JAVASCRIPT TESTING RECIPES

JAMES COGLAN

Contents

List of Figures

Change history

1. (1.0.0) March 23rd 2014

- The first edition.

2. (1.0.1) April 1st 2014

- Corrections for some minor typographical errors.

3. (1.0.2) December 30th 2014

- Upgrades all the npm packages to their latest versions.
- No changes to the text.

4. (1.1.0) May 31st 2016

- Upgrades all the npm packages to their latest versions. Examples should now run on Node.js versions 0.10, 0.12, 4, 5 and 6.
- Updates the sections on Testium, which has changed its user interface since the last edition. See Section 5.4.5, “Testing via the browser”, Section 5.5, “Client-side JavaScript”, and Section 5.6.3, “Full-stack testing”.
- Dependencies are now managed with npm3, which has made it easier to provide predictable installs and isolate packages that need to be excluded from the distribution.
- The `browserify` package no longer depends on an unlicensed release of `zpipe` and so is included in the bundle.
- The `ws` package no longer relies on native code and so `testem` is included in the bundle.
- `webdriver-http-sync` relies on native code and is excluded from the bundle, but the rest of the `testium` dependency tree is included.
- The example code is now covered by the Apache License, Version 2.0.

License and acknowledgements

This book is produced using free and open source software. The text is written using the AsciiDoc[1] language, formatted using DocBook XSL[2] and Pygments[3], and compiled using xsltproc[4], Calibre[5] and Apache FOP[6]. The code examples are tested using PhantomJS[7] and Node.js[8].

I am grateful to George Brocklehurst, Murray Steele, and Tom Stuart for their invaluable feedback and encouragement on my drafts, and for suggesting numerous improvements to the text. I must also thank Chris Strom for his help with the above tools used in the book’s production.

Finally, a big thank-you to Mazz, whose feedback, support and encouragement has kept me going during the last six months.

To industry…

[1]http://www.methods.co.nz/asciidoc/
[2]http://docbook.sourceforge.net/
[3]http://pygments.org/
[4]http://xmlsoft.org/XSLT/xsltproc2.html
[5]https://calibre-ebook.com/
[6]http://xmlgraphics.apache.org/fop/
[7]http://phantomjs.org/
[8]http://nodejs.org/

1. Introduction

"Well isn't this unexpected."

— Mark Z. Danielewski *House of Leaves*

In the novel *House of Leaves*, a mysterious corridor suddenly materialises inside the Virginia home of filmmaker Will Navidson. On further investigation, the corridor leads to a labyrinth — a pitch dark series of rooms, corridors and staircases that seem to go on without end. Navidson's explorations reveal that, not only is the labyrinth infinite, but it contorts itself into a series of Escheresque forms, often in response to the psychological disposition of those within it. As Navidson and his friends get more and more lost, their fear and the shape-shifting maze pull them farther and farther from home, until Navidson finds himself alone, standing on a floating slab in the center of a huge empty void, chuckling to himself and feigning surprise at his predicament.

Not only is this scenario familiar to anyone who builds software for a living, it's an apt metaphor for the rise of JavaScript itself. We saw it coming years in advance, but we continue to wonder at the fact that the world's most misunderstood programming language has finally eaten the web. The adoption of Node has pushed server-side JavaScript into the big time, we have databases with embedded JavaScript engines, we're even piloting drones with the stuff. Every other teach-yourself-to-code programme uses JavaScript as its main learning vehicle, and with some justification: it is now possible to build a complete web application, or indeed any conceivable program you like, using only this one programming language.

But JavaScript isn't exactly the most helpful tool when it comes to helping us avoid mistakes. It is a highly dynamic language, where you can add and remove data and methods from almost any object you choose at runtime. Indeed, there is *only* runtime: JavaScript does not go through a compiler that will find errors for you before you run your program. The very existence of the book *JavaScript: the Good Parts* tells you everything you need to know about the language: it is filled with warts, design missteps and gotchas just waiting to surprise and confound you. Did you know that `typeof null` evaluates to `"object"`? Or that `"5" + 2` is `"52"` while `"5" - 2` is `3`? Or that you can pass `null` to a `for-in` loop without getting an error? Or that `["10", "10", "10", "10"].map(parseInt)` is `[10, NaN, 2, 3]`? Or that `[0] == 0` is `true` but `[0] == [0]` is `false`?

Even when JavaScript does spot that you did something wrong, it can be less than clear about what the problem is. Errors like 'Object doesn't support this property or method' and 'undefined is not a function' are all too common. If you introduce a bug into your program, it can be very hard to discover the root cause, or even to discover the symptoms: it is all too easy to deploy the bug to production and let one of your users discover it.

In short, we're building anything and everything in a language that makes it really hard to make sure our code actually works. We need all the help we can get.

1.1. Why test?

Many people, when first introduced to the idea of writing automated tests, find it counter-intuitive. How can writing more code — often double the amount of code as you'd write for the app itself — possibly make us more productive?

JavaScript's dynamism makes it a good language for writing new code in, for quickly writing prototypes to try out an idea, but the things that make it great for building new things make it not-so-great for maintaining old things. If you want to change the arguments a function takes, or remove the function entirely, it can be hard to tell where that function is being used so you can update the dependent code. Since there is no static type system, it can be hard to tell what kinds of values a function accepts, and what properties they must have. It's too easy to accidentally mis-handle a value and let JavaScript implicitly convert it to a string. And it's not just your code you need to deal with; any non-trivial project will import thousands of lines of third-party code, all of which can add methods to built-in classes, hide important implementation details from you, and introduce its own bugs and surprises for you to avoid.

Faced with the task of maintaining software on a platform that does so little to alert us to our mistakes, we must spend a great deal of effort on testing to avoid shipping broken software to our users. Manual testing, where a developer or quality assurance person interacts with the product, going through every conceivable pathway and making sure everything works, on every browser your team supports, takes a huge amount of time. And since it’s boring repetitive work, it is easy to miss important steps and overlook fine details. Developers are quickly frustrated by long, boring error-prone processes that block them from shipping, and will often try to route around them, assuming we have thought of every eventuality and cannot possibly have missed any bugs.

The thing is, a lot of the things we check during manual testing can be done much more quickly and predictably by a computer. Computers are *great* at doing boring repetitive time-consuming work for us. Checking that if you visit the login page and enter a correct username and password, you are then taken to the dashboard page, can all be done using an automated script. The script can process hundreds, even thousands of pathways in your codebase in a few seconds, where it would take a person minutes or hours to do the same thing. The script can check the app in a lot more detail, since it can be made to interact with the internal functions inside the app and check every component that makes up the software, not just the end product. And, because it’s a mechanical computer program instead of a person, it’s a lot more predictable: it will always carry out exactly the same set of checks as last time, it doesn’t have the capacity to get bored and overlook things.

Most importantly, having an automated test suite means you can confidently make any change you want to the code, knowing you can run a quick program that tells you whether you broke anything. Not only does this mean you can continue to work quickly as your codebase grows, in some cases it can make the difference between making any change at all, or holding back out of fear of breaking something that seems to be working well enough.

To but it briefly: automated testing helps you manage risk. By putting checks in place to quickly ensure your software keeps working properly, you can take more risk in changing the software, trying out new things and throwing old things away, and reacting to the changing needs of your users.

1.2. Why should I change my code?

After the initial resistance to writing tests at all, the second most frequent problem I see when teaching people how to test is that they don’t understand why they should change how they write code in order to make it easier to test. Surely, the design of the code should be driven by the demands of the problem, and should not be changed to suit the demands of code we’re never going to run in production.

The way I see it, writing code to make it easy to test is another way of saying that code should, as many would agree, be easy to read. We like easy-to-read code because it helps us to understand what the program does, so that we can figure out how to use it and how we might usefully change it. Tests are another way of achieving the same ends, and more besides. While reading a program might tell you *what* it does, it might not tell you *why* it does that. Writing tests can help illustrate the code’s purpose, its *raison d'être*, by providing a set of representative use cases. Many developers view tests as documentation for how a module ought to be used; tests are often the examples you would put in a `README` document.

Just as important as knowing what code does, is knowing what it does *not* do. Unit testing helps us verify how a module interacts with other modules in the system, and gives us a way to check that certain interactions are prevented from happening. Tests are often used to check what happens when a program is given unexpected input, either due to user error or malicious attacks, ensuring the program deals with such inputs safely and gracefully. It is useful to have automated tools that deal with such concerns, so that once we’ve dealt with them we can better focus on what the program *should* do.

We often say that we should try to solve a problem with the least amount of code we can, that we should not build functionality that we don’t yet need, that we should aim to keep things simple. The fundamental reason for all these practices is that we want to build software as *cost-effectively* as possible, both in the sense of the initial build running quickly, and of the ensuing maintenance not becoming costly. Automated tests, by giving us the freedom to make changes and alerting us to our mistakes, can reduce the cost of maintenance over the medium and long term, often to the extent that it’s worth the additional work of designing your software to be amenable to testing. Remember that the cost of a software project is far more than the amount of time it takes us to type code; a big contributor to overall costs is the time we spend checking our work, and any change that can make a dent in that cost must be worth considering.

But it’s not just the testing burden you’re easing by writing code in a testable way. Testable code has virtues that tend to bestow other benefits. Modules are easiest to test when they are small, focussed on one concern, not excessively coupled to other parts of the system, and have well-defined APIs and few dependencies. These same attributes also make modules easier to understand, to reuse, and to recombine in unanticipated ways, making it easier to change how your software works in small pieces rather than rebuilding huge systems in one go. Making code more modular reduces the cognitive load of working on any component of it, and helps us work more productively.

Many programmers experienced in automated testing also view it as a design tool. The act of trying to write tests for their code pushes them to design good APIs, because writing tests makes you interact with the modules you build, as you build them. When something is painful to test, this acts as design feedback: maybe the test requires too much elaborate setup before the real work can be done, and this could indicate the module you’re testing is too coupled to other concerns.

But if we demand code that is easy to read, we must demand this doubly of our tests. If the purpose of tests is to mitigate risk, to make us more confident to change things, then it does not do much good to have tests that are just as hard to understand as the code they are testing. The confidence to change things comes from having an artifact that is *simpler* than the code itself, that tells us what the code really does in a way we can easily verify ourselves.

So at first, it can seem odd to write twice or three times as many lines of code in our tests as exist in our application, but the key is in what *kind* of code the tests contain. Tests will naturally be more verbose than the code they test; while the code will attempt to be *general*, to handle all possible situations in as few words as possible, tests are *specific*: they enumerate a set of concrete scenarios that explore different pathways in the code. But although they are more verbose, good tests are less complex; they rarely, if ever, are anything but a sequential list of steps to carry out, with no loops or conditional statements. This makes them much easier to read and understand than most production code, since there is no branching logic to hold in your head while reading them. Programming is often likened to writing a recipe for what you want a computer to do, but tests are really much closer to cooking recipes than code is.

In truth, this book is as much about software architecture as it is about testing *per se*. The key to effectively testing a system is often to break it into small pieces that are easier to test on their own, and there are many examples here of how to divide up programs in this way. Even when the design of the code is not explicitly called out, I’d encourage you to consider what it is that makes a component easy to test. Often the answer lies as much in those things the component does *not* do as in those it does.

1.3. What is this book not about?

The JavaScript ecosystem is vast, diverse and fast-moving, with new libraries and frameworks emerging all the time. While I have tried to cover a broad range of problems faced by most developers, there is simply not space to cover every tool in use today. And those tools themselves are under active development, with new features being added and existing features being tweaked. It’s a tough job keeping up with it all.

This book would be of little use to anyone if it was full of material that will be out of date next month. It is not intended to act as documentation for any of the tools it covers, nor to suggest that any of those tools represent the one true way to write testable JavaScript. Those libraries I have chosen are here because they are widely used today, or because I found them useful to illustrate certain concepts. I do not mean to imply that any of them are objectively better than the alternatives, and I encourage you to experiment with different tools to find out which ones work for you.

Rather than try to exhaustively cover every conceivable problem faced by today's developers, I've tried to use common problems to illustrate a more fundamental idea: that any problem you face when testing code can be tackled using a small set of general techniques. Writing modular code with few hard-coded dependencies, separating concerns, decoupling components via dependency injection or event systems, avoiding global shared state, and focussing on testing components in isolation rather than writing many full-stack integration tests are all themes that run through this material. You should read each example, not as a prescription for how you should always solve a problem, but as an illustration of how those techniques can be applied in many different contexts.

The challenge in each case comes from the program's interaction with its environment: the way it talks to platform APIs and other components of the application. The trick is to not let the specifics of the problem cloud your thinking; interacting with a database and receiving messages via a WebSocket are not that different if you ignore what the code 'means' and focus on its 'language-level' properties. How do the objects and functions you're using interact, which properties and methods do they use, and how might you structure things so that those interactions are simpler and more explicit?

Ultimately, this book is only about JavaScript in the sense that it covers common platform-related issues faced by JavaScript developers. The techniques used to solve these problems are the same techniques I use when writing Ruby or Python; not only do they transcend whatever framework you're using today, they have a rich history in the wider object-oriented programming ecosystem, and you can learn just as much about testing your JavaScript by dabbling in other languages and seeing how they do things.

1.4. How the book is structured

The range of JavaScript applications being as broad as it is, this book covers a lot of different kinds of problems, both in the browser and on the server. While I think it will improve your understanding to read examples for programs you don't usually write, I have tried to organise the material in a modular way, so that you do not have to read the entire thing from start to finish.

Chapter 2, *Building blocks* is the one chapter I recommend that everyone read, since it covers all the general concepts we'll use throughout the rest of the book. It explains the essential ingredients we have to work with when writing tests, and what those ingredients ought to be used for, in general terms. The rest of the material really just consists of many different applications of the small set of general principles in this chapter, and so refer to it frequently.

The other chapters can be read in any order, although I would advise that you read Chapter 3, *Events and streams* first. Although not essential, many problems in JavaScript programs rely on the material in this chapter and you may find it helpful to start there. The chapters themselves often use a single core example application, building it up in stages to illustrate different ideas, and should largely be read from start to finish.

1.5. Coding conventions

JavaScript, being the small and malleable language that it is, and being written by so many people with different backgrounds, and being applied in so many different problem domains, is written in a variety of different styles. There are many schools of thought on how one should declare a variable, or a function, or a class, or even on whether classes are even a thing. In this book's code examples, I have aimed for a style that makes good use of JavaScript's native features, allows for the application of a range of object-oriented programming techniques, and that does not rely on any external libraries.

Technically speaking, JavaScript does not have ‘classes’ in the sense that other object-oriented languages do. However, many JavaScript users use the word *class* to refer to a constructor function and its prototype. For example, say we want a class to represent blog posts, with a method to get the word count of the article. In this book, I will write it like this:

Figure 1.1. Defining a class

```
var BlogPost = function(title, body) {
  this._title = title
  this._body  = body
}

BlogPost.prototype.wordCount = function() {
  return this._body.split(/\s+/).length
}
```

You would use this class as follows; creating an object from a class is known as *instantiation* or *construction*.

Figure 1.2. Instantiating a class

```
var post = new BlogPost("Classes in JavaScript", "Technically, JS does not have classes")

post.wordCount()
// returns 6
```

The above example demonstrates a few naming conventions we will use throughout the book. None of these is enforced by JavaScript itself, they are merely conventions many people follow. First, functions that are intended to be used as constructors (i.e. called with `new`) are named in camel-case with a leading uppercase letter. Private attributes and methods and named in camel-case with a leading underscore; JavaScript does not actually have true private attributes, but this naming convention serves to tell people they're not supposed to access certain properties from outside an object. Regular functions, methods, and variables are named in camel-case, following the pattern of JavaScript's built-in APIs.

When we want to inherit from a class, we'll use the `util.inherits()` method. This is a built-in method in Node, and I have provided an implementation of it for the browser. For example, if we wanted a new kind of blog post that has tags, we would write:

Figure 1.3. Subclassing using `util.inherits()`

```
var TaggedBlogPost = function(title, body, tags) {
  BlogPost.call(this, title, body)
  this._tags = tags
}
util.inherits(TaggedBlogPost, BlogPost)
```

This demonstrates the general pattern we will use when subclassing. We create a new constructor function, and that applies the ‘parent’ constructor to the new object — that's the `BlogPost.call(this, title, body)` line — before adding its own constructor logic. Once we've defined the constructor, we call `util.inherits()`, which sets up the prototype chain so that `TaggedBlogPost` inherits all the methods from `BlogPost`.

The subclass is then free to override any method it likes, and add more of its own. For example, say we wanted to override the `wordCount()` method to include the tags in the count. We would store a reference to the original definition, then add a new `wordCount()` method to `TaggedBlogPost.prototype` that calls the original one and adds the number of tags to the result.

Figure 1.4. Overriding a method

```
var wordCount = BlogPost.prototype.wordCount

TaggedBlogPost.prototype.wordCount = function() {
  return wordCount.call(this) + this._tags.length
}
```

One final naming convention you should be aware of is that if you see a variable or property that’s named in `UPPERCASE_WITH_UNDERSCORES`, that value is intended to be a constant. Like private variables, JavaScript does not actually have proper constants and it will still let you change a value named like this; the name serves to indicate the author’s intent.

If you’re not familiar with how functions work in JavaScript, see Appendix A, *JavaScript functions*.

1.6. Example code

This book contains hundreds of examples of how to test various sorts of programs. Where possible, these examples have been included in executable form in the `Code` directory that came with the book. You’re encouraged to run these examples for yourself and try changing them so you can see how they work.

The examples are organised by the type of platform they run on — `browser`, `node`, and so on. You’ll also find directories called `node_modules` and `vendor`, which contain third-party libraries used in the examples. Any file named `test.html` can be opened in a web browser and will run the tests for that example and display the results. Likewise, any file named `test.js` can be run with Node[1] and will display the results in the terminal, as shown in Figure 1.5, “Terminal output using the default `dot` format”.

Figure 1.5. Terminal output using the default `dot` format

```
$ node node/hello_world/test.js
Loaded suite: Hello, world!

.

Finished in 0.007 seconds
1 test, 1 assertion, 0 failures, 0 errors
```

You can change the output format using an environment variable, for example:

Figure 1.6. Terminal output using the `spec` format

```
$ FORMAT=spec node node/hello_world/test.js

Hello, world!
  * runs a test

Finished in 0.007 seconds
1 test, 1 assertion, 0 failures, 0 errors
```

You can also get machine-readable output such as the JUnit XML format:

Figure 1.7. Terminal output using the `xml` format

```
$ FORMAT=xml node node/hello_world/test.js
<?xml version="1.0" encoding="UTF-8"?>
<testsuites>
    <testsuite name="Hello, world!" tests="1" failures="0" errors="0" time="0.002">
        <testcase classname="Hello, world!" name="runs a test" time="0.001"/>
    </testsuite>
</testsuites>
```

[1]http://nodejs.org/

If you have PhantomJS[2] installed, you can also run any of the browser examples in your terminal using the supplied `phantom.js` script; you can use the `FORMAT` setting just as you would for Node scripts.

Figure 1.8. Running browser tests with PhantomJS

```
$ phantomjs phantom.js browser/hello_world/test.html
Loaded suite: Hello, world!

.

Finished in 0.007 seconds
1 test, 1 assertion, 0 failures, 0 errors
```

Finally, the `Code` directory contains a file called `package.json`, which lists all the third-party Node modules used in the examples. For licensing and portability reasons, not all those modules are included in the download; this file will tell you which version of each module I used when writing the examples, should you need to install them yourself.

[2]http://phantomjs.org/

2. Building blocks

Before we can start discussing how to test real programs, we need to get to know our tools — the ingredients that go into writing automated tests. There are a lot of great tools for testing JavaScript out there, but rather than focus on any of them in particular let's focus on what they have in common: what are the underlying concepts that all these frameworks are based on?

2.1. What is an automated test suite?

An automated test suite is simply a program that tells you whether another program does its job correctly. If that sounds a little weird at first, think of it like this: you've probably done a lot of manual testing in the past, interacting with websites and checking everything works. A lot of that was probably boring, repetitive, time-consuming work. Well, computers are great at boring, repetitive, time-consuming work, and we can write programs that do a lot of this work for us.

A test suite is a program that pokes at your application and makes sure it does the right thing, only because it's a program and not a human process, we can run it whenever we like and it'll run through exactly the same steps. It will also do this much *faster* than a person can, saving you a lot of time over the life of a project.

Test suites typically work by calling functions within the application, either directly via a literal function call or indirectly via an HTTP request or running a command-line application, and checking that these functions return the right values, or have the right side effects. Imagine we have a function that adds two numbers together:

Figure 2.1. A simple `add()` function

```
var add = function(x, y) {
  return x + y
}
```

It's a silly example, and it's 'obviously' correct[1], but let's say we want an automated test for this. In most testing frameworks, we do this by giving a few example inputs to the function and checking they produce the right output, something like:

Figure 2.2. Assertions for the `add()` function

```
assertEqual( 2, add(1, 1) )
assertEqual( 4, add(2, 2) )
assertEqual( 8, add(9, -1) )
```

Some frameworks put the expected (hard-coded) value first, and the actual (expression-based) value second, while others do it the other way around. See Section 2.4, "Assertions" for further explanation.

We want to be able to run these assertions and find out whether the `add()` function works. The simplest implementation of `assertEqual()` therefore is one that throws an error if its inputs are not equal.

Figure 2.3. A possible `assertEqual()` implementation

```
var assertEqual = function(expected, actual) {
  if (expected !== actual) {
    throw new Error("Expected " + expected + ", got " + actual)
  }
}
```

[1]Where programming is concerned, there is no such thing as obviously correct. If you find yourself thinking this, you're probably staring right at the source of a bug.

Sure enough, if we paste the `add()` and `assertEqual()` functions into a browser console and try out some examples, an error is thrown if we supply a deliberately wrong value:

```
> assertEqual( 5, add(2, 2) )
Error: expected 5, got 4
```

That's really all an automated test suite is: a program that states certain assumptions about your program, calls functions and tells you if these assumptions aren't true.

2.2. Testing frameworks

In practice, almost no-one writes their own assertion functions like we have above. Except for very small projects, having your test suite be a single script with a lot of assertions and no structure tends to scale poorly and becomes hard to maintain. People like to organise their assertions into related groups, apply shared setup functions to them, run subsets of the tests rather than the whole suite, and get pretty output that's suitable for their platform, for example some clearly presented results on a web page instead of some log messages in the console.

This is what *testing frameworks* do. They provide a way to organise your tests into groups, run them either together or individually, and get helpful output, sometimes including machine-readable output like TAP[2] or XML.

