-watch --save-dev
```

Building the Gruntfile and Creating Tasks

Now you have to create `Gruntfile.js`, which is the meat of your Grunt implementation. This file contains all the configuration and tasks you want to execute, for example:

```
module.exports = function(grunt) {

  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),

    // concatenation
    concat: {
      options: {
        separator: ';'
      },
      dist: {
        src: ['src/**/*.js'],
        dest: 'dist/<%= pkg.name %>.js'
      }
    },

    // minification
    uglify: {
      options: {
        banner: '/*! <%= pkg.name %> <%= grunt.template.today("dd-mm-
yyyy") %> */\n'
```

```
      },
      dist: {
        files: {
          'dist/<%= pkg.name %>.min.js': ['<%= concat.dist.dest %>']
        }
      }
    },

    // unit testing
    qunit: {
      files: ['test/**/*.html']
    },

    // linting
    jshint: {
      files: ['gruntfile.js', 'src/**/*.js', 'test/**/*.js'],
      options: {
        // options here to override JSHint defaults
        globals: {
          jQuery: true,
          console: true,
          module: true,
          document: true
        }
      }
    },

    // automated task running
    watch: {
      files: ['<%= jshint.files %>'],
      tasks: ['jshint', 'qunit']
    }
  });

  // dependencies
  grunt.loadNpmTasks('grunt-contrib-concat');
  grunt.loadNpmTasks('grunt-contrib-uglify');
  grunt.loadNpmTasks('grunt-contrib-qunit');
  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-contrib-watch');

  // tasks
  grunt.registerTask('test', ['jshint', 'qunit']);

  grunt.registerTask('default', ['jshint', 'qunit', 'concat', 'uglify']);

};
```

To understand what's going on in this Gruntfile, let's rebuild it from scratch, starting with:

```
module.exports = function(grunt) {

  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json')
  });

};
```

This snippet starts the config's initialization function by caching the settings in your `package.json` file. That way those values can be referenced elsewhere in the Gruntfile, as you'll soon see. Next, configure a task for concatenating the scripts in your app:

```
module.exports = function(grunt) {

  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    concat: {
      options: {
        separator: ';'
      },
      dist: {
        src: ['src/**/*.js'],
        dest: 'dist/<%= pkg.name %>.js'
      }
    }
  });

};
```

This code configures Grunt to pull all the `.js` files from the `src/` directory. These files will be concatenated and saved to the `dist/` directory, using the name you defined in `package.json`. Next configure another task to handle minification using UglifyJS:

```
uglify: {
  options: {
    // the banner text is added to the top of the output
    banner: '/*! <%= pkg.name %> <%= grunt.template.today("dd-mm-yyyy") %>
*/\n'
  },
  dist: {
    files: {
      'dist/<%= pkg.name %>.min.js': ['<%= concat.dist.dest %>']
    }
  }
}
```

This snippet configures Uglify to minify all the files produced in the `concat` task (`concat.dist.dest`), and then save them to the `dist/` directory. Next, configure QUnit:

```
qunit: {
  files: ['test/**/*.html']
}
```

The QUnit configuration simply defines the location of the test runner files, which you'll learn how to set up later this section. Now configure a task to handle linting:

```
jshint: {
  files: ['gruntfile.js', 'src/**/*.js', 'test/**/*.js'],
  options: {
    // options here to override JSHint defaults
    globals: {
      jQuery: true,
      console: true,
      module: true,
      document: true
    }
  }
}
```

This configuration first sets the files you want to lint with JSHint, then defines a few options. Linting is a useful step in any test suite, since it checks the syntax of your JavaScript for any errors or poor formatting. Next configure the watch plugin, which runs tasks automatically whenever a file changes:

```
watch: {
  files: ['<%= jshint.files %>'],
  tasks: ['jshint', 'qunit']
}
```

This code configures the watch plugin to watch for any changes in the app's files, in which case it triggers the `jshint` and `qunit` tasks. With the watch configuration in place, any changes you make to your code will be automatically linted and unit tested. That way you can be certain that the code remains error-free and that coding conventions are being enforced.

Next, load in the Grunt plugins you installed earlier:

```
grunt.loadNpmTasks('grunt-contrib-concat');
grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-contrib-qunit');
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-watch');
```

Finally, set up the tasks you want to run, most importantly the default task:

```
grunt.registerTask('test', ['jshint', 'qunit']);

grunt.registerTask('default', ['jshint', 'qunit', 'concat', 'uglify']);
```

To run the default task, simply type `grunt` on the command line. Likewise, the test task can be run using `grunt test`.

Using QUnit

Now that Grunt is set up, you already have a bit of testing coverage through JSHint. But linting only tests the JavaScript for syntax errors and misformatting. You still need to set up unit tests to verify the actual logic of your app. Unit tests ensure that your app behaves as expected in a variety of situations. They provide a fine-grained analysis of your app's functionality, verifying that every step executes as it should.

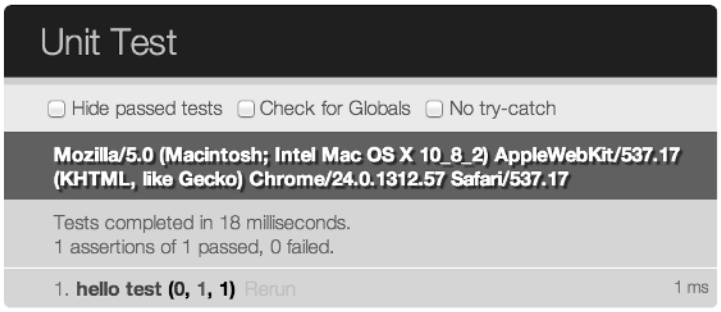
You got a brief glimpse of QUnit when configuring Grunt, where you set up the task runner to run QUnit tests whenever a file changes. But as it currently stands, there are no unit tests in the `test/` directory. This subsection teaches you how to use QUnit and create the tests for your app. QUnit is a JavaScript testing framework employed by jQuery and a variety of other projects. It provides easy-to-use and feature-rich unit testing capabilities.

QUnit Basics

First, download the script and stylesheet from `http://qunitjs.com`. Follow the guidelines on the homepage for setting up the markup for your unit tests page and then run the example test:

```
test('hello test', function() {
  ok(1 == '1', 'Passed!');
});
```

Here the `test()` function defines the title of the unit being tested (`hello test`) and then passes an anonymous function with the tests to run. In this example, the first argument in the `ok()` test passes (`1 == '1'`), so QUnit outputs `Passed!`, as shown in Figure 1-12.



Unit Test

Hide passed tests   Check for Globals   No try-catch

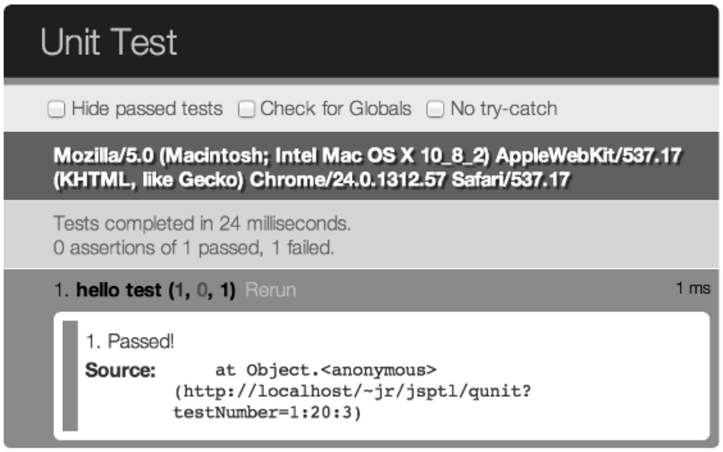
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_2) AppleWebKit/537.17 (KHTML, like Gecko) Chrome/24.0.1312.57 Safari/537.17

Tests completed in 18 milliseconds.
1 assertions of 1 passed, 0 failed.

1. hello test (0, 1, 1) Rerun 1 ms

Figure 1-12 This unit test has passed. QUnit tells you how quickly it ran and allows you to rerun the test.

If you change the argument here to `1 == '2'`, the unit test fails, as shown in Figure 1-13.


Unit Test

Hide passed tests Check for Globals No try-catch

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_2) AppleWebKit/537.17 (KHTML, like Gecko) Chrome/24.0.1312.57 Safari/537.17

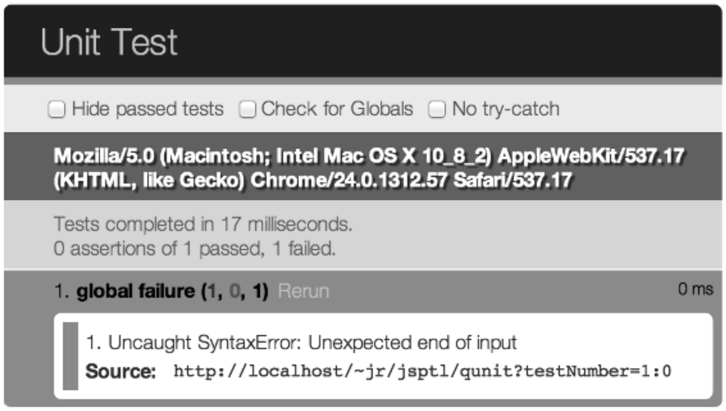
Tests completed in 24 milliseconds.
0 assertions of 1 passed, 1 failed.

1. hello test (1, 0, 1) Rerun 1 ms

1. Passed!
Source: at Object.<anonymous> (http://localhost/~jr/jsptl/qunit?testNumber=1:20:3)

Figure 1-13 When the unit test fails, QUnit outputs a fail message and the source of the error.

Additionally, even if you have a syntax error in your code, QUnit outputs a global error, as shown in Figure 1-14.


Unit Test

Hide passed tests Check for Globals No try-catch

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_2) AppleWebKit/537.17 (KHTML, like Gecko) Chrome/24.0.1312.57 Safari/537.17

Tests completed in 17 milliseconds.
0 assertions of 1 passed, 1 failed.

1. global failure (1, 0, 1) Rerun 0 ms

1. Uncaught SyntaxError: Unexpected end of input
Source: http://localhost/~jr/jsptl/qunit?testNumber=1:0

Figure 1-14 A syntax error is throwing this global error.

Digging into QUnit

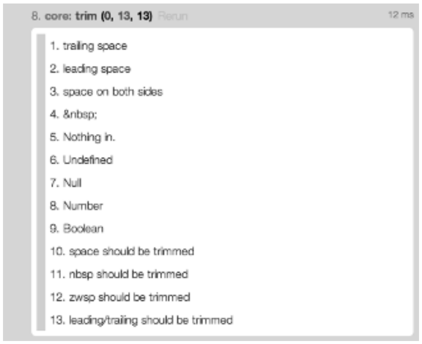
These examples have all been pretty basic. To get an idea of how QUnit works on an actual project, take a look at the QUnit tests for jQuery, which you can download here: `https://github.com/jquery/jquery/tree/master/test/unit`.

Notice that the unit tests are sequestered in a separate directory: `test/unit/`. Unit tests should never make their way into the production codebase; they're strictly for debugging and typically contain a considerable amount of extra code.

Now, open `core.js` and scroll down to `test("trim", function{ ... })`—it's at line 233 in the jQuery v1.9.1 that I'm using at the time of this writing. Here you see the following tests for the `jQuery.trim()` method:

```
test("trim", function() {
  expect(13);

  var nbsp = String.fromCharCode(160);

  equal( jQuery.trim("hello  "), "hello", "trailing space" );
  equal( jQuery.trim("  hello"), "hello", "leading space" );
  equal( jQuery.trim("  hello   "), "hello", "space on both sides" );
  equal( jQuery.trim("  " + nbsp + "hello  " + nbsp + " "), "hello",
"&nbsp;" );

  equal( jQuery.trim(), "", "Nothing in." );
  equal( jQuery.trim( undefined ), "", "Undefined" );
  equal( jQuery.trim( null ), "", "Null" );
  equal( jQuery.trim( 5 ), "5", "Number" );
  equal( jQuery.trim( false ), "false", "Boolean" );

  equal( jQuery.trim(" "), "", "space should be trimmed" );
  equal( jQuery.trim("ipad\xA0"), "ipad", "nbsp should be trimmed" );
  equal( jQuery.trim("\uFEFF"), "", "zwsp should be trimmed" );
  equal( jQuery.trim("\uFEFF \xA0! | \uFEFF"), "! |", "leading/trailing
should be trimmed" );
});
```

These assertions test the functionality of `jQuery.trim()` using a range of possible input parameters. These tests ensure the expected return value when these parameters are passed in: trailing spaces, leading spaces, empty strings, Boolean values, and any other unique situation you can think of. As you can see, the jQuery team has covered a wide array of edge cases in these tests. And as shown in Figure 1-15, jQuery passes all these tests.