There are lots of popular testing frameworks for JavaScript, and while they all have different approaches to performing the above tasks, and different features sets and APIs, they all do basically the same thing. Jasmine[3], QUnit[4], and Mocha[5] are among the current favourites, and are all extensively documented, blogged and discussed online. I encourage you to try them all out and see which one you're happiest with.

For this book I'm going to use `jstest`[6]. Since I wrote this framework, I am of course biased, but it looks similar to various other tools and is based on many of the same concepts. It has the nice property that it's a single JavaScript file that works on any JS platform out of the box, so we don't have to waste time on getting tools set up before we can really get our teeth into writing some tests.

You will find analogies to almost everything in this book in every other testing framework, and over the course of your career you will probably end up using several of them, so try to focus on the underlying concepts rather than the particular API of the tools we're using.

2.3. Hello, world!

When you pick up a new framework, it's helpful to write a 'hello world' test with it so you know how to set it up. Here's a quick example for `jstest`, which you can find in `browser/hello_world/test.html`. (See Section 1.6, "Example code" for instructions on running the bundled code examples.)

[2]http://testanything.org/
[3]http://jasmine.github.io/
[4]http://qunitjs.com/
[5]https://mochajs.org/
[6]http://jstest.jcoglan.com/

Figure 2.4. `browser/hello_world/test.html`

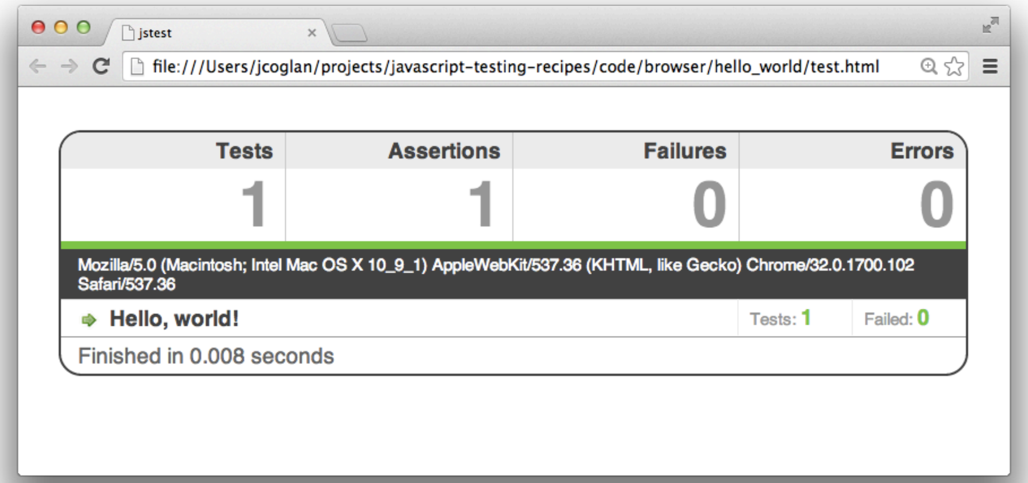
```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>jstest</title>
  </head>
  <body>

    <script src="../../node_modules/jstest/jstest.js"></script>

    <script>
      JS.Test.describe("Hello, world!", function() { with(this) {
        it("runs a test", function() { with(this) {
          assertEqual( "Hello, world!", ["Hello", "world!"].join(", ") )
        }})
      }})

      JS.Test.autorun()
    </script>

  </body>
</html>
```

There is very little going on here. We have a bare-bones HTML document that loads `jstest.js` using a `<script>` tag, defines a test using `JS.Test.describe()`, then runs the tests using `JS.Test.autorun()`. That final command runs all the tests we've defined — just one in this case — and displays the results in a format suitable for the current environment.

Don't worry about the syntax of the test itself, we'll get to that shortly.

If you open this example in a browser, you'll see the result of the test as shown in Figure 2.5, "'Hello world' in the browser": one test, with one assertion, and no failures or errors. You can click on the test names to see their contents, and click the little arrows on the left to run only that group or test.

Figure 2.5. 'Hello world' in the browser


jstest
file:///Users/jcoglan/projects/javascript-testing-recipes/code/browser/hello_world/test.html

Tests	Assertions	Failures	Errors
1	1	0	0

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/32.0.1700.102 Safari/537.36

Hello, world! Tests: 1 Failed: 0

Finished in 0.008 seconds

If you're a Node user, you should also familiarise yourself with what 'hello world' looks like on that platform. It's really the same process that we used in the browser, only instead of a `<script>` tag we use the `require()` function to load the framework. After that the code is exactly the same.

Figure 2.6. `node/hello_world/test.js`

```
var JS = require("jstest")

JS.Test.describe("Hello, world!", function() { with(this) {
  it("runs a test", function() { with(this) {
    assertEqual( "Hello, world!", ["Hello", "world!"].join(", ") )
  }})
}})

JS.Test.autorun()
```

If you run this file with Node, you'll see the result of the test in your terminal. The `JS.Test.autorun()` function detects which platform you're using and displays the test results in an appropriate way.

Figure 2.7. 'Hello world' in the terminal

```
$ node node/hello_world/test.js
Loaded suite: Hello, world!

.

Finished in 0.007 seconds
1 test, 1 assertion, 0 failures, 0 errors
```

These examples should demonstrate that a test framework is really no different from any other JavaScript module you're used to using; you load it in the usual way and use its API to tell it what to do. The only thing that will change as our examples get more involved is that we'll also be loading our application code from other files, and the tests may move into their own files too rather than being inline in the HTML document or Node script. In fact, keeping the tests in separate files makes it easy to run them on Node *or* in the browser, if you're trying to write cross-platform code!

Now, let's get into the ingredients that make up a test suite, beginning with assertions.

2.4. Assertions

Assertions are the foundation of test suites. An assertion is a statement about the program you're testing, saying that a certain thing must be true in order for the program to be considered correct. Assertions are typically written as checks on the output of functions, to make sure they do what we expect, or as statements about the state of a system to make sure some series of commands had the right effect. For example, we might check that some user interaction left the DOM in the correct state, or saved the right information in a database.

The most basic assertion is one that tells you if a statement is true or false, for example:

```
assert( 2 + 2 === 4 )
```

The `assert()` function throws an error if the value given to it is falsey[7], and silently does nothing otherwise. Most assertion functions work this way: they throw an error if what they assert is not true.

But `assert()` on its own is not very useful, since when it fails it can't give you helpful output to say *why* it failed. All it knows is that the value you passed in was not truthy — it can't see the expression you used in the test, only its value — and so all its error will say is something like "`false` is not `true`". And so most testing frameworks provide a rich collection of assertions that do more complicated things and provide better feedback about why things are broken.

[7]In JavaScript, the following values are treated as `false` for Boolean logic: `false`, `null`, `undefined`, `NaN` (not a number), `0` (zero), and `""` (the empty string). All other values, including all arrays, objects, and functions, are *truthy*.

The most common kind of assertions you'll see deal with equality. We express things like "`2 + 2` should equal `4`", or "`localStorage.username` should equal `"bob"`". Since this is such a widely used concept, I'll cover how several testing frameworks deal with it here.

2.4.1. The Node `assert` module

Node contains an `assert`[8] module whose API is widely mimicked. It provides these functions for checking if two values are equal or not:

Figure 2.8. Node equality assertions

```
assert.equal(actual, expected)              // checks actual == expected
assert.notEqual(actual, expected)           // checks actual != expected

assert.strictEqual(actual, expected)        // checks actual === expected
assert.notStrictEqual(actual, expected)     // checks actual !== expected

assert.deepEqual(actual, expected)          // recursively checks equality
assert.notDeepEqual(actual, expected)       // recursively checks inequality
```

The difference between coercive (==) and strict (===) equality is beyond the scope of this book[9] but it's important to know which one your tests are using. The concept of *deep equality* differs slightly between frameworks, but broadly speaking two objects or arrays are `deeply equal' if they contain the same data, i.e. all the data within one object is either deeply or strictly equal to the corresponding data in the other object.[10] In Node, `assert.deepEqual()` says two values are equal if at least one of the following is true:

- They are strictly equal according to the === operator
- They are both `Buffer` objects of the same length and containing the same bytes
- They are both `Date` objects that represent the same time
- They are both regular expressions with the exact same source and flags
- They are both primitives[11] and equal according to the == operator
- They are both objects or arrays whose elements are deeply equal according to this definition

So, `assert.deepEqual()` will tell you if two values are 'equivalent', `assert.strictEqual()` will tell you if two values are the same primitive value or the same object, and `assert.equal()` will tell you if they're the same after type coercion. For example:

[8]http://nodejs.org/api/assert.html
[9]Briefly speaking, '===` returns `true` if both the operands are references to the same object or are primitives with the same type and value, while == *coerces* the operands to similar types before comparing them. So, `42 == "42"` is `true` but `42 === "42"` is `false`. == has many surprising behaviours, for example `0 == ""` and `[42] == 42` are `true`, but `[] == []` is `false`.
[10]This definition is necessary since this concept is not built into the language; in JavaScript `a == b` and `a === b` only tell you if `a` and `b` are the same object, not if they contain equivalent data. This is closer to how the == operator works in Java than in Python or Ruby.
[11]In JavaScript, the 'primitive' types are booleans, numbers and strings. All other types are 'reference' types.

Figure 2.9. Node assertion examples

```
// passes because the arrays contain the same data:
assert.deepEqual([1,2,3], [1,2,3])
// fails because an element is added:
assert.deepEqual([1,2,3], [1,2,3,4])
// fails because an element is changed
assert.deepEqual([1,5,3], [1,2,3])

var array = [1,2,3]

// passes because both values refer to the same object
assert.strictEqual(array, array)
// fails because the values refer to separate array instances
assert.strictEqual(array, [1,2,3])
// passes because strings are primitives and equal using ===
assert.strictEqual("123", "123")
// fails because 123 !== "123"
assert.strictEqual(123, "123")

// passes because 123 == "123"
assert.equal(123, "123")
```

2.4.2. Jasmine matchers

While Node uses simple assertion functions, Jasmine takes a more elaborate approach and uses matchers[12]. You'll see this style in a lot of frameworks but Jasmine is currently the most widespread example of it. It replaces the simple `assertEqual(a, b)` style with the more English-sounding `expect(a).toEqual(b)`. Here's the full list of Jasmine's equivalents for Node's equality assertions:

Figure 2.10. Correspondence between assertion functions and Jasmine matchers

```
// Node assertion function               Jasmine matcher

assert.strictEqual(actual, expected)       expect(actual).toBe(expected)
assert.notStrictEqual(actual, expected)    expect(actual).not.toBe(expected)

assert.deepEqual(actual, expected)         expect(actual).toEqual(expected)
assert.notDeepEqual(actual, expected)      expect(actual).not.toEqual(expected)

assert.equal(actual, expected)             // no analog
assert.notEqual(actual, expected)          // no analog
```

Since each framework must define this concept itself, Jasmine's notion of deep equality is slightly different to that in Node, but the important part is it offers recursive comparison of objects and arrays.

2.4.3. Chai assertion chains

Chai[13] is a popular assertion library that takes the matcher approach to extremes, allowing assertions to be chained.

[12]http://jasmine.github.io/2.4/introduction.html#section-Included_Matchers
[13]http://chaijs.com/

Figure 2.11. Correspondence between assertion functions and Chai chains

```
// Node assertion function                Chai chain

assert.strictEqual(actual, expected)      chai.expect(actual).to.equal(expected)
assert.notStrictEqual(actual, expected)   chai.expect(actual).to.not.equal(expected)

assert.deepEqual(actual, expected)        chai.expect(actual).to.deep.equal(expected)
assert.notDeepEqual(actual, expected)     chai.expect(actual).to.not.deep.equal(expected)

assert.equal(actual, expected)            // no analog
assert.notEqual(actual, expected)         // no analog
```

The Chai BDD interface goes well beyond this and allows chains like:

Figure 2.12. Complex Chai assertion chain

```
chai.expect(number).to.be.at.least(7).and.at.most(11)
```

This takes Jasmine's 'reads like English' design aesthetic even futher, going so far as to include code that does nothing other than make the test more sentence-like. Some tokens in Chai (like `to`, `be`, `and` and `at`) are do-nothing bits of syntactic sugar, while other tokens like `not` and `deep` are getters[14] that modify the rest of the chain. It can be tricky to remember which is which, and whether the order of them (e.g. `to.not` vs. `not.to`) is important. But you might find that this style's improvements in ease of reading are worth the added complexity — again, choosing which tool to use comes down to personal taste, there isn't one "right" answer.

2.4.4. QUnit assertions

Finally, back at the other end of the complexity spectrum, we have QUnit. It's a full framework rather than just an assertion library, but its API style is much more minimalist than Jasmine. A QUnit test suite is a collection of `test()` functions, and there are very few built-in assertions, which are top-level functions.

The example in `browser/qunit_assertions/equality_spec.js` shows the three flavours of equality assertion, which work the same as their Node counterparts.

Figure 2.13. `browser/qunit_assertions/equality_spec.js`

```
test("deep equality", function() {
  deepEqual( [1,2,3], [1,2,3] )
  notDeepEqual( [1,2,3], [1,2,3,4] )
  notDeepEqual( [1,5,3], [1,2,3] )
})

test("strict equality", function() {
  var array = [1,2,3]
  strictEqual( array, array )
  notStrictEqual( array, [1,2,3] )

  strictEqual( "123", "123" )
  notStrictEqual( 123, "123" )
})

test("equality", function() {
  equal( 123, "123" )
  notEqual( 42, "" )
})
```

[14] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty

2.4.5. `jstest` assertions and matchers

Now that we've seen various approaches to equality testing, let's look at the ones provided by the framework we'll actually be using[15]. `jstest` provides two main functions, `assertEqual()` and `assertSame()`, along with their negative counterparts, `assertNotEqual()` and `assertNotSame()`.

`assertSame()` and `assertNotSame()` are very simple: they compare their inputs using the strict `===` and `!==` operators, i.e. they pass if the values passed into them are the same primitive value or the same object.

`assertEqual()` is more complex. It implements deep equality on objects, arrays and dates, but not on platform-specific things like buffers, so it tells you if two values are 'equivalent'. However it also allows the values themselves to decide what equality means. When you call `assertEqual(expected, actual)`, it checks to see if `expected` has a method called `equals()`. If so, it invokes `expected.equals(actual)` and uses the result instead of its built-in equality algorithm. This method is also used when deep-comparing objects and arrays, so objects that contain objects that respond to `equals()` can be compared in this way.

This allows us to write 'matcher objects' whose `equals()` method can be used to test things. For example, whereas in Jasmine we would write

```
expect(["testing", "is", "easy"]).toContain(["easy"])
```

in `jstest` we would write this instead:

```
assertEqual( arrayIncluding("easy"), ["testing", "is", "easy"] )
```

`arrayIncluding()` is a function built into `jstest` that returns an object with a custom `equals()` method; in this case one that returns `true` if the array passed into it contains the required value.

This combination of a powerful equality assertion and matcher objects makes it easy to create new kinds of tests. For example, let's make a matcher that recognises people with a certain name:

Figure 2.14. `node/custom_matcher/test.js`

```
var JS = require("jstest")

var personNamed = function(name) {
  return {
    equals: function(value) {
      return (typeof value === "object") && (value.name === name)
    },
    toString: function() {
      return "Person named " + name
    }
  }
}

JS.Test.describe("Person", function() { with(this) {
  it("has a name", function() { with(this) {
    assertEqual( personNamed("Bond"), {name: "Bond"} )
  }})
}})

JS.Test.autorun()
```

The custom `toString()` method means we'll get helpful output when we break the test, for example by passing in `{name: "James"}`:

[15]All the `jstest` assertions are documented at http://jstest.jcoglan.com/assertions.html.

Figure 2.15. Custom matcher failure output

```
$ node node/custom_matcher/test.js
Loaded suite: Person

F

1) Failure: Person has a name
<Person named Bond> expected but was
<{ "name": "James" }>

Finished in 0.008 seconds
1 test, 1 assertion, 1 failure, 0 errors
```

This matcher capability is why `assertEqual()` takes the expected value *first*, and the actual value second. It's quite easy to remember that `assertEqual(a, b)` translates to `a.equals(b)` from the order of the arguments. And since `equals()` turns equality into a method rather than an operator, one of the operands gets to decide how it's implemented. It wouldn't make sense to ask the *actual* value to handle the `equals()` method, since that value came from the code we're trying to test — we don't know if its methods work properly, that's what the tests are supposed to prove. So, since the matcher is the one we should call `equals()` on, it goes on the left.

This concept of matcher objects will come up again when we take a look at stubbing methods in Section 2.6.2, "Pattern-matching arguments".

2.4.6. Other types of assertions

We've focussed very heavily on equality assertions here because they are the most commonly used, and every assertion library provides versions of them. Many frameworks and assertion libraries provide various other assertions for common tasks such as comparing numbers, matching DOM nodes, and testing data structure contents, and you can find them in the documentation for whichever tool you're using.

Whether to use functions or matchers[16] is largely a matter of personal taste. You should pick a style based on which you find easiest to read, and which is easiest write new assertions/matchers in when you need to. Testing is primarily about reducing risk, and one way to do this is by keeping your tests as simple and easy to read as possible. If the tests are hard to read and understand, then they are just as much of a liability as the code being tested, and they aren't helping you reduce risk at all.

I will briefly mention one further assertion that most libraries provide. It's unusual because instead of testing some property of a value, it tests a side effect of running some code. The assertion is called `assertThrow()`, and it fails unless the function you give it throws an error when run.

Figure 2.16. `browser/assert_throw/assert_throw_spec.js`

```
JS.Test.describe("assertThrow", function() { with(this) {
  it("makes sure an error is thrown", function() { with(this) {
    assertThrow(TypeError, function() { "spline".reticulate() })
  }})
}})
```

`assertThrow()` runs the given function and checks that it throws the given kind of error. In this case, trying to invoke a non-existent method on a string makes JavaScript throw a `TypeError`, as specified in the test. If the wrong type of error is thrown, or if no error is thrown at all, the test fails.

[16]You will often hear 'assertion functions' referred to simply as 'assertions'. Jasmine-style matchers are sometimes referred to as 'BDD style'.

There is a companion assertion to this called `assertNothingThrown()`, which takes a function and checks that running it does *not* throw an error. However, since a testing framework will catch any errors thrown by your tests anyway, this is not especially useful. It should only really be used in conjunction with `assertThrow()`, to call out that while one usage of an API should throw an error, another kind of usage should not. It signals explicitly to the reader that the code inside `assertNothingThrown()` is there to check it runs without errors, whereas just having the code in the test with no other assertions may confuse a reader and make them believe the test is not really testing anything.

For example, to complement the above test we might have one that shows no error is thrown when we call a valid method on a string.

Figure 2.17. `browser/assert_throw/assert_nothing_thrown_spec.js`

```
JS.Test.describe("assertNothingThrown", function() { with(this) {
  it("makes sure an error is not thrown", function() { with(this) {
    assertNothingThrown(function() { "spline".toUpperCase() })
  }})
}})
```

2.5. Organising tests

In Section 2.3, “Hello, world!”, we saw an example of a basic test using `jstest`:

```
JS.Test.describe("Hello, world!", function() { with(this) {
  it("runs a test", function() { with(this) {
    assertEqual( "Hello, world!", ["Hello", "world!"].join(", ") )
  }})
}})
```

Now, you might be thinking that’s a lot of boilerplate considering that the only line that really expresses anything important is the assertion:

```
assertEqual( "Hello, world!", ["Hello", "world!"].join(", ") )
```

What’s the purpose of the other lines, with the `describe()`, the `it()`, the anonymous functions and the `with(this)` blocks?

Well, let’s deal with `with(this)` first. `with` is a JavaScript feature that makes the properties of an object look like local variables, which in `jstest` we use to make it easier to access all the testing methods without prefixing them with ‘`this.`’. This is explained in more detail in Appendix B, *The ‘`with`’ statement*.

As we covered in Section 2.2, “Testing frameworks”, having one big script full of assertions doesn’t tend to scale well. It becomes hard to maintain and organise the tests, and they become harder to use: if you have one big script and an assertion half-way through throws an error, you can’t check any of the later assertions until that one is fixed. This becomes a problem if the tests for the component you’re working on appear after some tests that are broken for unrelated reasons. It’s useful to be able to isolate assertions from one another, and to put them into groups where the application is put into some known state before each test.

So, most frameworks have some mechanism for running subsets of the assertions, skipping all the other assertions in the suite. To do this, you must group your assertions into *tests*. In `jstest`, and other frameworks like Jasmine and Mocha, a test is represented by an `it()` block. Each test expresses some constraint about the program we’re testing. For example, if we were testing the `Array` class we might specify a few things about how its interface works:

Figure 2.18. *`node/array_tests/array_spec.js`*

```
JS.Test.describe("Array", function() { with(this) {
  it("has a length", function() { with(this) {
    assertEqual( 0, [].length )
    assertEqual( 3, [1, 2, 3].length )
  }})

  it("returns the value at an index", function() { with(this) {
    assertEqual( undefined, [][0] )
    assertEqual( "b", ["a", "b", "c"][1] )
  }})

  it("returns the index of a value", function() { with(this) {
    assertEqual( -1, [].indexOf("thing") )
    assertEqual( 2, ["this", "wooden", "idea"].indexOf("idea") )
  }})
}})
```

Separating the assertions into tests grouped by `it()` blocks means we can run subsets of them, which in `jstest` is done using an environment variable:

Figure 2.19. Using the `TEST` environment variable to select tests

```
$ FORMAT=tap node node/array_tests/test.js
1..3
ok 1 - Array has a length
ok 2 - Array returns the index of a value
ok 3 - Array returns the value at an index

$ TEST=length FORMAT=tap node node/array_tests/test.js
1..1
ok 1 - Array has a length

$ TEST=index FORMAT=tap node node/array_tests/test.js
1..2
ok 1 - Array returns the index of a value
ok 2 - Array returns the value at an index
```

Using the `TEST` environment variable, we can run only those tests whose name includes some specified string. This is useful for running only the tests for the component you're working on, without waiting for all the other tests to run or sifting through errors from other tests that don't immediately concern you.

The aim with this organisational approach is to keep groups of assertions focussed on one behaviour of the system: one method, or one subset of a method's behaviour. This provides granularity that makes it easy to run focussed sets of assertions when something is not working, and helps you narrow down what isn't working in the first place.

2.5.1. Setting up with `before()` blocks

A lot of tests require some setup; they require the system to be put into some known state before an interaction is tested. This can be as simple as creating an object in a known state before the start of each test, so each test can make some assertions about the object's behaviour. In `jstest` we use a `before()` block for this.

Figure 2.20. `node/array_scoped_tests/array_spec.js`

```
JS.Test.describe("Array", function() { with(this) {
  before(function() { with(this) {
    this.array = ["this", "wooden", "idea"]
  }})

  it("has a length", function() { with(this) {
    assertEqual( 3, array.length )
  }})

  it("returns the value at an index", function() { with(this) {
    assertEqual( "wooden", array[1] )
  }})

  it("returns the index of a value", function() { with(this) {
    assertEqual( 2, array.indexOf("idea") )
  }})
}})
```

You will also see `before()` referred to as `beforeEach()` or `setup()` in some frameworks. Also, some frameworks will encourage you to share state between the `before()` blocks and the `it()` blocks by using a local variable rather than assigning the state to `this`. For example, Jasmine and Mocha ask you to write tests like this:

Figure 2.21. `browser/jasmine_before_each/array_spec.js`

```
describe("Array", function() {
  var array

  beforeEach(function() {
    array = ["this", "wooden", "idea"]
  })

  it("has a length", function() {
    expect(array.length).toBe(3)
  })

  it("returns the value at an index", function() {
    expect(array[1]).toBe("wooden")
  })

  it("returns the index of a value", function() {
    expect(array.indexOf("idea")).toBe(2)
  })
})
```

Using a local variable that's visible by all the enclosed `before()` and `it()` blocks is usually fine, but it can cause to state to leak between tests — as far as the code that uses this variable is concerned, it's essentially a global and causes the same sorts of problems.

However, storing state on `this` only solves this problem if the testing framework gives you a new object bound to `this` for each test, so the state you create is really isolated to that test. `jstest` and Jasmine provide a new *context* object for each test, but Mocha does not.

2.5.2. Wrapping up with `after()` blocks

Just as tests often need some state to be set up beforehand, many tests need to destroy that state after they've finished, to leave a clean slate for the next test to run. This will usually be the case if your tests create any global state beyond just making some variables that are scoped to the test. If they put HTML into the DOM, leave data in `localStorage`, or start up a web server, that state needs to be removed at the end of the test: the DOM must have fixture data removed, `localStorage` must be emptied, or the server must be shut down.

For this purpose, we have `after()` blocks, sometimes called `afterEach()` or `teardown()` in other frameworks. Just like `before()` blocks, an `after()` block runs once for each test, but at the end of the test instead of the beginning. In general, the framework runs the tests by looping through all the `it()` blocks, and running the `before()` block, the `it()` block and the `after()` block in that order.

Figure 2.22. `browser/after_blocks/dom_spec.js`

```
JS.Test.describe("DOM fixtures", function() { with(this) {
  before(function() { with(this) {
    this.fixture = $("#fixture")
    fixture.append('<p class="message">Hello</p>')
  }})

  after(function() { with(this) {
    fixture.empty()
  }})

  it("creates a paragraph saying 'Hello'", function() { with(this) {
    assertEqual( "Hello", fixture.find("p").text() )
  }})

  it("creates a paragraph with a 'message' class", function() { with(this) {
    assert( fixture.find("p").hasClass("message") )
  }})

  it("creates exactly one paragraph", function() { with(this) {
    assertEqual( 1, fixture.find("p").length )
  }})
}})
```

We clean up after our tests because we want to make sure there's no state lingering between tests that could make them interfere with one another. In the above example, we use `fixture.empty()` to remove the fixture HTML we added in the `before()` block; if we did not then the final test would see three paragraph elements instead of one, since another paragraph is added by the `before()` block before each test.

In the browser, we want to make sure all DOM nodes and event handlers are removed between tests, or handlers from one test might attach to DOM nodes in another. When running database-backed tests, we want to make sure data saved during one test does not affect the outcome of another, so we delete it all. If we leave a web server running, any subsequent test will not be able to use the port when it tries to run the server.

You should write tests such that they can be run in any order. If they are order-dependent, it means there is state leaking between them that will mislead you about what's really going on, about whether your code really works. It also makes it impossible to run tests individually, making it harder to debug problems as you maintain the codebase.

2.5.3. Contexts with `describe()` blocks

Many frameworks allow you to nest `describe()` blocks. When you have nested `describe()` blocks, the framework runs *all* the `before()` blocks from any `describe()` blocks that enclose the current `it()` block, then the `it()` itself, followed by all the `after()` blocks from enclosing `describe()` blocks.

This tends to get used in one of two ways. The first is as a purely organisational tool, where we set up an object with a single `before()`, and group the tests of the object's methods into one `describe()` block per method. For example, we might test some jQuery methods like this:

Figure 2.23. `browser/jquery_tests/jquery_spec.js`

```
JS.Test.describe("jQuery", function() { with(this) {
  before(function() { with(this) {
    this.fixture = $("#fixture")
    fixture.html('<p class="message">Hello</p>')
    this.p = fixture.find("p")
  }})

  after(function() { with(this) {
    fixture.empty()
  }})

  describe("#hasClass", function() { with(this) {
    it("returns true for existing classes", function() { with(this) {
      assert( p.hasClass("message") )
    }})

    it("returns false for non-existent classes", function() { with(this) {
      assertNot( p.hasClass("error") )
    }})
  }})

  describe("#text", function() { with(this) {
    it("returns the node's inner text", function() { with(this) {
      assertEqual( "Hello", p.text() )
    }})
  }})
}})
```

In this situation, the inner `describe()` blocks have no *technical* purpose: you could remove them and the `it()` blocks inside would work just the same. They're just being used to group related tests so that we can run all the tests for one method without having to write its name out for every test.

But nested `describe()` blocks really come into their own when they contain setup in a `before()` block. The `before()` block lets you modify the state created in the outer `before()`, *only* for the tests inside the inner `describe()`.

In the previous example we wrote two tests for `jQuery.hasClass()`, based on checking two different class names: one that the paragraph has and one that it does not. But there's another way to express that: call `jQuery.hasClass()` with the *same* class name, once in a context where the class exists and once where it does not. Rather than assert that one thing is true and another thing false, we assert that the same thing can be true or false depending on context.

To do this, we write two `describe()` blocks that express the two contexts. Each one has a name that explains the context, and a `before()` block whose code implements the context by making some state changes. When the tests run, `jstest` runs the outer `before()` block to set up some general state, then the inner `before()` block that makes the contextual change, then runs the `it()` block.

Figure 2.24. `browser/jquery_tests/context_spec.js`

```
JS.Test.describe("jQuery", function() { with(this) {
  before(function() { with(this) {
    this.fixture = $("#fixture")
    fixture.html('<p>Hello</p>')
    this.p = fixture.find("p")
  }})

  after(function() { with(this) {
    fixture.empty()
  }})

  describe("with a 'message' class", function() { with(this) {
    before(function() { with(this) {
      p.addClass("message")
    }})

    it("returns true for existing classes", function() { with(this) {
      assert( p.hasClass("message") )
    }})
  }})

  describe("without a 'message' class", function() { with(this) {
    before(function() { with(this) {
      // no-op
    }})

    it("returns false for non-existent classes", function() { with(this) {
      assertNot( p.hasClass("message") )
    }})
  }})
}})
```

So nested `describe()` blocks let you set up tests with different combinations of states. This can be very useful for testing programs whose behaviour depends on some system state, like the records in a database, or the state of the DOM. But it can be easily abused; if you nest too deep or there are many tests inside each `describe()` block, it becomes hard to read the tests and reason about the state within each one. It can be tempting to follow the DRY principle[17] and move any form of duplicated setup between tests into increasingly elaborate `before()` blocks, but this tends to create coupling and complexity and makes the tests harder to understand and refactor.

Remember: it's only worth having tests if they're significantly less complex than your code, so resist the urge to excessively reuse and couple code in tests. Repeating yourself a bit can be great if it means everything you need to understand what a test does is right there with the tests, rather than scattered across a lot of setup functions.

2.6. Stubs, mocks and spies

Aside from assertions and an organisational model, one of the most important tools a testing framework gives you is the ability to replace parts of your application and its environment with fakes. In many situations you don't want to test against the real implementation of something; for example, if you're trying to test some front-end code that uses `jQuery.get()` to talk to the server, you might want to fake out the server's response during the test rather than run your real application server with all its attendant complexity. Anything that keeps your JS tests based on static HTML and JavaScript files is often a win.

[17]http://en.wikipedia.org/wiki/Don't_repeat_yourself

Fortunately, JavaScript being the incredibly dynamic language it is, replacing the real functionality with a fake version is quite easy. Imagine we have some code that makes it so that when a certain link is clicked, we fetch some HTML from the server and display it:

Figure 2.25. `browser/hello_stubs/hijack_links.js`

```
var hijackLink = function(selector, target) {
  $(selector).on("click", function() {
    var url = $(this).attr("href")
    $.get(url, function(response) {
      $(target).html(response)
    })
    return false
  })
}
```

We would like to test this function, but we don't want to spin up a server for it to talk to. That's fine, since JavaScript lets you set any property you like on most objects, we can just tack a fake implementation of `get()` onto the `jQuery` object that does what we want. Here's a test that sets up some fixture HTML, creates a fake `jQuery.get()` function that returns a canned response, and then exercises the above function by using the Syn[18] library to simulate a user's click.

Figure 2.26. `browser/hello_stubs/manual_stub_spec.js`

```
JS.Test.describe("Manual stubbing", function() { with(this) {
  before(function() { with(this) {
    this.fixture = $("#fixture")
    fixture.html('  <p></p>\
                    <a href="/">Home</a>')

    this.jqueryGet = $.get
    $.get = function(url, callback) {
      callback("Hello from the server")
    }

    hijackLink("#fixture a", "#fixture p")
  }})

  after(function() { with(this) {
    fixture.empty()
    $.get = jqueryGet
  }})

  describe("when the link is clicked", function() { with(this) {
    before(function(resume) { with(this) {
      syn.click(fixture.find("a"), resume)
    }})

    it("displays the server response", function() { with(this) {
      assertEqual( "Hello from the server", fixture.find("p").text() )
    }})
  }})
}})
```

This works fine, but there's a problem. Since we only want `jQuery.get()` to be faked out for the duration of this test, we need to make sure that the fake version we create doesn't hang around after each test is finished. We're making changes to a globally accessible library API, and we'd better clean up after ourselves! So, we need to hang on to the *original* `jQuery.get()` function, which we store

[18]https://github.com/bitovi/syn

in `this.jqueryGet`. Then, in an `after()` block, we restore `jQuery.get()` to the real version that we stashed away before the test.

As you start to fake more stuff out, managing these changes becomes harder. And quite often, you're probably going to want a fake function to respond in different ways depending on the arguments you pass in; in this case different URLs should give different responses. You can do this with an `if` statement in your fake function, but again that becomes tedious to write and hard to manage.

2.6.1. Enter stubs

The act of replacing real functions with fakes that return canned responses is called stubbing, and many testing frameworks provide a mechanism for doing this that includes tracking which things have been faked out and restoring the real versions easily, and providing different output based on the arguments.

`jstest` has a system for this[19], and it lets you replace all this code:

Figure 2.27. Manual function stubbing

```
before(function() { with(this) {
  this.jqueryGet = $.get
  $.get = function(url, callback) {
    callback("Hello from the server")
  }
}})

after(function() { with(this) {
  $.get = jqueryGet
}})
```

with a one-liner before-hook:

Figure 2.28. Automated function stubbing with `stub()`

```
before(function() { with(this) {
  stub($, "get").yields(["Hello from the server"])
}})
```

This line says: when the function `$.get()` is called, and its last argument is a callback function, then invoke the callback function with the argument list `["Hello from the server"]`. That is, `yields()` makes the fake invoke its callback with the given array mapped onto the callback's arguments.

`jstest` implicitly resets all registered stubs at the end of every test, so you don't need an `after()` block for this. If you're using a third-party stubbing library you might need to explicitly tell it to restore the real functions after a test.

This change lets us slim down our test suite a little bit:

[19]The `jstest` stubbing API is documented at http://jstest.jcoglan.com/mocking.html

Figure 2.29. `browser/hello_stubs/auto_stub_spec.js`

```
JS.Test.describe("Automated stubbing", function() { with(this) {
  before(function() { with(this) {
    this.fixture = $("#fixture")
    fixture.html('  <p></p>\
                    <a href="/">Home</a>')

    stub($, "get").yields(["Hello from the server"])

    hijackLink("#fixture a", "#fixture p")
  }})

  after(function() { with(this) {
    fixture.empty()
  }})

  describe("when the link is clicked", function() { with(this) {
    before(function(resume) { with(this) {
      syn.click(fixture.find("a"), resume)
    }})

    it("displays the server response", function() { with(this) {
      assertEqual( "Hello from the server", fixture.find("p").text() )
    }})
  }})
}})
```

For functions that use `return` statements instead of a callback, we use the `returns()` modifier. For example, jQuery has an alternative Ajax API where, instead of taking a callback, the `$.get()` function returns a *promise*, which has a `then()` method that you can use to register callbacks. We call it like this:

```
$.get(url).then(function(response) {
  // ...
})
```

If we want to stub a function's return value, then we use the `returns()` modifier. In this case we could make `$.get()` return a pre-prepared `jQuery.Deferred` object containing the response data:

Figure 2.30. Returning a canned promise from a stub

```
var xhr = new $.Deferred()
xhr.resolve("Hello from the server")
stub($, "get").returns(xhr.promise())
```

2.6.2. Pattern-matching arguments

Rather than hand-writing fake functions with `if` statements to choose a response based on the arguments passed in, it's easier to specify these responses declaratively. So, `jstest` includes another modifier to its `stub()` function, called `given()`, which lets you pattern-match on arguments. As a simple example, we could make `jQuery.get()` yield different responses depending on the URL:

Figure 2.31. Yielding canned responses for different URLs

```
stub($, "get").given("/").yields(["Homepage HTML"])
stub($, "get").given("/users/18787.json").yields(['{"username":"jcoglan"}'])
```

The first example says that if you call `$.get("/", function(r) { … })` then the callback function will be invoked with `r` set to `"Homepage HTML"`. If we were writing stubs for a function that returns something, we could write:

Figure 2.32. Returning canned addition results

```
stub("add").given(2, 2).returns(4)
```

This creates a global function called `add()` that returns `4` when called with `2` and `2`. If you call the function with any other arguments, an error is thrown:

Figure 2.33. Calling a stub with unexpected arguments

```
> add(2, 2)
4
> add(2, 2, 0)
Error: <[object Window]> unexpectedly received call to add() with arguments: ( 2, 2, 0 )
```

When you use `yields()`, the stub implicitly expects a function as an argument after those you specify in `given()`, that is `given()` only matches arguments up to the callback. If you don't use `given()`, then the stub will return/yield the specified value no matter what arguments you pass to it.

Finally, just as `assertEqual()` uses `equals()` on its arguments to determine equality, so does the stub function dispatcher. When you call a stubbed function, `jstest` compares all the argument lists from `given()` with the arguments you actually passed to pick which return value to use. If the `given()` arguments have an `equals()` method, that is used to match the incoming argument. `jstest` includes a few built-in matchers for common cases, for example in Section 2.4.5, "`jstest` assertions and matchers" we came across `arrayIncluding()`, which matches an array containing the specified items:

Figure 2.34. Matching a stub call with `arrayIncluding()`

```
> stub("valid").given(arrayIncluding("hello", "world")).returns(true)
> valid(["hello", "to", "the", "world"])
true
> valid(["hi"])
Error: <[object Window]> unexpectedly received call to valid() with arguments: ( [ "hi" ] )
```

There are several other useful matchers:

- `anything()` matches absolutely any value
- `instanceOf(type)` matches any value for which `typeof value === type`, or `value instanceof type` is `true`, for example `instanceOf(Array)` would match `[]`, and `instanceOf("string")` would match `"hello"`
- `arrayIncluding(items)` matches any array containing all the given `items`
- `objectIncluding(properties)` matches any object containing all the given `properties`
- `match(pattern)` matches any string matching the regex `pattern`, or, if `pattern` has a `match()` method, any value for which that method returns `true`

These matchers can be composed, for example

```
arrayIncluding(objectIncluding({name: match(/cog/)}))
```

would match the value `["hello", {name: "jcoglan", city: "London"}]`.

Finally, there is one special matcher that means 'any further arguments'. It's used to say that the function may be called with any arguments after those that are explicitly stated.

Figure 2.35. Using the `anyArgs()` matcher

```
> stub("add").given(1, 2, anyArgs()).returns(3)
> add(1, 2)
3
> add(1, 2, 4, 8, 16)
3
> add(1, 4)
Error: <[object Window]> unexpectedly received call to add() with arguments: ( 1, 4 )
```

As shown in Figure 2.14, “`node/custom_matcher/test.js`”, you can create your own matcher functions by returning an object with an `equals()` method to match incoming arguments.

2.6.3. Testing interactions with mocks

Mocks are like stubs with one extra feature: if the stubbed function isn’t called with the expected arguments during the test, an error is thrown. When you stub out a function, you’re saying, “given this stubbed function is assumed to respond like *so*, how should the function I’m testing behave?” Whereas when you mock a function, you’re saying “when I call the function I’m testing, which other functions should it call?”

Put another way, mocks are like assertions, except rather than check the return/callback output of a function, they check the function’s side effects on other parts of the system.

For example, say we have a function that helps us write debug messages to the console, and all it does is prepend the current timestamp to the message before logging it:

Figure 2.36. *`browser/hello_mocks/debug.js`*

```
var debug = function(message) {
  console.info(new Date().getTime() + ' ' + message)
}
```

We can’t write a test by running this function, and then reading from the console — there’s no browser API for that. But notice that this function’s output is a *side effect*: it doesn’t return anything, but it tells another object to do something using a method call. This is a perfect candidate for mock-based testing.

Figure 2.37. *`browser/hello_mocks/debug_mock_spec.js`*

```
JS.Test.describe("debug() with mocks", function() { with(this) {
  it("writes the message to the console with a timestamp", function() { with(this) {
    expect(console, "info").given(match(/^[0-9]{13} hello$/))
    debug("hello")
  }})
}})
```

The mocking API is exactly like stubbing, except we replace `stub()` with `expect()`. Notice that the mocking is done *before* we call the function we’re testing, whereas assertions are usually done afterward. This is because the mocked function must be set up to record calls before any of those calls happen, while assertions are checks on what comes out of a function after we call it.

Now, if you remove the `console.info()` call from `debug.js` then you should see the test fail with this error:

```
<{ "info": #function }> expected to receive call
info( match(/^[0-9]{13} hello$/) )
```

When you use `expect()`, the `given()` modifier says that the function *must* be called with those arguments. If you leave the `given()` modifier out, then the function can be called with any arguments to make the test pass. And just like with stubbing, you can use the `returns()` and `yields()` modifiers to make the function return a canned response if the function under test expects one; if you don’t specify a return value the function will return `undefined`.

Mocking is most useful for checking that a function tells some other object to do something, or that it sends some data somewhere. Checking that a form posts some data to a server, or that a server writes something to a database or logging system, or that an interaction with a UI component triggers some behaviour in another, are all good candidates for mock-based testing. It lets you check that two components interact in the right way without needing the secondary component to be present and running during the test.

It is easy to get tricked into testing internal implementation details when mocking, so don't assume you should test *all* the internal function calls in your system. Mocking should only be used where you have a well-defined API boundary between two things, where that API is reasonably stable, and where you understand the API being mocked very well, so you know exactly how it should be used.

A good example is a web app that supports pluggable storage backends: people can supply an object that implements some standard interface and the server talks to that *adapter*, instead of directly to the filesystem or database. In this situation it's useful to test the server's interaction with the *abstract interface*, so you know that whatever adapter people supply will be called correctly. For example, you'd test that when the server receives an HTTP `POST` request, it calls `save()` on the adapter with the data from the request body. Mocking is great for testing these integration contract scenarios.

2.6.4. Checking your wiring with spies

The final common tool for manipulating and inspecting functions during tests is called *spying*. Like mocking, spying lets you check that certain function calls were made during a test, but unlike mocking, it leaves the original functions intact instead of stubbing them out.

`jstest` doesn't have a spying API, but there's a great library for this called Sinon[20]. Sinon has a different API style: instead of setting up the call expectations before running the code, we set up a spy, then run some code, then inspect or assert what the function was called with afterward. We also need to explicitly restore the spied functions after the test, so I've separated the setup and teardown of the spy into before/after hooks.

Figure 2.38. *`browser/hello_mocks/debug_spy_spec.js`*

```
JS.Test.describe("debug() with spies", function() { with(this) {
  before(function() { with(this) {
    sinon.spy(console, "info")
  }})

  after(function() { with(this) {
    console.info.restore()
  }})

  it("writes the message to the console with a timestamp", function() { with(this) {
    debug("hello")
    sinon.assert.calledWithMatch(console.info, /^[0-9]{13} hello$/)
  }})
}})
```

Notice how when you run this test, the real `console.info()` is still invoked and we see output in the terminal. This differs from mocking, where the mocked function is replaced with a do-nothing fake. Whether to invoke the real functionality or fake it out when testing interactions is a judgement call in each scenario, and it comes down to risk management. If the thing you're mocking isn't well-covered by its own tests, or the API contract isn't well defined, then there's a case against faking it out until it's stable and you're sure how it behaves having tested it. Having fakes that don't behave like the real thing is a risk, and in some cases that risk outweighs the benefit of isolating the thing you're testing.

A full discussion of the Sinon API is outside the scope of this book, but its style of verifying interactions makes it worth checking out.

2.7. Asynchronous tests

JavaScript being what it is, you will often need to test asynchronous APIs. In Section 2.6, "Stubs, mocks and spies", we used a fake version of `jQuery.get()` that invoked its callback synchronously, and this

[20]http://sinonjs.org/

is one way of removing asynchrony from your code to make it easier to test. But oftentimes you'll run into situations where you can't avoid async code and it needs to be tested.

Most testing frameworks provide a mechanism for doing this, and in `jstest` it's done by adding a parameter to your `before()`, `after()` and `it()` blocks. The parameter is a function that you must call when all the asynchronous tasks have completed, and the test should continue.

Let's look at an example. Here's a class whose only function is to increment an integer asynchronously:

Figure 2.39. `browser/hello_async/counter.js`

```
var Counter = function() {
  this.count = 0
}

Counter.prototype.inc = function(n, callback) {
  var self = this
  setTimeout(function() {
    self.count += n
    callback()
  }, 10)
}
```

We can test it like this — notice the `resume` parameter to the `it()` block, which tells the framework the test is asynchronous so it should wait for `resume()` to be called.