```
8. core: trim (0, 13, 13) Rerun                12 ms
1. trailing space
2. leading space
3. space on both sides
4. &nbsp;
5. Nothing in.
6. Undefined
7. Null
8. Number
9. Boolean
10. space should be trimmed
11. nbsp should be trimmed
12. zwsp should be trimmed
13. leading/trailing should be trimmed
```

Figure 1-15 `jQuery.trim()` passes all its unit tests.

Digging deeper into the code, the first line in the test function is `expect(13)`, which asserts that the test runs exactly 13 assertions. Then, for the first actual assertion, you see:

```
    equal( jQuery.trim("hello  "), "hello", "trailing space" );
```

This line uses QUnit's `equal()` method, which determines whether the first two parameters are equal. In this case, the test expects `jQuery.trim("hello ")` to return `"hello"`, in which case it passes.

You can do a lot more with QUnit, and I encourage you to read the QUnit docs (`http://api.qunitjs.com`) and to poke around in the jQuery tests to get a feel for how the test framework works. Or check out the tests for QUnit itself.

Setting Up Your Own QUnit Tests

Now that you know the basics of working with QUnit, you're ready to set up your own tests. It will take a little more time than coding without tests, but it shouldn't be too hard if you're already following best practices (object orientation, separating concerns, and so on).

The main place you should focus is on making your tests atomic—they should test as little functionality as possible. That doesn't mean you should test every single line separately, but rather that you should split your code into the smallest tasks possible, and test each one. The finer your tests are, the easier it will be to track down the cause of a test failure.

You should also make sure the tests are completely independent—the order you run your tests shouldn't matter, and you should be able to run any individual test without requiring another. Additionally, make sure that you document the tests with meaningful error messages. If a unit test fails the last thing you want to do is track down what the test is even checking.

Last but not least, don't wait until your app is built to write your tests—you should be writing them as you go along. Every time you add a new feature, it should be complimented with a new suite of tests. Get yourself in the mindset of TDD; a new piece of code isn't finished until you've finished the tests.

Save every test you write to the `tests/` directory of your project, so that Grunt can run them automatically. Now, whenever you save a change to your scripts, Grunt will perform a thorough sanity check and ensure that bad code never even makes it into the codebase.

Summary

In this chapter, you discovered the importance of loose coupling and separation of concerns, which I focus on throughout this book. You found out about MVCs and the different roles of the model, view, and controller. You also learned the importance of using JavaScript templates to display the view.

Next you read about the Chrome Developer Tools and how you can use them to debug your app and profile performance. You also learned how to access these tools in any browser and on any device, using the remote tool Weinre. Additionally, you discovered the importance of using version control, and also how to speed up your CSS development with CSS preprocessing.

Then you learned how to establish a test driven development pattern. You discovered the task runner Grunt and used it to automate repetitive tasks such as concatenating and minification. You also used it to automatically test the codebase whenever a file changes, using both linting and unit tests. Finally, you explored QUnit and learned how to create effective unit tests.

Now that you've learned some best practices and established a thorough testing approach, you can be confident that the code you'll write in the rest of this book will be of the highest quality.

Additional Resources

Engineering Best Practices

Separation of Concerns: `http://en.wikipedia.org/wiki/Separation_of_concerns`

Best Practices When Working With JavaScript Templates: `http://net.tutsplus.com/tutorials/javascript-ajax/best-practices-when-working-with-javascript-templates/`

Introduction to Test Driven Development: `http://www.agiledata.org/essays/tdd.html`

Chrome Developer Tools

Chrome Developer Tools Documentation: `https://developers.google.com/chrome-developer-tools/docs/overview`

Google I/O 2011: Chrome Developer Tools Reloaded: `http://www.youtube.com/watch?v=N8SS-rUEZPg`

Breakpoints: `https://developers.google.com/chrome-developer-tools/docs/scripts-breakpoints`

Weinre

Weinre Documentation: `http://people.apache.org/~pmuellr/weinre-docs/latest/Home.html`

Weinre — Web Inspector Remote — Demo: `http://www.youtube.com/watch?v=gaAI29UkVCc`

Version Control

Pro Git by Scott Chacon: `http://git-scm.com/book`

A Visual Guide to Version Control: `http://betterexplained.com/articles/a-visual-guide-to-version-control/`

7 Version Control Systems Reviewed: http://www.smashingmagazine.com/2008/09/18/the-top-7-open-source-version-control-systems/

CSS Preprocessing

LESS Documentation: `http://lesscss.org/`

Sass Documentation: `http://sass-lang.com/`

Testing

Johansen, Christian. *Test-Driven Javascript Development*. (Pearson Education, Inc., 2010)

Grunt: http://gruntjs.com/

QUnit. "Introduction to Unit Testing": `http://qunitjs.com/intro`

QUnit. "QUnit API Documentation": `http://api.qunitjs.com`

JSHint Documentation: `http://www.jshint.com/docs/`