Figure 2.40. `browser/hello_async/counter_spec.js`

```
JS.Test.describe("Counter", function() { with(this) {
  before(function() { with(this) {
    this.counter = new Counter()
  }})

  it("increments the count", function(resume) { with(this) {
    counter.inc(5, function() {
      resume(function() {
        assertEqual( 5, counter.count )
      })
    })
  }})
}})
```

Notice how we pass *another* function containing the assertions to `resume()`. We could instead write the test like this, with the assertion preceeding the `resume()` call:

Figure 2.41. `browser/hello_async/async_error_spec.js`

```
JS.Test.describe("Counter with async assertion", function() { with(this) {
  before(function() { with(this) {
    this.counter = new Counter()
  }})

  it("increments the count", function(resume) { with(this) {
    counter.inc(5, function() {
      assertEqual( 5, counter.count )
      resume()
    })
  }})
}})
```

But, since the callback to `counter.inc()` is run asynchronously, the framework can't catch the potential error thrown by `assertEqual()` using `try/catch` and must instead catch it using `window.onerror` (or `process.on("uncaughtException")` on Node). This means some information about the error, for example its type and its stack trace, will usually be lost. If we run the assertions in a function passed to `resume()`, the framework can run that function synchronously and catch any exceptions in the normal way.

So, the latter will work, but the former will give you better failure messages. Of course, errors thrown in async code are sometimes unavoidable. Some frameworks do not catch and report async errors, and can mislead you about what's wrong with your program. Whichever framework you use, check whether it does this so you're not caught out[21]. If your framework doesn't catch async errors, it can cause the framework to hang due to a required callback not being invoked, or the framework will time your tests out.

The `resume()` function also serves the important task of making sure the assertions are run at all. Consider what would happen if the test was written like this:

Figure 2.42. An async test that does not check its callback

```
it("increments the count", function() { with(this) {
  counter.inc(5, function() {
    assertEqual( 5, counter.count )
  })
}})
```

It is possible for this test to pass while the code is broken: if `counter.inc()` does not invoke its callback at all, the assertion will not be run and the test will pass. It's important that we make sure our tests do not admit code that appears to work by accident, and so our strategy for testing async APIs *must* check that any callbacks used in the test are actually invoked. `resume()` gives us a way to do this; if you add the `resume` parameter to your test, `jstest` will wait for `resume()` to be invoked before continuing. If `resume()` is never called, the test will time out and `jstest` will report this error to us.

2.7.1. The error-first callback convention

The other thing you need to know about `resume()` is that it obeys the 'error-first' callback convention. In Node, it is conventional for any function that takes a callback to invoke this callback with an error as the first argument, if one occurs. This acts as a reminder to the user that they should handle the error, and allows for control-flow abstractions based on the assumption that errors will always be surfaced in this way. For example, the Node API for reading a file is:

Figure 2.43. The error-first callback convention

```
var fs = require("fs")

fs.readFile("path/to/file.txt", function(error, contents) {
  // ...
})
```

If the file was read successfully, then `error` will be `null` or `undefined`, and `contents` will be a `Buffer`. But if there was an error, such as the file not existing or the current process not having read access to it, then `error` will be an `Error` object such as you'd see in a `catch` statement.

The `jstest resume()` API integrates with this convention; if the first argument passed to `resume()` is a string or an object then `jstest` will treat this as an error and report it. This means we can write async setup steps using minimal code, while letting the framework detect any errors that happen. For example, this `before()` block will report an error if the file could not be written:

Figure 2.44. Catching errors using the `resume()` function

```
before(function(resume) { with(this) {
  fs.writeFile("config/settings.json", JSON.stringify(config), resume)
}})
```

[21]At time of writing, Mocha, QUnit and `jstest` catch async errors, Jasmine doesn't catch them but makes the test time out. See the examples in `browser/async_comparison` for details.

2.7.2. Multi-step async tests

In more complex integration tests, it’s easy to end up with the dreaded ‘pyramid of doom’ of callbacks within callbacks. It’s paramount that your tests remain as readable as possible so that they’re easy to understand, and async tests quickly become clogged with syntactic noise if you’re not careful. `jstest` provides a tool called asynchronous stories[22] to deal with this, which lets you write a series of functions describing the steps in a test and then wire those functions together without having any callbacks in the tests themselves.

For example, imagine we have a little app built on the Express[23] web framework, that uses the `GET` and `PUT` HTTP methods to let you read and write an in-memory storage object.

Figure 2.45. `node/async_steps/app.js`

```
var connect = require("connect"),
    express = require("express"),
    app     = express(),
    storage = {}

app.use(connect.urlencoded())

app.get("/:key", function(request, response) {
  var key = request.params.key
  if (storage.hasOwnProperty(key)) {
    response.send(200, storage[key])
  } else {
    response.send(404, "Not found")
  }
})

app.put("/:key", function(request, response) {
  storage[request.params.key] = request.body.value
  response.send(201, "Created")
})

app.delete("/", function(request, response) {
  storage = {}
  response.send(200, "OK")
})

module.exports = app
```

We can store data in this server using `PUT` and retrieve it with `GET`, and we can clear the stored data with `DELETE`:

Figure 2.46. Using `curl` to interact with an Express server

```
$ curl -X GET http://localhost:8000/meaning_of_life
Not found
$ curl -X PUT http://localhost:8000/meaning_of_life -d "value=42"
Created
$ curl -X GET http://localhost:8000/meaning_of_life
42
$ curl -X DELETE http://localhost:8000/
OK
$ curl -X GET http://localhost:8000/meaning_of_life
Not found
```

[22]http://jstest.jcoglan.com/async.html
[23]http://expressjs.com/

We'd like to reproduce this manual checking we've done with `curl` using an automated JavaScript test. This test would involve starting a server, sending a `PUT` to store some data, sending a `GET` to retrieve the data, checking the response is the same as the data you put in, before clearing the server's data and shutting it down. Most of those steps involve calling functions with callbacks or listening for events, and even a simple test can get very messy, even if we use a library like Request[24] to make doing the HTTP bits easier.

So, rather than write one big `it()` block with all that logic, especially when many of those steps will be reused across a test suite, it's helpful to write a few helper functions to encapsulate the steps.

Figure 2.47. *`node/async_steps/server_steps.js`*

```
var request = require("request"),
    app     = require("./app")

var ServerSteps = {
  startServer: function(port, callback) {
    this.host   = "http://localhost:" + port
    this.server = app.listen(port, callback)
  },

  stopServer: function(callback) {
    this.server.close(callback)
  },

  clearData: function(callback) {
    request.del(this.host + "/", callback)
  },

  get: function(path, callback) {
    var self = this
    request(this.host + path, function(error, response, body) {
      self.response = response
      self.response.body = body
      callback()
    })
  },

  put: function(path, params, callback) {
    request.put(this.host + path, {form: params}, callback)
  },

  checkBody: function(body, callback) {
    this.assertEqual(body, this.response.body)
    callback()
  }
}

module.exports = ServerSteps
```

With these helper functions defined, we can write some tests that use them to exercise the server. The `startServer()`, `stopServer()` and other helper functions are made available in the tests by the `include(steps)` directive, which imports a set of methods into the current context.

When the step functions are invoked, `this` refers to the currently executing test, so we can access all the `jstest` assertion methods through it.

[24]https://npmjs.org/package/request

Figure 2.48. `node/async_steps/async_callbacks_spec.js`

```
var JS    = require("jstest"),
    steps = require("./server_steps")

JS.Test.describe("Storage server (callbacks)", function() { with(this) {
  include(steps)

  before(function(resume) { with(this) {
    startServer(4180, resume)
  }})

  after(function(resume) { with(this) {
    clearData(function() {
      stopServer(resume)
    })
  }})

  it("saves and retrieves a value", function(resume) { with(this) {
    put("/meaning_of_life", {value: "42"}, function() {
      get("/meaning_of_life", function() {
        resume(function(resume) {
          checkBody("42", resume)
        })
      })
    })
  }})

  it("deletes all the data", function(resume) { with(this) {
    put("/meaning_of_life", {value: "42"}, function() {
      clearData(function() {
        get("/meaning_of_life", function() {
          resume(function(resume) {
            checkBody("Not found", resume)
          })
        })
      })
    })
  }})
}})
```

This is fine, the tests do their job, but despite us wrapping most of the logic in a high-level API they're still suffering from a touch of callbackitis. But we can fix that in a number of ways.

2.7.3. Sequencing functions with Async

Async[25] is a library for composing async functions, on the assumption that those functions use Node-style `function(error, result)` callbacks. We can use it to run a series of async functions in a test while keeping the code flat and easier to follow.

Here's our original test suite rewritten using `async.series()` to flatten the code. `async.series()` takes a list of functions and executes them sequentially. It does this by passing a callback to each of the functions and waiting for it to be called before running the next function in the list.

[25]https://npmjs.org/package/async

Figure 2.49. `node/async_steps/async_series_spec.js`

```
var JS    = require("jstest"),
    async = require("async"),
    steps = require("./server_steps")

JS.Test.describe("Storage server (async.series)", function() { with(this) {
  include(steps)

  before(function(resume) { with(this) {
    async.series([
      function(cb) { startServer(4180, cb) }
    ], resume)
  }})

  after(function(resume) { with(this) {
    async.series([
      function(cb) { clearData(cb) },
      function(cb) { stopServer(cb) }
    ], resume)
  }})

  it("saves and retrieves a value", function(resume) { with(this) {
    async.series([
      function(cb) { put("/meaning_of_life", {value: "42"}, cb) },
      function(cb) { get("/meaning_of_life", cb) },
      function(cb) { checkBody("42", cb) }
    ], resume)
  }})

  it("deletes all the data", function(resume) { with(this) {
    async.series([
      function(cb) { put("/meaning_of_life", {value: "42"}, cb) },
      function(cb) { clearData(cb) },
      function(cb) { get("/meaning_of_life", cb) },
      function(cb) { checkBody("Not found", cb) }
    ], resume)
  }})
}})
```

This has reduced some of the line noise of the original test, flattening the code out and visually separating the callback functions from the test logic. There's still quite a lot of boilerplate though, and we can go one step better.

2.7.4. Generating steps with continuables

A 'continuable' is a specially refactored type of async function that makes it a bit easier to compose and sequence things. Specifically, it involves currying[26] the function so that instead of taking all the arguments followed by a callback, it takes the arguments and *returns* another function that takes the callback. So, to take one of the helper functions as an example, we'd replace

Figure 2.50. A callback-taking function

```
  startServer: function(port, callback) {
    this.host   = "http://localhost:" + port
    this.server = app.listen(port, callback)
  }
```

with this:

[26]See Appendix C, *Currying*.

Figure 2.51. A continuable function

```
    startServer: function(port) {
      var self = this
      return function(callback) {
        self.host   = "http://localhost:" + port
        self.server = app.listen(port, callback)
      }
    }
```

Notice what we’ve done here: `async.series()` takes a list of functions that each take a callback. We’ve made it so that when we invoke our helper functions, we get back a function that takes a callback and does the actual work, which is exactly what we can feed into `async.series()`. The currying conversion can be done automatically with some metaprogramming[27], rather than rewriting all the functions by hand, and lets us rewrite the test as follows:

Figure 2.52. `node/async_steps/async_continuables_spec.js`

```
var JS    = require("jstest"),
    async = require("async"),
    steps = require("./server_steps"),
    curry = require("./curry")

steps = curry.object(steps)

JS.Test.describe("Storage server (continuables)", function() { with(this) {
  include(steps)

  before(function(resume) { with(this) {
    async.series([
      startServer(4180)
    ], resume)
  }})

  after(function(resume) { with(this) {
    async.series([
      clearData(),
      stopServer()
    ], resume)
  }})

  it("saves and retrieves a value", function(resume) { with(this) {
    async.series([
      put("/meaning_of_life", {value: "42"}),
      get("/meaning_of_life"),
      checkBody("42")
    ], resume)
  }})

  it("deletes all the data", function(resume) { with(this) {
    async.series([
      put("/meaning_of_life", {value: "42"}),
      clearData(),
      get("/meaning_of_life"),
      checkBody("Not found")
    ], resume)
  }})
}})
```

So that’s how you can get very readable step sequences using general-purpose libraries. But there’s one final method that can remove even more boilerplate.

[27]See Figure C.1, “`node/async_steps/curry.js`”.

2.7.5. Async stories with `jstest`

`jstest` has a built-in async step sequencer called *async stories*. By wrapping our helper module with a call to `JS.Test.asyncSteps()`, we can convert its methods into simpler-looking ones that don't need callbacks[28]. You can write your tests as a flat list of steps, without the nesting and line noise, for example:

Figure 2.53. *`node/async_steps/async_steps_spec.js`*

```
var JS    = require("jstest"),
    steps = require("./server_steps")

steps = JS.Test.asyncSteps(steps)

JS.Test.describe("Storage server (async steps)", function() { with(this) {
  include(steps)

  before(function() { with(this) {
    startServer(4180)
  }})

  after(function() { with(this) {
    clearData()
    stopServer()
  }})

  it("saves and retrieves a value", function() { with(this) {
    put("/meaning_of_life", {value: "42"})
    get("/meaning_of_life")
    checkBody("42")
  }})

  it("deletes all the data", function() { with(this) {
    put("/meaning_of_life", {value: "42"})
    clearData()
    get("/meaning_of_life")
    checkBody("Not found")
  }})
}})
```

This test suite is very high-level and easy to read: it's just a flat list of function calls describing the steps we want to take. As your test suite grows, maintaining this level of readability in async tests can be really beneficial; for example it's much easier to reorder the steps you run just by moving lines up and down, without dealing with the nesting problems of callbacks.

This technique can keep async tests very readable, but can be a bit tricky to use, since due to the 'magic' involved in providing the callback-less API you cannot put any logic in the tests themselves — all logic must be inside the step functions to make sure it's called in the right order.

Since it's deeply integrated with the test runner, `JS.Test.asyncSteps()` can eliminate even more boilerplate (including the `resume` callbacks) than general-purpose libraries can, and I'll be using it throughout the rest of the book. But remember: you don't have to use my pet tools to write code this way, you can do it in any framework that supports async testing.

[28]Technically, calling the callback-less functions in the test has the effect of putting all the functions into a queue, which `jstest` then executes sequentially by adding callbacks where necessary.

3. Events and streams

It’s often said that JavaScript is an ‘event-driven’ language. While not strictly true — there’s nothing about events in the core language itself — it is true that events form a core part of most JavaScript programs, whether that’s reacting to user input in the browser or handling asynchronous I/O in Node[1].

What makes JavaScript particularly *good* for writing event-driven code is that its functions are both lambdas[2] and closures[3], which lets us attach functions as event listeners that can refer to variables in their enclosing scopes. Anyone who’s spent five minutes with jQuery[4] will have written event-driven JavaScript, and if you’ve used Node you’ve probably encountered its Stream API[5], but it’s not always obvious how to test event-driven code. We’ll explore that in this chapter.

3.1. Event emitters

An event emitter is any object that publishes notifications of some kind to tell you when interesting things happen to it. For example, DOM nodes emit events to tell you when and how the user interacts with them, or a Node HTTP server emits an event whenever it receives a request. In Node, there is a built-in class called `EventEmitter`[6] that provides this functionality, but many libraries, both on the browser and server side, implement the same concept in different ways.

When we test an event emitter, our core question is: does the object emit events at the right times and with the right data. The source of these event triggers could be external, caused by calls to the object’s API methods, or internal, where the object itself is monitoring some resource like a socket or a timer and emitting events when things change.

Let’s start with a basic event-publishing object based on Node’s `EventEmitter`. We’ll call our class `Buzzer`, and we’ll give it a `press()` method that simply causes the buzzer to emit a `"buzz"` event using the `emit()` method it inherits from `EventEmitter`.

Figure 3.1. *`node/event_emitters/buzzer.js`*

```
var events = require("events"),
    util   = require("util")

var Buzzer = function() {
  events.EventEmitter.call(this)
}
util.inherits(Buzzer, events.EventEmitter)

Buzzer.prototype.press = function() {
  this.emit("buzz")
}

module.exports = Buzzer
```

This is a very simple piece of code; `press()` is a one-line method that calls a built-in Node function. No conditional logic, no loops, no complexity. However, as we’ll see, there are a variety of options for testing events that all have their strengths and weaknesses.

[1]http://nodejs.org/
[2]A *lambda* is an anonymous function, one that can be invoked without being bound to a name. This means that functions are first-class values and can be passed as arguments to, and given as return values from, other functions.
[3]*Lexical closures* are functions that are syntactically nested, where a function can access not just its own local variables, but those defined in any enclosing function, even after those enclosing functions have returned.
[4]http://jquery.com/
[5]http://nodejs.org/api/stream.html
[6]http://nodejs.org/api/events.html

3.1.1. Black-box testing

As a first attempt, we'll start by testing `Buzzer.press()` as a black box, in the same way we'd test a function that returns a value. We're going to interact with the object from the outside and see what comes out, by registering an event listener that sets a flag, calling `buzzer.press()` and seeing if the flag was set.

Figure 3.2. `node/event_emitters/buzzer_spec.js`

```
var JS     = require("jstest"),
    Buzzer = require("./buzzer")

JS.Test.describe("Buzzer", function() { with(this) {
  before(function() { with(this) {
    this.buzzer = new Buzzer()
    this.called = null
  }})

  it("emits a 'buzz' when pressed", function() { with(this) {
    buzzer.on("buzz", function() { called = true })
    called = false
    buzzer.press()
    assert( called )
  }})
}})
```

At the end of the test, we say `assert(called)`, i.e. we make sure the local variable `called` is set to something truthy. Now, the only code that modifies the `called` flag is the event listener callback we pass to `buzzer.on("buzz")`, so if `called` is true then the event must have been triggered.

That's all well and good, so let's try something a little harder. Many event emitters pass arguments to listeners, for example in jQuery event listeners are passed a DOM `Event` object, and when you make an HTTP request in Node the `"response"` event listenter is passed an `http.IncomingMessage` instance.

Here's another event emitter that passes data to listeners. It's not a dumb `Buzzer`, it's an `Announcer` that lets us broadcast a custom message across a crowded train station.

Figure 3.3. `node/event_emitters/announcer.js`

```
var events = require("events"),
    util   = require("util")

var Announcer = function(prelude) {
  this._prelude = prelude
  events.EventEmitter.call(this)
}
util.inherits(Announcer, events.EventEmitter)

Announcer.prototype.announce = function(message) {
  message = this._prelude + ", " + message
  this.emit("message", message)
}

module.exports = Announcer
```

We can test this much as we did before, treating `Announcer` and its API as a black box. We attach an event listener to the `announcer` that stashes the message it receives in a local variable, before publishing an announcement and checking the message we received. See Figure 3.4, "`node/event_emitters/announcer_spec.js`" below.

Note that this approach *implicitly* checks that the function was called at all, since `message` would remain `null` otherwise. It's a subtle distinction, but bear in mind we're testing two things here: as in the previous example, we're checking that `EventEmitter.emit()`[7] invokes callbacks passed to `EventEmitter.on()`, and we're *also* checking that arguments passed to `emit()` are piped through to the listeners.

Figure 3.4. `node/event_emitters/announcer_spec.js`

```
var JS        = require("jstest"),
    Announcer = require("./announcer")

JS.Test.describe("Announcer", function() { with(this) {
  before(function() { with(this) {
    this.announcer = new Announcer("Attention passengers")
  }})

  it("broadcasts an announcement", function() { with(this) {
    var message = null
    announcer.on("message", function(m) { message = m })
    announcer.announce("there is free cake in the ticket office")
    assertEqual( "Attention passengers, there is free cake in the ticket office", message )
  }})
}})
```

Again, this approach works just fine, resulting in a reasonably easy to read test and sensible failure messages — if `message` is not what we expect it to be, the fact that there's only one assertion in the test means the message "`expected <foo> but was <bar>`" will lead us quickly to the origin of the failure.

Now let's consider a more complicated example. With jQuery you can use `$.on()` to listen to events on a whole collection of elements, that is, rather than binding an event listener to each individual element, we bind one to a collection and we are notified when any member of that collection emits the event. In the event listener, `this` is bound to the element emitting the event and the event object is passed as an argument.

We can write a test for this by listening for clicks on a collection of paragraphs, then using Syn[8] to trigger a click on one particular paragraph and checking our listener is called with the triggering element bound to `this`.

[7]`Buzzer.press()` calls `this.emit()`, which refers to the `emit()` method it inherits from `EventEmitter`.
[8]https://github.com/bitovi/syn

Figure 3.5. *`browser/event_emitters/blackbox_spec.js`*

```
JS.Test.describe("jQuery events (black-box)", function() { with(this) {
  extend(HtmlFixture)

  fixture(" <p></p> <p></p> <p></p> ")

  before(function() { with(this) {
    this.paras  = fixture.find("p")
    this.second = paras.get(1)
  }})

  it("invokes callback with 'this' bound to the element", function(resume) { with(this) {
    var receiver, event

    paras.on("click", function(e) {
      receiver = this
      event    = e
    })

    syn.click(second, function() {
      resume(function() {
        assertSame( second, receiver )
        assertEqual( objectIncluding({target: second, type: "click"}), event )
      })
    })
  }})
}})
```

`HtmlFixture` is a helper module you'll find in the example code that provides the `fixture()` function, used to set up HTML for the test. We use `paras.get(1)` to get a raw DOM element reference out of a jQuery collection, since collections cannot be compared for equality but the underlying DOM nodes can. jQuery binds `this` to a raw node in its event callbacks. We use `objectIncluding()` as a quick matcher for a DOM `Event` object, which has a lot of properties that are incidental to the test.

This approach still technically works, but it begins to get less helpful the more aspects of the callback we check. If we remove the call to `syn.click(second)` then the failure message we get isn't very insightful:

Figure 3.6. An `assertEqual()` error message

```
<[object HTMLParagraphElement]> expected but was
<undefined>
```

This failure message is ambiguous because it doesn't tell us whether the callback was called at all — it may have been called but the arguments we are checking may have been `undefined`. And even in the case where the test passes, we don't know the callback wasn't invoked multiple times and we could only be checking the arguments to the last call; we could check for that with a counter but it's tedious to write and read. So while these assertions are fine for checking the return value of a function, they're not so great at testing callbacks.

But, we have something in our toolbox that's great for checking that a function was called, how many times and with which arguments: mocking.

3.1.2. Mock-based testing

Instead of manually recording the arguments and `this` binding of callbacks ourselves, we can lean on mocking tools[9] to express these requirements much more tersely and get better error messages. In our

[9]See Section 2.6.3, "Testing interactions with mocks".

example, we want to check that the callback is invoked once, with `this` bound to the triggering node, and a single argument which is a DOM event.

We can do this using mocking; if we call `expect(this, "callback")` then `jstest` will generate a function called `callback` inside the test, and we can set expectations about how it should be called. We can then bind this `callback` function as an event listener and check what happens to it.

Figure 3.7. `browser/event_emitters/mock_spec.js`

```
JS.Test.describe("jQuery events (mock)", function() { with(this) {
  extend(HtmlFixture)

  fixture(" <p></p> <p></p> <p></p> ")

  before(function() { with(this) {
    this.paras  = fixture.find("p")
    this.second = paras.get(1)
  }})

  it("invokes callback with 'this' bound to the element", function(resume) { with(this) {
    expect(this, "callback")
        .on(second)
        .given(objectIncluding({target: second, type: "click"}))
        .exactly(1)

    paras.on("click", callback)
    syn.click(second, resume)
  }})
}})
```

We set the expected `this` binding using `on()`, the expected arguments with `given()`, and say it should be called `exactly(1)` time. We can then hand this function off to jQuery as our event listener, and trigger the event, and at the end of the test the framework will check that `callback()` was invoked correctly.

Using mocks for testing event triggering makes it easier to express how the function should be called correctly, and gives us better error messages when it isn't. If we remove the call to `syn.click(second)` this time, we get this error:

Figure 3.8. A mock expectation failure message

```
Mock expectation not met
<[object HTMLParagraphElement]> expected to receive call
callback( objectIncluding({ "target": [object HTMLParagraphElement], "type": "click" }) )
```

The wording "`foo` expected to receive call `callback()`" is a little weird in this case, just remember that it means that `callback()` should have been invoked with `this` bound to `foo`, not that there is a literal expression like `foo.callback()` in the codebase. If you're not familiar with the difference, see Appendix A, *JavaScript functions*.

When using mocks, we'll also get a detailed error if the callback is invoked with unexpected arguments, which again would require us to write more code if we were using the black-box style above.

3.1.3. Cutting out the listener

Recall our very first example of an event emitter:

```
var events = require("events"),
    util   = require("util")
```

```
var Buzzer = function() {
  events.EventEmitter.call(this)
}
util.inherits(Buzzer, events.EventEmitter)

Buzzer.prototype.press = function() {
  this.emit("buzz")
}

module.exports = Buzzer
```

All our tests so far have worked by attaching an event listener, and then making sure that methods that trigger events invoke the listener with the right arguments. Doing it this way has led to rather a lot of complexity due to the nature of testing callbacks, and it would be nice to cut that out. Now, notice that most of the functionality we've been invoking in these tests, i.e. the fact that `EventEmitter.emit()` invokes the listener callbacks, is provided by an external library that's not part of our code, i.e. the `EventEmitter` class. We *could* just assume that that works, and check that our code integrates with it correctly.

That is, given the assumption that `EventEmitter.emit()` works, does our code call it in the right way? We can do that by using mocks to check that `emit()` is called by our class's methods.

Figure 3.9. `node/event_emitters/mock_spec.js`

```
var JS     = require("jstest"),
    Buzzer = require("./buzzer")

JS.Test.describe("Buzzer (mock)", function() { with(this) {
  before(function() { with(this) {
    this.buzzer = new Buzzer()
  }})

  it("emits a 'buzz' when pressed", function() { with(this) {
    expect(buzzer, "emit").given("buzz")
    buzzer.press()
  }})
}})
```

This expresses the intention of the test much more directly and tersely: when we call `buzzer.press()`, it should emit a `"buzz"` event. There are, however, a couple of problems with this.

The first problem is that mocking tools will usually not complain if the function you're faking out does not actually exist. This is by design: a big use case for mocking is passing a completely fake object into a function and checking which methods are invoked on it, and it would be annoying if we had to put a bunch of no-op methods on that object rather than just writing `{}`. But in this case, it means that the following code would pass the test:

Figure 3.10. A broken object that works under mock-based testing

```
var Buzzer = function() {}

Buzzer.prototype.press = function() {
  this.emit("buzz")
}

module.exports = Buzzer
```

This will work because `expect(buzzer, "emit")` places a fake `emit()` method on the object, so that call to `this.emit("buzz")` will not throw an error. But, when used for real, this class will not work since it has not inherited the right functionality from `EventEmitter`.

This is not an argument outright against using mocks for event testing, it simply means that in cases like this, you should probably have an ‘integration’ test as well to check that you can attach listeners to the object and they actually get called.

An important question to ask whenever you’re writing tests, is: do your tests admit implementations that don’t work? It’s fine to have tests that only cover one little aspect of the program — in fact that’s quite desirable — but you do need to make sure the whole program works. Most of the details should be covered by small focussed unit tests, but you need a few integration tests to make sure everything’s wired up correctly too.

The second problem is that if a method emits multiple events, this can be hard to maintain mocks for. For example, calling `model.set({…})` on a Backbone[10] model emits a change for every attribute you’ve set, plus a catch-all `change` event. For example:

Figure 3.11. `browser/event_emitters/backbone_spec.js`

```
JS.Test.describe("Backbone.Model (mock)", function() { with(this) {
  var Person = Backbone.Model.extend({
    toString: function() {
      return "Person{" + this.get("name") + "}"
    }
  })

  before(function() { with(this) {
    this.person = new Person({name: "Alice"})
  }})

  it("emits events when attributes change", function() { with(this) {
    expect(person, "trigger").given("change:name", person, "Merlin", {})
    expect(person, "trigger").given("change:occupation", person, "ceramicist", {})
    expect(person, "trigger").given("change", person, {})

    person.set({name: "Merlin", occupation: "ceramicist"})
  }})
}})
```

This completely specifies the events emitted by `person.set()`. If we remove one of the `expect()` lines, we get an error:

```
Error: <Person{Merlin}> unexpectedly received call to trigger() with arguments:
( "change", Person{Merlin}, {} )
```

Mocked and stubbed methods throw errors when you give them arguments they weren’t expecting, precisely because they’re supposed to check the API contract between two components rather than just let anything go. In situations like this, you can’t mock-test the triggering of one event without testing *all* the events[11], and this can become very complex and hard to maintain, especially when events are triggered that are incidental to the functionality a test focuses on.

Faced with this problem, it’s often easier to listen to the events for real rather than mocking an `emit()` or `trigger()` method. It can often be tricky to express ordering with mocks, whereas if you attach event listeners then you can push the events you see into an array, and check the array’s order at the end of the test. Requiring mocked methods to be called in a certain order is sometimes a sign that there’s a hidden dependency in your application that you should model more explicitly rather than require a certain call

[10]http://backbonejs.org/
[11]Technically, you could add a clause saying only `expect(person, "trigger")`, which would allow `person.trigger()` to be called with any arguments. This means you don’t need to exhaustively list all the events that are triggerd, but it means you won’t get errors if unexpected events happen, which may be undesirable.

order, but sometimes it's important — for example a Node `Stream` should not emit any `"data"` events after it emits `"end"`.

3.1.4. Spy-based testing

If mocking causes you too much trouble, but you still don't like the idea of manually recording function arguments, there's a middle-ground: spies. The Sinon stubbing library[12] provides a spying interface that lets you instrument functions and check what they were called with at the end of the test. Since the checking is done after the calls, rather than setting up expectations beforehand, a spy will not throw errors on unexpected arguments. That means you can focus on checking that one particular event was emitted out of all the other noise going on in your program.

Here's a spy-based version of the above test, focussing on one event and ignoring the others:

Figure 3.12. `browser/event_emitters/spy_spec.js`

```
JS.Test.describe("Backbone.Model (spy)", function() { with(this) {
  var Person = Backbone.Model.extend({
    toString: function() {
      return "Person{" + this.get("name") + "}"
    }
  })

  before(function() { with(this) {
    this.person = new Person({name: "Alice"})
  }})

  it("emits events when attributes change", function() { with(this) {
    var trigger = sinon.spy(person, "trigger")
    person.set({name: "Merlin", occupation: "ceramicist"})
    sinon.assert.calledWithExactly(trigger, "change:name", person, "Merlin", {})
  }})
}})
```

Sinon provides a rich API for checking the arguments, `this` binding, call count, and whether a function was invoked using `new`, both as query methods on spies themselves and as a set of assertions based on those methods. In some situations it can give you just the right balance between mocking and end-to-end testing.

3.2. Event listeners

We've talked a lot about event emitters, but what about event listeners? In some applications we write far more code that listens to events rather than code that emits them. Consider this simple example, which displays the text of a clicked link in a heading element.

Figure 3.13. `browser/event_listeners/bind_link_events.js`

```
var bindLinkEvents = function(container) {
  container.find("a").on("click", function(event) {
    event.preventDefault()
    var linkText = $(this).text()
    container.find("h2").text(linkText)
  })
}
```

[12]http://sinonjs.org/

We can test this fairly easily by setting up an HTML fixture, calling `bindLinkEvents()` on the fixture element, triggering a click event and checking the contents of the heading. You can think of this as an end-to-end test, where we spin up our ‘app’ on some example HTML, trigger some events that simulate user interaction, and then check the final state of the page. We are not instrumenting any internal function calls, we’re sending in user input and checking the displayed output.

Figure 3.14. *`browser/event_listeners/bind_spec.js`*

```
JS.Test.describe("bindLinkEvents()", function() { with(this) {
  extend(HtmlFixture)

  fixture(' <a href="/">Home</a> \
            <h2></h2> ')

  it("displays the text of a clicked link", function(resume) { with(this) {
    bindLinkEvents(fixture)
    syn.click(fixture.find("a"), function() {
      resume(function() {
        assertEqual( "Home", fixture.find("h2").text() )
      })
    })
  }})
}})
```

That works, but notice what we have to do to trigger some behaviour we wanted to test: we need to set up some HTML to represent the user interface, trigger events on that HTML, then check the end result of running the event handler. There’s a lot of coupling there, and it becomes a problem on more complex apps.

For example, what if the same behaviour can be triggered by multiple kinds of events? To test those events we’d need to trigger each of them, and check the end result of the triggered handler to make sure they work. The handler might include a lot of complex logic, or have hard-to-test effects, leading to a lot of duplicated and highly-coupled test code. Triggering the event in the first place might be hard, involving setting up some complex HTML or data model. And triggering some events is undesirable in tests, for example clicking a link makes the browser load the URL in its `href` attribute, unloading the current page and abandoning your tests.

3.2.1. Decoupling listeners from actions

The core purpose of event systems is to decouple things: rather than each object needing to know about all the side effects it has elsewhere in the system and hard-coding them, an object can publish an event. Anything that needs to react to that object’s changes can then listen out for them without the object itself needing to know anything; it publishes one event and any number of listeners can react to it. On the flip side, a listener can be triggered by multiple events, for example an image carousel widget can transition to the next image *either* when the user clicks on a thumbnail, or operates a slider control, or after some amount of time has elapsed. Running end-to-end tests for all these combinations becomes exhaustingly hard to maintain, and works against the event system: rather than keeping things decoupled, we’ve coded a lot of coupled examples into our tests.

So it helps to think of event listeners purely as wiring between pre-existing functions in the system. You have objects with APIs that express what they can do, they emit events when they change in interesting ways, then we use events to wire events and API methods together. Applying this idea to our examples, we have an action — changing the text of a heading — that’s triggered by a click on a link. We can refactor the one big `bindLinkEvents()` into these two ideas, where the event listener extracts the important information and then calls the action.

Figure 3.15. *`browser/event_listeners/widget.js`*

```
var Widget = function(container) {
  this._links   = container.find("a")
  this._heading = container.find("h2")

  var self = this

  this._links.on("click", function(event) {
    event.preventDefault()
    var linkText = $(this).text()
    self.changeHeading(linkText)
  })
}

Widget.prototype.changeHeading = function(headingText) {
  this._heading.text(headingText)
}
```

The `Widget` class represents the UI component we're modelling, listens to any important events emanating from the component's DOM nodes and provides an API for controlling the component. The event listeners and the actions they are bound to have been split up: we can invoke the actions as methods, without triggering an event. This means we can test the event listener and the action it calls independently.

The action method in this case is `changeHeading()`, which sets the heading text of the component. We can easily test that by creating a widget, calling the method and checking the DOM afterward.

Figure 3.16. *`browser/event_listeners/action_spec.js`*

```
JS.Test.describe("changeHeading()", function() { with(this) {
  extend(HtmlFixture)
  fixture(' <h2></h2> ')

  it("changes the heading text", function() { with(this) {
    var widget = new Widget(fixture)
    widget.changeHeading("Welcome to Coventry")
    assertEqual( "Welcome to Coventry", fixture.find("h2").text() )
  }})
}})
```

Once we've established that `changeHeading()` works as expected, all we need to test is that a link click invokes that method correctly, and we can do this using mocking:

Figure 3.17. *`browser/event_listeners/listener_spec.js`*

```
JS.Test.describe("event listener", function() { with(this) {
  extend(HtmlFixture)
  fixture(' <a href="/">Home</a> ')

  it("changes the heading to the link text", function(resume) { with(this) {
    var widget = new Widget(fixture)
    expect(widget, "changeHeading").given("Home")
    var link = fixture.find("a").get(0)
    syn.click(link, resume)
  }})
}})
```

This is a common theme in unit testing: we make sure each method works correctly on its own, and then check the whole system works by testing that those methods are wired up correctly. Also notice how each test only needs *half* of the original HTML fixture: by making the tests more focussed we've decreased the complexity of the context we need to set up for them.

3.2.2. Decoupling and framework patterns

You don't need to give up this design approach when using a framework like Backbone. A Backbone view is just a shorthand for declaring the event bindings that we set up ourselves in the `Widget` class:

Figure 3.18. `browser/event_listeners/backbone_view.js`

```
var View = Backbone.View.extend({
  events: {
    "click a": "handleLinkClick"
  },

  handleLinkClick: function(event) {
    event.preventDefault()
    var linkText = $(event.target).text()
    this.$("h2").text(linkText)
  }
})
```

Notice how we've fallen back to our tightly coupled design where one method — `handleLinkClick()` — deals with both event-related concerns, like preventing the default browser behaviour and extracting the link text, and with implementing the action that's triggered, i.e. changing the heading text. When you try to test this you'll get some hints that the design is problematic.

Figure 3.19. `browser/event_listeners/backbone_view_spec.js`

```
JS.Test.describe("Backbone view", function() { with(this) {
  extend(HtmlFixture)

  fixture(' <a href="/">Home</a> \
            <h2></h2> ')

  before(function() { with(this) {
    this.view = new View({el: fixture})
  }})

  it("changes the heading to the link text", function(resume) { with(this) {
    syn.click(fixture.find("a"), function() {
      resume(function() {
        assertEqual( "Home", fixture.find("h2").text() )
      })
    })
  }})

  describe("handleLinkClick()", function() { with(this) {
    it("prevents the default click behaviour", function() { with(this) {
      var event = {}
      expect(event, "preventDefault")
      view.handleLinkClick(event)
    }})

    it("sets the heading text", function() { with(this) {
      var event = {target: fixture.find("a").get(0)}
      stub(event, "preventDefault")
      view.handleLinkClick(event)
      assertEqual( "Home", fixture.find("h2").text() )
    }})
  }})
}})
```

The first test in this example is end-to-end: we don't stub or mock anything, we just trigger a click event and check the state of the DOM afterward, exactly how we tested `bindLinkEvents()` in Figure 3.14, "`browser/event_listeners/bind_spec.js`". The other tests attempt to unit-test the `handleLinkClick()` method; the first uses mocking to check that it cancels the default link click behaviour, and the second checks that it changes the heading text.

There are a couple of design smells here. The first is that testing that an event handler cancels the browser's link click behaviour using `event.preventDefault()` or by returning `false` should not really be necessary, since if your code doesn't cancel the default behaviour then the browser will navigate away from your page during the test. In other words, this test will never be able to show you a helpful failure message, since the tests will simply crash spectacularly. The second is that, in trying to test that our code changes a heading properly, we need to construct an event object with a `target` property from which we extract the new heading text, and it needs a stub `preventDefault()` method to stop `handleLinkClick()` throwing when it tries to cancel the event. That's rather a lot of indirection and ceremony for just checking the heading-change functionality, and it suggests there should be a more direct interface onto performing that action that doesn't require an event object. In other words: decouple the event listener from the action.

For internal implementation reasons, it is hard to write mock-based tests that make sure you've wired up your `events` declarations properly without actually invoking the event listener, so that's not a very productive option. This is explained in more detail in Section A.3.1, "Mocking and bound methods".

So don't let framework conventions dictate how you should factor your code. If you're doing Backbone, you can split that `handleLinkClick()` into two and have one method call the other. Whichever framework you're using, you can always break something out into a vanilla JavaScript function and test that, before figuring out how to wire it into your framework.

3.3. Pitfalls of event testing

The use of events to decouple components of a system is generally very beneficial and makes things much easier to test. But, if you're not careful, your tests can leave gaps that admit programs that pass the tests but don't actually work.

Consider this contrived but simple example, where two objects communicate via events. One is a counter that emits each natural number in sequence as an event, and the other is an accumulator that sums whatever numbers are fed into it. Here's the `Counter` class:

Figure 3.20. `node/event_emitters/counter.js`

```
var events = require("events"),
    util   = require("util")

var Counter = function() {
  events.EventEmitter.call(this)
  this._count = 0
}
util.inherits(Counter, events.EventEmitter)

Counter.prototype.count = function() {
  this._count += 1
  this.emit("number", this._count)
}

module.exports = Counter
```

And here's `Accumulator`:

Figure 3.21. `node/event_emitters/accumulator.js`

```
var Accumulator = function(emitter) {
  this.sum = 0
  var self = this
  emitter.on("number", function(n) {
    self.sum += n
  })
}

module.exports = Accumulator
```

We can quite easily write tests for these two classes; one should check that `Counter` does indeed emit the natural numbers:

Figure 3.22. `node/event_emitters/counter_spec.js`

```
var JS      = require("jstest"),
    Counter = require("./counter")

JS.Test.describe("Counter", function() { with(this) {
  before(function() { with(this) {
    this.counter = new Counter()
  }})

  it("emits the natural numbers in order", function() { with(this) {
    var numbers = []
    counter.on("number", function(n) { numbers.push(n) })
    for (var i = 0; i < 3; i++) {
      counter.count()
    }
    assertEqual( [1, 2, 3], numbers )
  }})
}})
```

And the other should check that `Accumulator` sums whatever is emitted to it:

Figure 3.23. `node/event_emitters/accumulator_spec.js`

```
var JS          = require("jstest"),
    events      = require("events"),
    Accumulator = require("./accumulator")

JS.Test.describe("Accumulator", function() { with(this) {
  before(function() { with(this) {
    this.emitter     = new events.EventEmitter()
    this.accumulator = new Accumulator(emitter)
  }})

  it("sums the numbers produced by the emitter", function() { with(this) {
    [1, 1, 2, 3, 5, 8].forEach(function(n) {
      emitter.emit("number", n)
    })
    assertEqual( 20, accumulator.sum )
  }})
}})
```

See how we've decoupled the two concepts: `Counter` doesn't care what happens to the numbers it emits, and `Accumulator` doesn't care where the numbers came from or what they are; we used a generic `EventEmitter` to test it and we didn't use the natural numbers, and it worked.

But now suppose that someone decides that `Counter` should not emit an event called `"number"`, the event should be `"data"`, because they want it to work as a Node stream. They make their changes to `counter.js` and `counter_spec.js`, and all seems well. But now, any code that integrates a `Counter` with an `Accumulator` will not work, because the two classes no longer agree on an event name. Since `EventEmitter` doesn't validate the event names you use — you can listen or emit on any event name you like, at any time, it's just a string — the system doesn't alert you to its brokenness. No errors are thrown, the system just silently fails until someone notices the bug in production.

This is one of the most common problems I've seen when teams work on complex JavaScript UI codebases, or really any somewhat modularised software architecture: they've decoupled their architecture but find it hard to make sure all the emitters and listeners continue to work together.

Broadly speaking, there are three common solutions to this problem: change your culture, change your tests, or change your code.

3.3.1. Events as a public interface

The cultural solution to this problem is to acknowledge that the events emitted by an object form part of its public API: the event names will be used and depended upon by clients using the object. Because of this, you should apply the same level of caution to the events an object emits as to the public methods it exposes.

The problem is that, if you change a method name, then code calling the old name will throw, and you'll see your programs emitting errors[13], but when you change an event name, listeners on the old name will silently sit there, never being invoked, so event renames are harder to catch. The way to catch these problems is to develop a habit of unit testing the events an object emits, and treating your tests as documentation: your tests state what the public interface of your objects is, and you should feel uneasy and pause to consider the consequences whenever you change them.

3.3.2. Selective integration testing

The testing solution is to add a judicious amount of integration tests, to augment your base of unit tests. The integration tests should tell you what to change in your unit tests, which should tell you what to change in your code. Now, what 'integration testing' means will differ between projects and teams, but here are some common examples.

Integration testing could mean writing further tests, within your JavaScript testing framework, that exercise common combinations of event emitters and listeners. For example we could test `Counter` and `Accumulator` together:

Figure 3.24. `node/event_emitters/counting_spec.js`

```
var JS          = require("jstest"),
    Counter     = require("./counter"),
    Accumulator = require("./accumulator")

JS.Test.describe("Counting", function() { with(this) {
  before(function() { with(this) {
    this.counter     = new Counter()
    this.accumulator = new Accumulator(counter)
  }})

  it("sums the natural numbers", function() { with(this) {
    for (var i = 0; i < 5; i++) {
      counter.count()
    }
    assertEqual( 15, accumulator.sum )
  }})
}})
```

[13]It's good practice to find some way of logging all errors emitted by your application in production, including those occurring in the browser, and having a monitoring system that makes a lot of noise when things aren't healthy.

Depending on the number of possible combinations in your system, this may become prohibitively expensive and will not be a very good time investment. If you find yourself wanting to do too much of this, it means you're not confident enough in the stability and design of the system, so consider the advice of the previous section. However, you should have *some* integration tests, so try to prioritise the combinations of things you test by what commonly gets used in practice. In the same way that good unit tests cover the public API that clients will interact with, good integration tests emphasise the common use cases to make sure they work well.

This doesn't just apply at the application structure level, about which code commonly gets integrated together. It also applies at the user interface level. You can use tools like Selenium[14] to run full-stack tests of your web app, focus on the most common user journeys and interactions, and make sure they are stable[15]. This level of integration testing tends to be extremely slow, hard to set up, and therefore expensive, and should be used sparingly: test the most common paths like this, and try and push all the details and edge cases down into unit tests, which are cheaper to write and to quicker to run.

3.3.3. Changing your API design

Finally, you should take a hint when you find it hard to verify that code works correctly: if it's hard to test, it's probably hard to use, whereas code that's easy to interact with and gives you good feedback tends to be easier to test.

In the case of events, our problem was that changing an event name can leave dangling listener code that silently fails to receive the data it's expecting. We mentioned that changing the name of the method usually gives us better feedback, because JavaScript will throw if we invoke a method that doesn't exist. We can improve this by changing the design of our event interfaces.

The vast majority of JavaScript event systems work by having objects that can emit many types of events, which are named using strings. Here's a highly simplified version of what most of these systems provide:

Figure 3.25. A simplified `EventEmitter` implementation

```
var EventEmitter = function() {
  this._events = {}
}

EventEmitter.prototype.on = function(event, listener) {
  if (this._events[event] === undefined) {
    this._events[event] = []
  }
  this._events[event].push(listener)
}

EventEmitter.prototype.emit = function() {
  var args      = Array.prototype.slice.call(arguments),
      event     = args.shift(),
      listeners = this._events[event]

  if (listeners === undefined) return

  for (var i = 0, n = listeners.length; i < n; i++) {
    listeners[i].apply(this, args)
  }
}
```

[14]http://www.seleniumhq.org/
[15]See Section 5.4.5, "Testing via the browser".

An `EventEmitter` instance has a property called `_events`, an object that maps event name strings to arrays of listener functions. The `on()` method lazily creates an array for each event type and stores the function it's given, and `emit()` looks up the list for the given event name and invokes all the functions in it with the given arguments[16].

This implementation is designed to be highly *dynamic*: we can emit and listen to events with any name at any time, without needing to declare the allowed names in advance. Anything goes, and in systems that allow anything, you will never get helpful errors.

Now, there's a hidden abstraction in this implementation that we can pull out: `_events` is an object whose properties are event names. What if we made that the public interface, rather than our string-based one. That is, instead of calling `object.on("nameOfEvent", function() { … })` we would call `object.nameOfEvent.listen(function() { … })`. Let's implement that:

Figure 3.26. *`node/event_emitters/listenable.js`*

```
var Listenable = function(object) {
  this._object    = object
  this._listeners = []
}

Listenable.prototype.listen = function(listener) {
  this._listeners.push(listener)
}

Listenable.prototype.emit = function() {
  for (var i = 0, n = this._listeners.length; i < n; i++) {
    this._listeners[i].apply(this._object, arguments)
  }
}

module.exports = Listenable
```

We pass in an object in the constructor because we still want to be able to invoke the listeners with `this` bound to the object that's emitting the event.

Let's redesign the `Person` model that we tested in Figure 3.11, "`browser/event_emitters/backbone_spec.js`" around this `Listenable` concept. Because we no longer have a dynamic event name register, the model must create listenables for the events it wants to publish on creation, ready for clients to listen to.

[16]`Array.prototype.slice.call(arguments)` copies all the arguments passed to the function into a new `Array`. Unfortunately, the JavaScript `arguments` object is not itself an `Array`; it *is* an ordered list of values but it doesn't have all the `Array` methods, and so we cannot call `arguments.slice()`. We must instead borrow the `slice()` method from `Array.prototype` and invoke it on the `arguments` object. Copying the arguments into a new array means we can mutate it, for example using `args.shift()` to pop the first argument off the front of the list. You may also see the expression `[].slice.call(arguments)` in the wild, which does the same thing only it gets the `slice()` method from an actual `Array` instance rather than from the prototype.

Figure 3.27. `node/event_emitters/person.js`

```
var Listenable = require("./listenable"),
    util       = require("util")

var Person = function(attributes) {
  this.change            = new Listenable(this)
  this.change.name       = new Listenable(this)
  this.change.occupation = new Listenable(this)

  this._attributes = attributes
}

Person.prototype.set = function(attributes) {
  var changed = false
  for (var key in attributes) {
    if (this._attributes[key] === attributes[key]) {
      continue
    }
    this._attributes[key] = attributes[key]
    this.change[key].emit(attributes[key])
    changed = true
  }
  if (changed) this.change.emit()
}

module.exports = Person
```

A client using this class would listen to changes like so:

Figure 3.28. Listening to changes on a `Listenable` interface

```
var person = new Person({name: "Bob"})

person.change.name.listen(function(newName) {
  // ...
})
```

The important point about this interface is that if the event names are changed, then above expression will throw an error like, “cannot call method `listen` of `undefined`,” because the properties we are trying to access do not exist. This design is less dynamic, but leans on the language more to give us better feedback about our code not working.

As we did in Figure 3.11, “`browser/event_emitters/backbone_spec.js`”, we can test that the `Person.set()` method publishes events using mocking. But because each event is a separate object with its own `emit()` method, rather than one object having one `emit()` method that takes the event name as an argument, we can mock individual events and we don’t get errors for not having mocked all of them.

Figure 3.29. `node/event_emitters/person_spec.js`

```
var JS     = require("jstest"),
    Person = require("./person")

JS.Test.describe("Person", function() { with(this) {
  before(function() { with(this) {
    this.person = new Person({name: "Alice"})
  }})

  it("emits events when attributes change", function() { with(this) {
    expect(person.change.name, "emit").given("Merlin")
    person.set({name: "Merlin", occupation: "ceramicist"})
  }})
}})
```

In practice, it pays to use a mixture of all the above methods — culture, testing, and API design — to make it easier to write working software, and figuring out the balance takes time, practice and consideration of each problem. For example, the event-name-as-string convention is so strong in JavaScript that something like `Listenable` just won't feel right to developers, despite some of its technical advantages. There is no silver bullet, and different people manage risk in different ways, so be thoughtful about how you apply these techniques on each project.

3.4. Streams

Streams are one of the main building blocks of Node programs, in fact it's reasonable to say streams are *the* core feature of Node. Pretty much all I/O is done using them, and Node provides simple abstractions for piping data from input to output streams and processing it along the way. Almost any non-trivial Node program will have to deal with — and have tests involving — streams.

The 'hello world' Node program is an HTTP server, and that's where most people are introduced to streams. For example, you can serve a file from disk by piping a readable file stream into a writable HTTP response.

Figure 3.30. A streaming static file server

```
var fs   = require("fs"),
    http = require("http")

var server = http.createServer(function(request, response) {
  var file = fs.createReadStream(__dirname + request.url)
  file.pipe(response)
})

server.listen(8000)
```

Similarly, the standard I/O interfaces are all streams; `process.stdin` is a readable stream and `process.stdout` and `process.stderr` are writable streams. Streams make up a lot of the data flow plumbing in Node programs and many libraries export stream-based interfaces.

I bring up streams, and readable streams in particular, in this chapter because they are event emitters. A readable stream emits `"data"` events with each chunk of output and emits `"end"` when there is no more output to emit[17]. The `Stream.pipe()` method uses these events and other parts of the API to coordinate data transfer from a readable to a writable stream, dealing with back-pressure[18] and errors.

As a little example, here's a program that reads chunks from standard input, converts them to uppercase and writes them to standard output.

Figure 3.31. `node/streams/upcase.js`

```
process.stdin.on("data", function(chunk) {
  chunk = chunk.toString("utf8").toUpperCase()
  process.stdout.write(chunk)
})
```

If we pipe some data into this program we'll see it working as intended. To borrow James Halliday's favourite example[19]:

[17]These events are the original or 'streams1' API; originally there was a single `Stream` class with just one method, `pipe()`. If you wanted to make your own streams, you had to implement `emit("data")`, `emit("end")`, `pause()` and `resume()` methods for readable streams, and `write()`, `end()` and `destroy()` methods for writable streams. In Node v0.10, the 'streams2' API was introduced, including `Readable` and `Writable` subclasses of `Stream` that deal with most common problems, including back pressure and buffer management, for you. Although there are new APIs, such as `stream.read()` in place of `stream.on("data")`, the old event-emitter interface still works but after Node v0.10 you should try to use the new API.

[18]*Back-pressure* is what happens when a writable stream cannot process data as fast as the readable stream is piping that data in. In this situation, the writable stream should tell the readable stream to 'back off' until the writable stream is ready for more input.

[19]James Halliday, aka 'substack', maintains an excellent guide to using and implementing streams at https://github.com/substack/stream-handbook.

Figure 3.32. Transforming input using streams

```
$ (echo beep ; sleep 1 ; echo boop) | node node/streams/upcase.js
BEEP
BOOP
```

This example hints at a tricky aspect of testing stream output, which is that you can view the stream's behaviour in two ways: you can say how it affects the entire input — it converts `"beep\nboop\n"` into `"BEEP\nBOOP\n"` — or you can say it transforms each chunk of input to uppercase. It's a subtle distinction between the net effect and the details, and which view you use will depend on what type of logic you're testing, as we'll see.

3.4.1. Building a data source

Very often when dealing with streams, you will either be checking the output of a readable stream, or checking the effect of a transform stream — one that's both readable and writable, that transforms the data written to it and hands the result off to whatever is reading from it. To illustrate the latter, it helps to have some dummy source stream to feed data into the transformer, and this source will also be handy for illustrating how to test simple readables.

Let's create a class called `Source`, based on the 'streams2' `Readable` class. The `Readable` class works by invoking a method called `_read()` when a consumer wants to read data, either via the `stream.pipe()` or `stream.read()` method or the `stream.on("data")` listener. The subclass's `_read()` method must use `push()` to queue up the output it wants to emit, which `Readable` will either immediately hand off to the consumer or buffer in memory if the consumer is not ready for it yet.

The `Source` class take a list of chunks and a time interval, and emits the chunks at times separated by that interval. For example, if we wrote

Figure 3.33. Piping a `Source` to standard output

```
var stream = new Source(["testing ", "your ", "javascript"], 1000)
stream.pipe(process.stdout)
```

then the program would print `testing your javascript` with a one-second delay between each word. We can also use `stream.on("data")`:

Figure 3.34. Listening for `"data"` on a `Source`

```
// Prints "testing", "your", "javascript", one word per line

stream.on("data", function(chunk) {
  console.log(chunk.toString())
})
```

or, we can grab fixed numbers of bytes from it using `stream.read()`:

Figure 3.35. Reading fixed numbers of bytes from a `Source`

```
// Prints three lines: "test", "ing y", "our ja"

stream.read() // -> null, since the first timeout has not elapsed

setTimeout(function() {
  console.log(stream.read(4).toString())
  console.log(stream.read(5).toString())
  console.log(stream.read(6).toString())
}, 4000)
```

Here’s the implementation:

Figure 3.36. `node/streams/source.js`

```
var stream = require("stream"),
    util   = require("util")

var Source = function(chunks, interval) {
  stream.Readable.call(this)

  this._chunks   = chunks
  this._index    = 0
  this._interval = interval
}
util.inherits(Source, stream.Readable)

Source.prototype._read = function() {
  var self = this
  setTimeout(function() {
    if (self._index === self._chunks.length) {
      self.push(null)
    } else {
      self.push(self._chunks[self._index])
      self._index += 1
    }
  }, this._interval)
}

module.exports = Source
```

This class lets us create readables with canned output, where we can control both the content of the output and the rate at which it’s emitted, which is useful in many testing situations.

3.4.2. Checking a readable stream’s output

We said there were two ways of viewing a stream’s output: as a big blob, or as a series of small chunks. Let’s look the series-of-chunks view first, which we can test as follows. We can use a `"data"` listener on the stream to push everything it emits into an array. Then, once the stream emits `"end"`, we resume the test and check the contents of the array.

Figure 3.37. `node/streams/ondata_spec.js`

```
var JS     = require("jstest"),
    Source = require("./source")

JS.Test.describe("Source events", function() { with(this) {
  before(function() { with(this) {
    this.stream = new Source(["a", "b", "c"], 10)
    stream.setEncoding("utf8")
  }})

  it("yields output via the 'data' event", function(resume) { with(this) {
    var data = []
    stream.on("data", data.push.bind(data))

    stream.on("end", function() {
      resume(function() {
        assertEqual( ["a", "b", "c"], data )
      })
    })
  }})
}})
```

When you call `stream.on("data")`, the stream goes into 'flowing mode': instead of having to *pull* data out using `stream.read()`, the stream *pushes* data at you as soon as it's available. The `Readable` class deals with calling the stream's `_read()` method as often as required internally. `_read()` (with an underscore) is an internal implementation detail, while `on("data")`, `pipe()` and `read()` (no underscore) are the public API.

Testing via the `stream.on("data")` listener lets you see the structure of what the stream emits, i.e. how the chunks are divided up. This is important in some scenarios, for example the split[20] library splits the input stream on line breaks and should emit one `"data"` event per parsed line. Similarly, the websocket-driver[21] library emits one `"data"` event per complete message it receives, and expects messages to be written to it as one chunk per discrete message. In these situations, the boundaries between chunks in a stream are very important, and we should test for them using `"data"` events.

The other way we can test a stream is by collecting all of its output into one big blob; if we don't care about exactly how the chunks are arranged internally this means we can wait for all the output and check it as a single string. We could easily modify our previous test by calling `data.join()` to merge all the chunks together, but there's an easier way, using the concat-stream[22] library. `concat()` takes a function and returns a writable stream that collects all the data written to it and yields it as one big buffer to the function when the stream writing to it is complete.

Here's how we could test our `Source` class this way:

Figure 3.38. `node/streams/pipe_spec.js`

```
var JS     = require("jstest"),
    concat = require("concat-stream"),
    Source = require("./source")

JS.Test.describe("Source.pipe()", function() { with(this) {
  before(function() { with(this) {
    this.stream = new Source(["a", "b", "c"], 10)
    stream.setEncoding("utf8")
  }})

  it("emits all the output", function(resume) { with(this) {
    stream.pipe(concat(function(output) {
      resume(function() {
        assertEqual( "abc", output )
      })
    }))
  }})
}})
```

Testing via `stream.pipe(concat())` is great for situations where you just want all the output, without caring when bits of it arrive. A good example is running a shell command and collecting all the output it printed, or buffering the body of an HTTP response before parsing it with `JSON.parse()`. In these situations, the boundaries between chunks emitted through `stream.on("data")` have no semantic meaning beyond when the data happened to arrive, so it's not useful to preserve those boundaries when writing a test. Testing by checking the combined output is a good way of explicitly saying that the internal chunk structure doesn't matter.

3.4.3. Testing a transform stream

A transform stream is something you use all the time when using the shell. When you write something like this in your terminal:

[20]https://npmjs.org/package/split
[21]https://npmjs.org/package/websocket-driver
[22]https://npmjs.org/package/concat-stream

Figure 3.39. A shell pipeline

```
$ find code/ -type f | cut -d. -f2 | sort | uniq
```

you are using the `cut`, `sort` and `uniq` programs as transforms: they take data from standard input, process it in some way, and emit new data on standard output to be fed into the next program. In Node, a putative API for this might look like:

Figure 3.40. A Node stream pipeline

```
find("code/", {type: "f"}).pipe(cut({d: ".", f: 2})).pipe(sort()).pipe(uniq())
```

In this imaginary API, `find()` would return a readable stream, and `cut()`, `sort()` and `uniq()` would all return transform streams. In Node, a transform stream is one that's both readable and writable, and processes the data written to it in some way before handing the processed data off to the next consumer.

We've already mentioned one transform: the split module. `split()` returns a transform that scans the data written to it and splits it into chunks separated by line breaks (`"\n"`). It gives you the stream you piped in, with its existing chunk boundaries removed, and new boundaries put in where the line breaks were. If we wanted to test it, we could spin up an instance of our `Source` class to feed data into it, and collect the output on the other side using `"data"` events. Remember, in this case the internal boundary structure of the output is important, so we use events rather than concatenating all of the output.

Figure 3.41. `node/streams/split_spec.js`

```
var JS     = require("jstest"),
    Source = require("./source"),
    split  = require("split")

JS.Test.describe("split()", function() { with(this) {
  before(function() { with(this) {
    this.chunks = ["some chop", "ped up\nte", "xt to\n", "parse"]
  }})

  it("splits the input stream on line breaks", function(resume) { with(this) {
    var source = new Source(chunks, 10),
        data   = [],
        stream = source.pipe(split())

    stream.on("data", data.push.bind(data))

    stream.on("end", function() {
      resume(function() {
        assertEqual( ["some chopped up", "text to", "parse"], data )
      })
    })
  }})
}})
```

You can see how, if we were to use `stream.pipe(concat())` to test this, the only output we'd see is the string `"some chopped uptext toparse"`, telling us nothing about where `split()` set the chunk boundaries. We'd have thrown too much information away for the test to be any use.

3.4.4. Making your own transforms

Now, let's try making our own transform. There are some widely used libraries for building these, such as through[23], map-stream[24] and other modules by Dominic Tarr[25]. But since Node v0.10 introduced

[23]https://npmjs.org/package/through
[24]https://npmjs.org/package/map-stream

its own `Transform` class, let’s use that. Let’s revisit our `upcase.js` program from Figure 3.31, “`node/streams/upcase.js`” and make the glue that sat between `stdin` and `stdout` into a generic stream:

Figure 3.42. `node/streams/upcase_transform.js`

```
var stream = require("stream"),
    util   = require("util")

var Upcase = function() {
  stream.Transform.call(this)
}
util.inherits(Upcase, stream.Transform)

Upcase.prototype._transform = function(chunk, encoding, callback) {
  chunk = chunk.toString("utf8").toUpperCase()
  this.push(chunk)
  callback()
}

module.exports = Upcase
```

This is a simple example. It converts whatever chunk is given to it into uppercase, pushes the chunk into the read buffer and then runs `callback()` to tell Node it’s finished processing. Transforms can be much more complex than this, however; they can emit output asynchronously, they can emit more or fewer output chunks than input chunks, and so on. Whatever their internal logic, we can test them using our `stream.pipe(concat())` pattern or using `stream.on("data")` listeners, depending on whether we care about the output’s structure or not.

In this example, we don’t care about structure, we just want all the input turned into uppercase output, so we’ll use the `concat()` approach.

Figure 3.43. `node/streams/upcase_spec.js`

```
var JS     = require("jstest"),
    concat = require("concat-stream"),
    Source = require("./source"),
    Upcase = require("./upcase_transform")

JS.Test.describe("Upcase.pipe()", function() { with(this) {
  before(function() { with(this) {
    this.source = new Source(["a", "b", "c"], 10)
  }})

  it("transforms its input to uppercase", function(resume) { with(this) {
    var stream = source.pipe(new Upcase())
    stream.setEncoding("utf8")

    stream.pipe(concat(function(output) {
      resume(function() {
        assertEqual( "ABC", output )
      })
    }))
  }})
}})
```

But there’s one important difference between readable and transform streams: `Transform._transform()` takes a callback to tell us when it’s finished processing the input, whereas `Readable._read()` does not take a callback to tell us it’s finished emitting data. Being told when it’s done processing means we can actually use mocking to test transforms: rather than deal with all the

[25]https://github.com/dominictarr

complexity of shunting stream output around, we can just say what we expect `push()` to be called with, and invoke `_transform()` to check that's the case.

Figure 3.44. `node/streams/upcase_mock_spec.js`

```
var JS     = require("jstest"),
    Upcase = require("./upcase_transform")

JS.Test.describe("Upcase", function() { with(this) {
  before(function() { with(this) {
    this.upcase = new Upcase()
  }})

  it("pushes an uppercase string", function(resume) { with(this) {
    expect(upcase, "push").given("WHAT UP")
    upcase._transform(new Buffer("what up"), "utf8", resume)
  }})
}})
```

This is a nice option in simple cases where `push()` will invoked once or not at all, but if it's invoked multiple times you might find you need to specify the ordering of the data chunks that are passed to it. In that case, using a `"data"` listener or concatenating the output will probably be more reliable, and easier to set up, than asserting what order all the mocked calls should happen in.

3.4.5. Decoupled streams

As programs get more complicated, it often seems like it'll be harder to test them, and streams are no exception. But the stream ecosystem has sprung up a culture of keeping streams small, focussed and modular, and there are various tools for doing so. Following this pattern helps make streams easier to test by having each stream do something simple, and building more complex streams from those simple building blocks.

As an example, consider the Unix `grep` program, which reads from stdin and prints only those lines containing a given string to stdout.

Figure 3.45. Using the Unix `grep` program

```
$ (echo hello ; echo goodbye) | grep h
hello

$ (echo hello ; echo goodbye) | grep g
goodbye
```

If we were to try and write this in Node, we could build it as a monolithic script, or we could build a stream to do the filtering and then connect it to `stdin` and `stdout`.

Figure 3.46. `node/grep_stream/grep.js`

```
var grep  = require("./grep_stream"),
    query = process.argv[2]

process.stdin
    .pipe(grep(query))
    .pipe(process.stdout)
```

Furthermore, we can break the `grep()` abstraction down even further by noting that it's really three separate tasks. First, we split the input stream into individual lines; we have the `split()` stream for doing that. Then we can pass these lines through a filter that only allows matching chunks through, and finally we pass the matching chunks through a transform that puts the line breaks back in, so that we get each match printed on a new line.

If we were to write this program using arrays, it might look like this:

Figure 3.47. Composing array operations for `grep`

```
var output = process.stdin.read()
                .split("\n")
                .filter(function(line) {
                  return line.indexOf(query) >= 0
                })
                .join("\n")

process.stdout.write(output)
```

See how the `filter()` function doesn't care whether the elements of the array are individual lines of a file, and it doesn't care that they'll be presented line-by-line. It's *only* a filter, allowing those inputs matching some condition to pass through. The stream equivalent of this would be a transform stream that pushes each chunk that matches the query text.

Figure 3.48. `node/grep_stream/filter.js`

```
var stream = require("stream"),
    util   = require("util")

var Filter = function(query) {
  stream.Transform.call(this)
  this._query = query
}
util.inherits(Filter, stream.Transform)

Filter.prototype._transform = function(chunk, encoding, callback) {
  chunk = chunk.toString("utf8")
  if (chunk.indexOf(this._query) >= 0) {
    this.push(chunk)
  }
  callback()
}

module.exports = Filter
```

Since this stream does not have any logic about splitting the input on line breaks, it should not have any logic about adding line breaks to the output either. This makes it a generic filter rather than something we can only use on line-oriented input. We can build the line-delimiting separately as another transform, that essentially does the opposite of `split()`: it emits each chunk passed to it and appends a line break after each one. Let's use the `through()` module to bypass some of the boilerplate of making a transform stream.

Figure 3.49. `node/grep_stream/join.js`

```
var through = require("through")

var join = function(separator) {
  return through(function(chunk) {
    this.push(chunk)
    this.push(new Buffer(separator || "\n"))
  })
}

module.exports = join
```

Having built the splitting, filtering and joining ingredients, implementing `grep` is a matter of combining those streams into a single pipeline, which we can do using the stream-combiner[26] module:

Figure 3.50. `node/grep_stream/grep_stream.js`

```
var combine = require("stream-combiner"),
    split   = require("split"),
    Filter  = require("./filter"),
    join    = require("./join")

var grep = function(query) {
  return combine(split("\n"), new Filter(query), join("\n"))
}

module.exports = grep
```

This is a direct translation of the array-based program in Figure 3.47, “Composing array operations for `grep`” to streaming form. And, lo and behold, the combined stream works as expected:

Figure 3.51. Running the `grep.js` program

```
$ (echo hello ; echo goodbye) | node node/grep_stream/grep.js h
hello

$ (echo hello ; echo goodbye) | node node/grep_stream/grep.js g
goodbye
```

Separating the problem into smaller ingredients means it’s still quite easy to test, both using unit tests of the individual parts and using end-to-end tests of the whole. We can write a black-box test of the `grep` stream itself, using our friend `concat()` to collect all the output:

Figure 3.52. `node/grep_stream/grep_spec.js`

```
var JS     = require("jstest"),
    concat = require("concat-stream"),
    grep   = require("./grep_stream"),
    Source = require("../streams/source")

JS.Test.describe("grep", function() { with(this) {
  before(function() { with(this) {
    this.stream = new Source(["be", "ep\nb", "oo", "p"], 10)
  }})

  it("filters for matching lines in the input", function(resume) { with(this) {
    stream.pipe(grep("boo")).pipe(concat(function(result) {
      resume(function() {
        assertEqual( "boop\n", result.toString("utf8") )
      })
    }))
  }})
}})
```

And, we can use mock-based testing on the `Filter` class to check it does its job properly. We use `expect(filter, "push").exactly(0)` to say that `filter.push()` should not be called at all. Notice how this test is simplified because we don’t have line-separation logic to worry about — there’s no uncertainty about whether the filter calls `push(chunk + "\n")` or pushes each string separately, because this stream doesn’t handle that logic. We only worry about how filtering works here.

[26]https://npmjs.org/package/stream-combiner

Figure 3.53. `node/grep_stream/filter_spec.js`

```
var JS     = require("jstest"),
    Filter = require("./filter")

JS.Test.describe("filter", function() { with(this) {
  before(function() { with(this) {
    this.filter = new Filter("lo")
  }})

  it("pushes chunks that match the query", function(resume) { with(this) {
    expect(filter, "push").given("hello")
    filter._transform(new Buffer("hello"), "utf8", resume)
  }})

  it("doesn't push chunks that don't match the query", function(resume) { with(this) {
    expect(filter, "push").exactly(0)
    filter._transform(new Buffer("goodbye"), "utf8", resume)
  }})
}})
```

This separation of concerns is not just good for tests, it makes the code more *useful* since one bit of logic can be more freely glued to others, rather than having one monolith with all the functionality in it. You can think of streams as being like arrays, except they sequence elements in time rather than in memory. Imagine the operations you would use to process array-based data, and figure out how to convert that to streaming data.

Being easy to test is a side effect of being easy to use — consider the usability of the modules you write and you should find you end up writing simpler tests for them.

4. Browser applications

For nearly two decades, JavaScript has been the language of web. Many languages have sought to replace it, offering alternative syntax, type systems, functional programming support, and many other new features, but behind the scenes they all compile down to JavaScript. If you're writing code for the browser, you're using JS.

Client-side JavaScript is often regarded as being very difficult to test, at least compared to server-side code. To test server-side applications, you can just write a script in whatever language you happen to be using that calls into your application and checks things. Or, you spin up your application server and fire HTTP requests at it to check the app responds to them correctly. For a long time, JavaScript testing was seen as an extension of this approach; we would launch the application server and simulate user interaction through a web browser, checking that the full stack works from the end user's point of view. This model is embodied in tools like Selenium[1].

While full-stack testing still has its uses, it is far more productive to unit-test individual components of your front-end code in isolation, without having to spin up your entire stack. Tests written this way will run far quicker than full-stack tests, and are less easily broken by unrelated changes in the back-end code. We have always had the technology to do this — indeed all we need is a web browser, no specialist tools are required — but it does require us to structure our code so that it's amenable to this style of testing.

The way we write code and the way we test it are deeply intertwined, and this is especially true in the browser. Successfully testing code that interacts with browser APIs depends on constructing the code itself to make it easier to test, and architecting your application so that its components can be tested on their own plays a big role in this. So, this chapter will focus as much on how to write the code itself as on how to test it.

4.1. Form validation

Let's start with what was for a long time one of the few serious uses of JavaScript on the web: checking people typed valid data into forms[2]. As well as being a relatively simple problem, it's an excellent example of separating pieces of logic to make testing easier.

Suppose we have a sign-up form with two fields, `email` and `password`, and we want to validate them on the client before allowing the form to submit. We start out by writing the form as working HTML, and then place a `<script>` after it that layers on some validation logic. On form submission, it checks each field, adds an error to the form if the field is not valid, and then stops the form from submitting if *any* of the fields were invalid. This script is shown in Figure 4.1, "`browser/form_validation/form.html`".

Although this page works, it is very hard to test without resorting to slow full-stack testing with something like Selenium. We'll examine the reasons for this after the script.

[1] http://docs.seleniumhq.org/

[2] Although client-side validation can improve an app's responsiveness and make it more pleasant to use, you should *always* have the same validation implemented on the server. A server should not automatically trust data it receives over the wire.

Figure 4.1. `browser/form_validation/form.html`

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Sign up</title>
  </head>
  <body>

    <form method="post" action="/">
      <p>
        <label for="email">Email</label>
        <input type="text" name="email" id="email">
      </p>
      <p>
        <label for="password">Password</label>
        <input type="password" name="password" id="password">
      </p>
      <input type="submit" value="Sign up">
    </form>

    <script src="../../node_modules/jquery/dist/jquery.js"></script>

    <script>
      $("form").on("submit", function(event) {
        var form     = $(this),
            email    = form.find("[name=email]"),
            password = form.find("[name=password]")

        form.find(".error").remove()

        if (!email.val().match(/^[a-z]+@[a-z]+\.com$/)) {
          email.parents("p").after('<p class="error">Email address is not valid</p>')
        }
        if (password.val().length < 8) {
          password.parents("p").after('<p class="error">Password is too short</p>')
        }

        if (form.find(".error").length > 0) {
          event.preventDefault()
        }
      })
    </script>

  </body>
</html>
```

So, why is this hard to test? Well, when we write unit tests for browser code, we ideally want a static web page that loads code from our application, loads and executes our tests, and displays the test results. This setup means we can open the test page in any browser, on any device, and check that things work. As soon as we rely on server-side software or browser automation tools, we make it harder to run the tests since more software needs to be installed and running to support them. A static HTML page should always be your starting point.

Now, we've buried this markup and script in a page on our production website, so any test page we make has no way to load this code and interact with it. We can start to fix this by extracting the validation logic into a reusable function.

4.1.1. Extracting user interface logic

The example above uses a `<script>` tag to add some dynamic behaviour to the markup on the page. While that might work correctly, it's hard to test since we can't manipulate the behaviour of the page — the page loads, the script runs, and that's that. JavaScript test suites need to run without reloading the page, so we can see the results of all the tests we run without them vanishing on page refresh. So, our code needs to be written so that we can run it many times within the same page to see how it behaves in different scenarios.

The most basic way we can do this is by wrapping the script up in a function. Note how this function no longer hard-codes the `$("form")` selector but allows a `selector` to be passed in; this lets us make the function bind to our fixture HTML in our tests while still being able to pass in the 'real' selector when we integrate the validation into our production pages. You should try to avoid hard-coding the root selectors for elements your scripts bind to, but pass the selector in to give you more freedom to use the script in different situations.

Figure 4.2. `browser/form_validation/validate_form.js`

```
var validateForm = function(selector) {
  $(selector).on("submit", function(event) {
    var form     = $(this),
        email    = form.find("[name=email]"),
        password = form.find("[name=password]")

    form.find(".error").remove()

    if (!email.val().match(/^[a-z]+@[a-z]+\.com$/)) {
      email.parents("p").after('<p class="error">Email address is not valid</p>')
    }
    if (password.val().length < 8) {
      password.parents("p").after('<p class="error">Password is too short</p>')
    }

    if (form.find(".error").length > 0) {
      event.preventDefault()
    }
  })
}
```

This function does exactly the same thing as our original `<script>` element, but it allows us to reuse the logic and invoke it multiple times in our test suite. It's the first step towards making the code easier to test.

4.1.2. Extracting markup for tests

In order to test our user interface code we also need some HTML for it to interact with. How much you need this will vary depending on your framework and architecture; some apps will generate all their HTML using JavaScript while some will start from a working HTML document and use progressive enhancement[3] to layer scripted behaviour over the top. Whichever approach you use, you will probably need some elements on the page for the script to target.

It is initially tempting to put such markup into the test page itself — the `test.html` page in all our examples — but this tends to cause problems. The markup *must* be reset into its initial state before each test runs, so that the results of one test don't accidentally affect the behaviour of subsequent tests. Many changes can be made to the elements during each test: elements can be added, moved, and deleted, their

[3]http://alistapart.com/article/understandingprogressiveenhancement

text and attributes can be changed, and event listeners can be registered and cancelled. The code you're testing might have started a `setInterval()` loop that periodically changes the content, and if that loop is not cancelled and markup is shared between tests, then behaviour initiated in one test will affect the outcome of the next, producing strange and misleading results. It is often far easier to delete the markup at the end of each test and inject a fresh copy of the original before the next test begins.

A secondary problem is that, depending on how you choose to run your tests, you might not have access to the host page. There are many different frameworks for running browser-based unit tests[4] and many of them provide their own HTML context in which they execute your tests. So, while it pays to start with your own static HTML page, it is *also* advisable to put as much logic in your JavaScript as you can, so you can take it with you if you decide to change your test runner[5].

To set up HTML for each test I usually write a little helper function called `fixture()`. This function adds a `before()` block to the test suite that adds the given HTML to the page, and an `after()` block that removes it. It injects the HTML into an element with ID `fixture`, creating this element if it does not already exist. Some frameworks provide a function like this out of the box, but since `jstest` is platform-agnostic it does not come with any HTML-specific features.

Figure 4.3. `vendor/html_fixture.js`

```
var HtmlFixture = {
  fixture: function(html) {
    this.before(function() {
      var fixture = $("#fixture")

      if (fixture.length === 0) {
        $("body").prepend('<div id="fixture"></div>')
        fixture = $("#fixture")
      }
      this.fixture = fixture
      this.fixture.html(html)
    })

    this.after(function() {
      this.fixture.empty()
    })
  }
}
```

In `jstest` we can add sets of functions to the `describe()` block context using `extend()`. So, when you see `extend(HtmlFixture)` in these examples, that means 'make the `fixture()` function available here'.

The `before()` block created by `fixture()` creates a reference to the element containing the fixture HTML, so you can refer to this element as `fixture` during each test. When selecting elements you should use `fixture.find(selector)` rather than `$(selector)` to make sure you don't accidentally select elements that are part of the test framework's results display. Similarly, JavaScript widgets should use `find()` to only select elements inside the part of the page they are bound to, rather than using `$()` to select elements from any part of the page.

Now we have extracted our JavaScript logic and made a way to set up HTML for tests, we can write our first test suite. These tests use jQuery[6] to fill in the form fields with various values, submit the form

[4]See Section 4.8, "Running browser tests".
[5]Although all the examples in this book use `<script>` tags to load all the test code, in practice I usually use a dynamic script loader so that as much of the setup logic as possible is done in JavaScript. This means that, when I switch test runners, I only need to point the runner at a single JS file, which loads all the other code and tests, rather than duplicating the list of files to load for every runner. I have not done this in this book simply to reduce the conceptual overhead; you should be able to load your code however you like, no matter which test framework and runner you are using. For documentation on loading and running code with `jstest`, see http://jstest.jcoglan.com/browser.html.
[6]http://jquery.com/

by using Syn[7] to click the `submit` button, and check that the right errors are displayed. Note that calling `syn.click($("[type=submit]"))` will trigger the form's `"submit"` event, that is, it triggers the event listeners for the `"submit"` event rather than actually making the browser submit the form. However, unless one of those event listeners cancels the default behaviour for the event, the browser will submit the form and unload the current page, which would abruptly end the test suite.

Figure 4.4. `browser/form_validation/validate_form_spec.js`

```
JS.Test.describe("validateForm()", function() { with(this) {
  extend(HtmlFixture)

  fixture(' <form method="post" action="/" class="test-form"> \
              <p> \
                <label for="email">Email</label> \
                <input type="text" name="email" id="email"> \
              </p> \
              <p> \
                <label for="password">Password</label> \
                <input type="password" name="password" id="password"> \
              </p> \
              <input type="submit" value="Sign up"> \
            </form> ')

  before(function() { with(this) {
    validateForm(fixture.find(".test-form"))
  }})

  it("fails if the email address is invalid", function(resume) { with(this) {
    fixture.find("[name=email]").val("not-an-email")
    fixture.find("[name=password]").val("valid-password")

    syn.click(fixture.find("[type=submit]"), function() {
      resume(function() {
        assertEqual( "Email address is not valid", fixture.find(".error").text() )
      })
    })
  }})

  it("fails if the password is too short", function(resume) { with(this) {
    fixture.find("[name=email]").val("email@example.com")
    fixture.find("[name=password]").val("no")

    syn.click(fixture.find("[type=submit]"), function() {
      resume(function() {
        assertEqual( "Password is too short", fixture.find(".error").text() )
      })
    })
  }})
}})
```

Although short, this is a good example of an approach you can apply to lots of different JavaScript, no matter what framework you're using: create some HTML, bind some JavaScript logic to it, simulate user input (in this case, entering text and submitting a form) and then check what state the DOM is in afterward. jQuery and Syn give you most of the tools you need to do this with very expressive code. This sort of test tells you what the user will see when they perform certain actions.

I've included the fixture HTML inline here, but that's a matter of personal preference; I like the documentation afforded by seeing the HTML a piece of JavaScript interacts with inline with the tests. If you're using JavaScript templates, you might want to reuse your application's templates in your tests, even if they're simply shared strings, for example:

[7]https://github.com/bitovi/syn

Figure 4.5. *`browser/form_validation/templates/form_html.js`*

```
var FORM_HTML = '\
  <form method="post" action="/" class="test-form"> \
    <p> \
      <label for="email">Email</label> \
      <input type="text" name="email" id="email"> \
    </p> \
    <p> \
      <label for="password">Password</label> \
      <input type="password" name="password" id="password"> \
    </p> \
    <input type="submit" value="Sign up"> \
  </form> \
';
```

Figure 4.6. Using a shared HTML string as a fixture

```
fixture(FORM_HTML)
```

If you're using a templating language like Handlebars[8], then you can compile those templates either as a build step[9] before running the tests, or by invoking the templating library directly from the tests themselves. For example, here I've used the `handlebars` executable to compile my templates into a single JavaScript module, which I can then call from my tests to generate HTML.

Figure 4.7. *`browser/form_validation/templates/form_template.handlebars`*

```
<form method="post" action="/" class="{{formClass}}">
  <p>
    <label for="email">Email</label>
    <input type="text" name="email" id="email">
  </p>
  <p>
    <label for="password">Password</label>
    <input type="password" name="password" id="password">
  </p>
  <input type="submit" value="Sign up">
</form>
```

Figure 4.8. Using a Handlebars template as a fixture

```
fixture(Handlebars.templates.form_template({formClass: "test-form"}))
```

Reusing your real application templates in your tests certainly reduces the duplication required, but hard-coding 'duplicated' HTML into the tests does have one upside. It lets you cut out all the incidental complexity by removing markup that deals with copy or presentational aspects, leaving only the elements that the JavaScript needs to interact with. This provides very useful documentation about the integration contract between the HTML and the JavaScript, showing you the minimum HTML you need to start with if you want to reuse a JavaScript module.

On the other hand, basing your rendering on templates means you can write tests for the templates themselves, without coupling to the modules that use them. For example, we could write a test that calls `Handlebars.templates.form_template()` with different input arguments to check it produces the correct HTML. There's no one right way to do this and you should experiment to find the right trade-off for your project.

4.1.3. Separating business logic from the user interface

Although we've tested what `validateForm()` does when the data is invalid, we've not tested what happens when it *is* valid. That's because if the data is valid then we do not call

[8]http://handlebarsjs.com/
[9]See Section 4.7.1, "Precompiling templates".

`event.preventDefault()`, and so the browser submits the form and unloads the current page, abandoning our tests. There is one solution to that that involves mocking the DOM API, but before we get to that let's tackle a more general pattern: separating business logic from presentation.

In our validation example, we perform a series of checks by reading data directly from the DOM, checking it, and then adding an error to the DOM if that check fails. We have intermingled business logic — the notion of what a valid sign-up looks like — with user interface (UI) logic — how the user enters their sign-up data, how errors are displayed to them, and where they are taken on success. The business logic is independent of the presentation layer: the rules for sign-up data would be the same if the validation was being done on the server, or via an API, or using a command-line program.

We can refactor our validation logic so that it performs the following steps:

- Extract all the data from the form into a plain JavaScript object
- Apply validation rules to that object to generate a list of errors
- Display the errors on the page, submitting the form if no errors arise

This chain of actions separates the business logic — the middle step — from the UI logic — the first and last steps — allowing the two concepts to be separated. Expressing business logic in a platform-independent way usually makes it easier to test, since you can test it directly using the core JavaScript data types and functions, rather than poking at it indirectly by simulating user interaction. It means you can make changes to the UI without needlessly breaking the tests for the business logic you have not altered. And in our case, it means we can test the 'valid input' case without causing the page to unload.

Here's a new function that takes an object representing the form data in an abstract way, and returns an object containing the error message for each field, if one exists:

Figure 4.9. `browser/form_validation/validate_signup.js`

```
var validateSignup = function(data) {
  var errors = {}

  if (!data.email.match(/^[a-z]+@[a-z]+\.com$/)) {
    errors.email = "Email address is not valid"
  }
  if (data.password.length < 8) {
    errors.password = "Password is too short"
  }

  return errors
}
```

This function expresses the same rules as `validateForm()` without referencing the DOM in any way. In an MVC framework, this is the logic that belongs in the model: it expresses what the program logically does in an abstract way, without coupling to a particular UI.

It is simpler to test this function, since we don't need to set up HTML for it, or manipulate the DOM. We only need to use plain JavaScript objects and functions, making the tests more terse. As your model grows, you will be glad you wrote and tested it like this instead of doing everything through the UI.

Figure 4.10. *`browser/form_validation/validate_signup_spec.js`*

```
JS.Test.describe("validateSignup()", function() { with(this) {
  before(function() { with(this) {
    this.signup = {
      email:    "validname@example.com",
      password: "a-nice-long-safe-password"
    }
  }})

  it("returns no errors for valid data", function() { with(this) {
    assertEqual( {}, validateSignup(signup) )
  }})

  it("returns an error for an invalid email", function() { with(this) {
    signup.email = "not-valid"
    assertEqual( {email: "Email address is not valid"}, validateSignup(signup) )
  }})

  it("returns an error for an short password", function() { with(this) {
    signup.password = "hi"
    assertEqual( {password: "Password is too short"}, validateSignup(signup) )
  }})
}})
```

This spec demonstrates a common pattern for testing validation: start with a valid object, check that it's valid, and then in each test make one attribute invalid and check that an error arises. You don't need to test every combination of invalid attributes, you only need to check that each independent defect you introduce renders the whole object invalid. You could apply the same approach to our original tests, beginning with valid input and then changing each field in turn to check for errors.

We've also managed to introduce a test for the 'valid data' case, improving the test coverage compared to our original design.

4.1.4. Testing user interface logic

In the previous section, we extracted an important chunk of the logic of `validateForm()` so that it becomes easier to add tests as the validation rules grow. But this does not cover the entire behaviour of the system. Although the majority of the code in many systems ends up as business logic in the model, there is still a significant amount of UI logic that needs testing. The key is figuring out where to draw the line, and deciding what actually needs testing through the UI.

The UI usually forms the 'glue' between the user's interactions and the business logic. In our example, before we call `validateSignup()` we have to extract the data from the form, and when validation is complete we need to display the results and decide whether to let the submission proceed.

Notice that these actions are independent of what the validation rules actually are, so we can write this logic in a generic way. We need a class that can extract the data from a form (for which we can lean on `jQuery.serializeArray()`), hand the data to a validator function, display any errors that function returns, and decide whether to make the browser submit the form to the server.

The code for this `FormValidator` class is listed below; its constructor takes a reference to a `form` and a `validator` function, and it applies the given validation rules to the form. Rather than conditionally calling `event.preventDefault()`, we call `event.preventDefault()` in *all* cases and then conditionally call `form.submit()` if the data is valid. Calling `submit()` on a bare DOM element (rather than triggering a `"submit"` event via jQuery or Syn) forces the browser to submit the form, bypassing any event listeners. Using this approach will let us test the 'valid data' case using mocking, as we shall see.

Figure 4.11. `browser/form_validation/form_validator.js`

```
var FormValidator = function(form, validator) {
  this._form      = $(form)
  this._validator = validator

  var self = this

  this._form.on("submit", function(event) {
    event.preventDefault()
    self.handleSubmit()
  })
}

FormValidator.prototype.handleSubmit = function() {
  var data = this.getData(), errors = this._validator(data)

  this._form.find(".error").remove()
  this.displayErrors(errors)

  if (this._form.find(".error").length === 0) {
    this.submit()
  }
}

FormValidator.prototype.getData = function() {
  var array = this._form.serializeArray(), params = {}

  for (var i = 0, n = array.length; i < n; i++) {
    params[array[i].name] = array[i].value
  }
  return params
}

FormValidator.prototype.displayErrors = function(errors) {
  for (var key in errors) {
    this._form.find("[name=" + key + "]")
              .parents("p")
              .after('<p class="error">' + errors[key] + '</p>')
  }
}

FormValidator.prototype.submit = function() {
  this._form.get(0).submit()
}
```

Testing a UI class like this means testing that it successfully glues the user's interaction with the page up to the business logic. This means checking that it extracts the data from the form correctly and passes it to the `validator` function, that when the validator returns no errors the form is submitted, and that when the validator returns errors that they are displayed rather than submitting the form.

We've removed the need to test every case in the data validation logic and reduced it to two broad behaviour classes: what happens when the data is valid or invalid. This is the only difference the `FormValidator` class cares about, it doesn't know anything about the validation rules themselves.

We can test that `FormValidator` passes the form data to the validator correctly by using mocking. `expect(this, "validator").given({…})` creates a function called `validator` within the current test and sets up an expectation about what it should be called with. We can then fill in some data, submit the form, and check that the validator is indeed invoked as expected. This covers the data input responsibility of the class.

To check the output display responsibilities, we can use `stub(this, "validator").returns({…})` to create validator functions with canned responses, and pass those into `new FormValidator()`. We can then trigger the `"submit"` event listeners, and check that the validator's output is displayed to the user.

In the case where no errors are emitted, we want to check that the form is submitted for real. Since we're implementing this by calling `form.submit()` on the DOM `<form>` element, we can again use mocking: `expect(form, "submit").exactly(1)` in the success case and `expect(form, "submit").exactly(0)` in the failure case. Since mocking stubs out the mocked function with a fake, the browser won't actually submit the form and our tests will not be suddenly abandoned; to ensure this doesn't happen even in tests with no mock expectations, we call `stub(form, "submit")` at the start of all tests.

Figure 4.12. `browser/form_validation/form_validator_spec.js`

```
JS.Test.describe("FormValidator", function() { with(this) {
  extend(HtmlFixture)
  fixture(FORM_HTML)

  before(function() { with(this) {
    this.form = fixture.find("form").get(0)
    stub(form, "submit")
  }})

  it("extracts form data and passes it to the validator", function(resume) { with(this) {
    fixture.find("[name=email]").val("james@example.com")
    fixture.find("[name=password]").val("something")

    expect(this, "validator").given({email: "james@example.com", password: "something"})
    new FormValidator(form, validator)
    syn.click(fixture.find("[type=submit]"), resume)
  }})

  describe("when the validator returns no errors", function() { with(this) {
    before(function() { with(this) {
      stub(this, "validator").returns({})
      new FormValidator(form, validator)
    }})

    it("submits the form", function(resume) { with(this) {
      expect(form, "submit").exactly(1)
      syn.click(fixture.find("[type=submit]"), resume)
    }})
  }})

  describe("when the validator returns an error", function() { with(this) {
    before(function() { with(this) {
      stub(this, "validator").returns({email: "example.com addresses are not allowed"})
      new FormValidator(form, validator)
    }})

    it("doesn't submit the form", function(resume) { with(this) {
      expect(form, "submit").exactly(0)
      syn.click(fixture.find("[type=submit]"), resume)
    }})

    it("displays the error", function(resume) { with(this) {
      syn.click(fixture.find("[type=submit]"), function() {
        resume(function() {
          assertEqual( "example.com addresses are not allowed",
                       fixture.find(".error").text() )
      })})
    }})
  }})
}})
```

4.1.5. Simulating user input

You might be wondering why we're using this library called Syn, when it seems like everything it does can be done using jQuery. For example, `syn.click(element)` looks just like `$(element).click()`, and jQuery doesn't need that ugly callback function. Well, the difference is that calling `$(element).click()` *only* triggers a `"click"` event on `element`, and nothing else, whereas `syn.click(element)` triggers *all* the events that a real user click would do. When you click a link or form element in a browser, `"click"` is just one of many events that fires. First, the clicked element emits a `"mousedown"` event. Next, the previously focussed form element emits `"change"` if its contents were edited since it gained focus, and then it emits `"blur"`. The newly clicked element then emits `"focus"` and gains cursor focus, then emits `"mouseup"` followed by `"click"`. The `"mousedown"`, `"mouseup"` and `"click"` events bubble up the DOM, being emitted by the containing `<form>` element and so on up the tree, while the `"blur"`, `"change"` and `"focus"` events do not bubble. Using `syn.click()` simulates a real user action, and emits all the above events rather than just `"click"`.

In the above examples, we've used `element.val("some text")`, the jQuery API for setting the value of a form input. This doesn't simulate real user input at all since it does not emit any events. When a real person types into a text field, each key they press makes the `<input>` emit `"keydown"`, `"keypress"`, `"input"` and `"keyup"` events. If we used `syn.type(element, "some text")` instead, then the DOM would emit events as though someone was really typing into the field.

Now, in some situations this distinction doesn't matter. Our form validator only cares about the state of the DOM at the time when the form is submitted; how it got into that state is not important and so we can use jQuery to quickly set up the desired set of inputs. However, say that instead of validating the entire form on submission, we validated each field whenever its data was altered. The code might look like this:

Figure 4.13. `browser/form_validation/interactive_validator.js`

```
var InteractiveValidator = function(form, validator) {
  this._form      = $(form)
  this._validator = validator

  var self = this

  this._form.find("input").each(function(index, input) {
    if (input.name) self.validateInput(input)
  })
}

InteractiveValidator.prototype.validateInput = function(input) {
  var name = input.name, self = this

  $(input).on("keyup", function() {
    var data = self.getData(), errors = self._validator(data)
    if (!errors[name]) return
    self._form.find(".error-" + name).remove()
    $(input).parents("p")
        .after('<p class="error error-' + name + '">' + errors[name] + '</p>')
  })
}

InteractiveValidator.prototype.getData = function() {
  var array = this._form.serializeArray(), params = {}

  for (var i = 0, n = array.length; i < n; i++) {
    params[array[i].name] = array[i].value
  }
  return params
}
```

Since this validator triggers validation whenever the user presses a key, by binding to the `"keyup"` event, it won't react if our test enters the data using `jQuery.val()`. We *could* follow the call to `val()` with `jQuery.trigger("keyup")`, but it's best not to get into simulating browser behaviour yourself since it's rather complex and full of edge cases and vendor differences. It's best to use Syn, which already knows how to accurately simulate most user-triggered events for you.

The following passing test suite demonstrates the difference between using jQuery and using Syn to enter form data.

Figure 4.14. `browser/form_validation/interactive_validator_spec.js`

```
JS.Test.describe("InteractiveValidator", function() { with(this) {
  extend(HtmlFixture)
  fixture(FORM_HTML)

  before(function() { with(this) {
    stub(this, "validator").returns({password: "Password is too short"})
    new InteractiveValidator(fixture.find("form"), validator)
  }})

  it("triggers validation using type()", function(resume) { with(this) {
    syn.type(fixture.find("[name=password]"), "hi", function() {
      resume(function() {
        assertEqual( 1, fixture.find(".error").length )
        assertEqual( "Password is too short", fixture.find(".error").text() )
      })
    })
  }})

  it("doesn't trigger validation using val()", function() { with(this) {
    fixture.find("[name=password]").val("hi")
    assertEqual( 0, fixture.find(".error").length )
  }})
}})
```

There are also differences in the browser behaviour that jQuery and Syn can trigger. For example, the default browser response to a link being clicked is to redirect to the link's `href`. However, calling `$("a").click()` does not trigger this behaviour. If your code needs to prevent a default browser behaviour, for example by loading some content via Ajax instead of letting the browser follow the link, it helps if your tests can catch places where you've forgotten to do this. Using `jQuery.click()` to follow links won't trigger this behaviour and so won't warn you that your code is broken, whereas `syn.click()` *does* trigger the default behaviour, and your test page will be adandoned if your event listeners don't call `event.preventDefault()`.

But Syn is not perfect. For example, it does not emit `"input"` or `"change"` events when form fields are edited, so you may still need to use jQuery to trigger those yourself. If you run into problems, check which events Syn is actually emitting, check what events fire when you interact with the page for real, and fill in any gaps in the test behaviour yourself.

4.2. Modular interfaces

In Section 4.1, "Form validation", we explored how to test a single user interface module, a class that knows how to validate forms and give feedback to the user. This example is fine, as far as it goes, but real applications are usually much more complicated. Each page has many different UI components that behave in different ways, and that often interact with one another. We need an approach to building and testing such applications that stops the complexity becoming unmanageable.

With the following example, we'll see how to keep complexity under control by keeping our codebase modular; rather than coupling UI components together in the code, we will keep them as ignorant of each other as possible and find clean ways for them to communicate where necessary. By keeping UI components decoupled from one another, we retain the ability to test them easily *and* we are able to reuse our components more freely in different contexts.

Let’s say we’re building a to-do list app. There’s a `<form>` for the user to enter new to-do items with a `title` and a `body`, and there’s a `<table>` that will display a list of the user’s items and let them load, edit and delete them.

Figure 4.15. `browser/todo_list/demo.html`

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Todo List</title>
  </head>
  <body>

    <div class="todo-form">
      <p><a class="new-todo" href="/todos/new">Create item</a></p>
      <form method="post" action="/todos">
        <input type="hidden" name="id">
        <p>
          <label for="todo-title">Title</label>
          <input type="text" name="title" id="todo-title">
        </p>
        <p>
          <label for="todo-body">Body</label><br>
          <textarea name="body" id="todo-body"></textarea>
        </p>
        <input type="submit" value="Save">
      </form>
    </div>

    <table class="todo-list">
      <thead>
        <tr>
          <th scope="col">Title</th>
          <th scope="col">Actions</th>
        </tr>
      </thead>
      <tbody></tbody>
    </table>

    <script src="../../node_modules/jquery/dist/jquery.js"></script>
    <script src="../../node_modules/handlebars/dist/handlebars.runtime.js"></script>
    <script src="../form_validation/form_validator.js"></script>
    <script src="./lib/todo_form.js"></script>
    <script src="./lib/todo_list.js"></script>
    <script src="./templates/templates.js"></script>

    <script>
      var App = {
        todoForm: new TodoForm(".todo-form"),
        todoList: new TodoList(".todo-list")
      }
    </script>

  </body>
</html>
```

This page contains the HTML markup for the data-entry form and the table of to-do items as described above, and ends with a few scripts. We load jQuery, Handlebars and our `FormValidator` class from Figure 4.11, “`browser/form_validation/form_validator.js`”, followed by the `TodoForm` and `TodoList` JavaScript classes, one class for each UI component. Finally we create a global variable called `App` that contains an instance of each of these classes. This means the form component can be referenced globally as `App.todoForm`, and the list as `App.todoList`; this global state will be necessary for our initial design but in later iterations we will seek to remove it.

4.2.1. Directly communicating components

Our first attempt at a design for this application has a JavaScript class for each UI component on the page. Each class takes a CSS selector as input to tell it which component to bind to, and then sets up the dynamic behaviour of the component by listening to events and updating the DOM. All these are reasonable steps toward keeping the components testable: since each class has a focussed responsibility rather than trying to manage the state of the entire page, we can test each of them using a small amount of fixture HTML, and changing the behaviour or markup of one component won’t force us to update a lot of unrelated tests.

Let’s have a look at the code for these classes and examine them in more detail, starting with `TodoForm`.

Figure 4.16. `browser/todo_list/lib/todo_form.js`

```
var TodoForm = function(selector) {
  this._element = $(selector)
  this.loadItem({})

  var form      = this._element.find("form"),
      validator = new FormValidator(form, this.validateItem),
      self      = this

  this._element.find(".new-todo").on("click", function(event) {
    event.preventDefault()
    self.loadItem({})
  })

  validator.submit = function() {
    $.extend(self._todo, validator.getData())
    self.saveItem()
  }
}

TodoForm.prototype.loadItem = function(item) {
  this._todo = item
  var self = this

  $.each(["id", "title", "body"], function(i, field) {
    self._element.find("[name=" + field + "]").val(item[field] || "")
  })
}

TodoForm.prototype.validateItem = function(item) {
  var errors = {}
  if (!item.title) errors.title = "Title cannot be blank"
  return errors
}

TodoForm.prototype.saveItem = function() {
  App.todoList.saveItem(this._todo)
  this._element.find("[name=id]").val(this._todo.id)
}
```

The `TodoForm` class finds the right element to bind to using the given CSS selector, then sets its initial state by calling `this.loadItem({})`. This loads an empty to-do item (represented by an empty object) into the form, emptying all of its input fields. It also binds an event listener to the `Create item` link, making a click on this link load a new item into the form.

`TodoForm` also reuses `FormValidator` to handle validation logic, but overrides the validator's `submit()` method. Rather than actually submitting the form and causing a page refresh, we want to save the current item by adding it to the table, or updating an existing table row. So, we make the `submit()` method update the current to-do item with the data from the form, and then we call `App.todoList.saveItem(this._todo)` to send the item off to the `TodoList` we created in `demo.html`. This is why we need to store a global reference to the `TodoList`; the form needs to be able to tell the list that an item was just saved so the list can update itself.

Now let's take a look at the `TodoList` class, which manages the state of the `<table>` displaying the to-do items. This class is a little more complex, but again each function will be explained after the source code.

Figure 4.17. `browser/todo_list/lib/todo_list.js`

```
var TodoList = function(selector) {
  this._element = $(selector)
  this._itemId  = 0
  this._items   = {}
  this._rows    = {}
}

TodoList.prototype.saveItem = function(item) {
  item.id = item.id || ++this._itemId

  var itemHtml = this._rows[item.id]
  if (!itemHtml) {
    itemHtml = this._rows[item.id] = this.renderItem(item)
  }
  this._items[item.id] = item
  itemHtml.find(".todo-title a").text(item.title)
}

TodoList.prototype.renderItem = function(item) {
  var itemHtml = $(Handlebars.templates.todo_item(item)), self = this
  this._element.find("tbody").append(itemHtml)

  itemHtml.find("a").on("click", function(event) {
    event.preventDefault()
    self.loadItem(item.id)
  })

  itemHtml.find(".delete-form").on("submit", function(event) {
    event.preventDefault()
    self.removeItem(item.id)
  })

  return itemHtml
}

TodoList.prototype.loadItem = function(itemId) {
  App.todoForm.loadItem(this._items[itemId])
}

TodoList.prototype.removeItem = function(itemId) {
  this._rows[itemId].remove()
}
```

Just like `TodoForm`, `TodoList` takes a CSS `selector` that it uses to find the elements it should bind to. It also has a few other bits of internal state; since the form calls `TodoList.saveItem()` this class needs to decide whether incoming items are new or not, so it can either add them as a new row or update an existing one. So, it has an internal counter called `this._itemId` that it uses to assign new IDs, and it has a list of items in `this._items` that stores the data for each item. It also keeps a reference to the table row for each item in the `this._rows` object, so it can find the right row to update when an item changes.

When we call `TodoList.saveItem(item)`, it first checks whether the item has an ID and assigns one if necessary. It then retrieves the table row for that item ID, creating a new row and inserting it if necessary. And finally, it updates the information displayed by the row based on the item's data, in this case updating the link title that displays the name of the item.

The method `TodoList.renderItem()` deals with generating new table rows and binding event listeners to them. It uses this Handlebars template to generate a new row for new items:

Figure 4.18. *`browser/todo_list/templates/todo_item.handlebars`*

```
<tr>
  <td class="todo-title">
    <a href="/todos/{{id}}">{{title}}</a>
  </td>
  <td class="todo-actions">
    <form class="delete-form" method="post" action="/todos/{{id}}/delete">
      <input type="hidden" value="{{id}}">
      <input type="submit" value="Delete">
    </form>
  </td>
</tr>
```

After creating the HTML, it binds two event listeners: one to load an item into the form for editing when its title is clicked, and one to remove the item from the list when its `Delete` button is clicked. These events are dispatched to the `TodoList.loadItem()` and `TodoList.removeItem()` methods respectively.

`TodoList.removeItem()` is fairly trivial; it simply looks up the table row for the given `itemId` and removes it from the DOM. But `TodoList.loadItem()` is another source of coupling: it refers to the global variable `App.todoForm` since it needs to tell the form to load up the data for an item so it can be edited.

Now, this isn't a *huge* problem. It means that to test `TodoForm` we need to create a global object called `App.todoList`, and create `App.todoForm` to test `TodoList`. These globals don't need to be instances of the real thing, they just need to be *some* object that we can stub or mock methods on. We don't need to set up markup for the entire application either, since the components communicate via APIs rather than messing with one another's DOM trees. This fact on its own — making each component responsible for some sub-region of the DOM and not allowing components to access the DOM outside their particular region — removes a huge source of coupling problems as the app grows, and makes it easier to see the design of the system and reason about how it works.

It is possible to stub out global objects, creating fakes for components to interact with. If we want `App.todoList` to refer to an object, we can say `window.App = {todoList: {}}` and we're done. In tests, though, we don't want our fakes and their state to hang around after the end of each test, so we'd need to call `delete window.App` afterward. The `jstest` stubbing API provides for this use case: `stub("App", {todoList: {}})` creates a global variable called `App` whose value is `{todoList: {}}`, and it makes sure `App` is deleted after the test.

So, we could test `TodoForm` as follows, setting up the HTML for it, creating a fake `App.todoList` object, and checking that when the user interacts with the form it sends the correct commands to `App.todoList`. For example, this test checks that when the user enters the title `"Buy milk"` and clicks `Save`, then `TodoForm` calls `App.todoList.saveItem()` with the form data.

Figure 4.19. `browser/todo_list/spec/todo_form_spec.js`

```
JS.Test.describe("TodoForm", function() { with(this) {
  extend(HtmlFixture)
  fixture(FORM_HTML)

  before(function() { with(this) {
    stub("App", {todoList: {}})
    new TodoForm(fixture.find(".todo-form"))
  }})

  it("saves a new item", function(resume) { with(this) {
    expect(App.todoList, "saveItem").given({id: "", title: "Buy milk", body: ""})
    fixture.find("[name=title]").val("Buy milk")
    syn.click(fixture.find("[type=submit]"), resume)
  }})

  // and so on for other cases ...
}})
```

This approach is fine for the current problem, but tends to scale very badly. Most apps have many UI components that all need to coordinate with one another somehow, depending on the state of the user's data, which page they are currently on, communication with the server, and all sorts of other inputs. Assigning each component to a global produces a network of couplings that make it hard to understand the app's structure, and almost impossible to tease apart for testing.

4.2.2. Dependency injection

Dependency injection is the technique of removing hard-coded dependencies from software components and allowing them to be changed dynamically. In practice, one application of this technique is to replace references to global variables with *local* variables, by passing the things a function or object depends on in as arguments.

In our example, `TodoForm` has a hard-coded dependency on `App.todoList`: if this variable does not point to an object with a `saveItem()` method, then `TodoForm` will not work correctly. But, in the same way we pass in the CSS selector for the component to bind to, we can pass in references to other components it depends on, letting us pass in the real object in production and a fake object in our tests.

In other words, instead of `TodoForm` looking like this:

Figure 4.20. `TodoForm` with a hard-coded dependency

```
var TodoForm = function(selector) {
  this._element = $(selector)
  // ...
}

TodoForm.prototype.saveItem = function(item) {
  App.todoList.saveItem(item)
  // ...
}
```

we can pass a reference to a `TodoList` into the constructor, and `TodoForm` can store this reference and refer to it in its methods:

Figure 4.21. `TodoForm` with an injected dependency

```
var TodoForm = function(selector, todoList) {
  this._element  = $(selector)
  this._todoList = todoList
  // ...
}

TodoForm.prototype.saveItem = function(item) {
  this._todoList.saveItem(item)
  // ...
}
```

This eliminates a global variable and makes it a bit more convenient to test things. Instead of stubbing some global API, we could make a fake object in our tests, pass it into the constructor, and check how it's called, for example:

Figure 4.22. Using mocks to check the use of an injected dependency

```
before(function() { with(this) {
  this.todoList = {}
  new TodoForm(fixture.find(".todo-form"), todoList)
}})

it("saves a new item", function(resume) { with(this) {
  expect(todoList, "saveItem").given({id: "", title: "Buy milk", body: ""})
  fixture.find("[name=title]").val("Buy milk")
  syn.click(fixture.find("[type=submit]"), resume)
}})
```

This doesn't seem to buy us a lot in this small example, but what this construction *does* do is make you more aware of the couplings between components. If you avoid global variables and are disciplined about always passing dependencies into the objects that need them, the code you need to write to pass those objects around will be a reminder of the couplings in the system. It makes the design more explicit, since you can see which objects a class depends on by looking at its constructor, and you don't need to be afraid of the whole house of cards falling over if you change the name of a global variable.

Dependency injection might work for `TodoForm`, but how do we fix `TodoList` in the same way? The two components need to talk to one another, but we can't pass each one to the other's constructor: this creates a chicken-and-egg situation where whichever one we pick to construct, we need to construct an instance of the other one first so we can inject it.

Whenever you encounter a circular dependency, there's a temptation to roll the two interdependent components back into a single component to resolve the problem. But this can often multiply the complexity of the resulting solution, making it harder to understand. Especially in UI code, following the network of dependencies and merging things together can lead to one monolithic object that runs the entire application and that everybody is scared of changing. A better solution is to break the circle by having both components depend on some third object, rather than on one another. In many applications, this third object ends up being the *model*.

4.2.3. Introducing a model

A *model* is just that: a model, or a simulation, of the problem your app is designed to solve. In many development frameworks, the classes that represent the entities in your domain — users, to-do lists, items, and so on — are called ‘the models’, and in some frameworks those model classes map directly onto database tables or REST resources. But really it's *all* of those classes and any other functions that represent your business logic that *collectively* form your application's model. A model doesn't have to represent each entity in the system as a class; the model could be a single flat object with methods that cover all of the app's responsibilities.

> Remember that the job of your model layer is not to represent objects but to answer questions. Provide an API that answers the questions your application has, as simply and efficiently as possible.
>
> — Laurie Voss *http://seldo.com/weblog/2011/08/11/orm_is_an_antipattern*

Essentially, the model is your app's business logic, that exists independently of any UI. For example, the notion of a list of to-do items, each with a title and body, is an abstract concept that can exist unchanged no matter how the UI represents it. In our example, a to-do item appears in the table as only its title, but when loaded into the editing form its body is displayed as well.

Rather than communicate with one another directly, it is often useful for UI components to communicate instead with a model, which tracks the abstract state of the application and coordinates change. One component can save a change to the model, and the model can inform other components of this change so that they can update what they display to the user. This architecture is incredibly common and you will see variations of it in all the popular JavaScript MVC frameworks. Some make it explicit, some implicit, but all of them have it.

You'll notice that our example has some model logic baked into it already: the `TodoList` class not only manages the information displayed in the table, but also stores the list of to-do items containing their data. It knows how to 'save' an item and assign IDs to newly created ones. This all sounds like abstract data-processing logic, not UI code, and we can extract it into a model.

Let's rethink our UI code, writing as though a model already exists. We will write UI code that doesn't do any data management itself, but assumes that some model object will deal with that for us. Writing our code this way will help us discover what API this model will need. We will still consider each UI component in isolation, and consider what it needs from the model layer.

Starting with `TodoForm`, we know that it needs to be able to save the data in the form, and it needs to be able to load existing items into the form for editing. So, the model will need a method to let us save items. It will also need to be able to tell the form to load an item, but rather than make the model depend on the `TodoForm` API, we can achieve this by having the model emit an event that the form listens to.

Here's a variation on our `TodoForm` class, called `TodoEditor`. I've subclassed `TodoForm` and changed a couple of things to adapt it to our new architecture[10]. The constructor new takes a `model` as an argument — applying the techniques in Section 4.2.2, "Dependency injection" — and listens for a `"load"` event to tell it to load an item for editing. When the user clicks `Save`, `TodoEditor` calls `model.saveItem()` to send the item data down the model layer.

Here's the code for `TodoEditor`:

[10]I am only using subclassing here so that I can limit the example code to showing the things I've changed about `TodoForm`, rather than reproducing the whole class. Also, introducing a new name makes it easier for me to refer to the different versions of the solution. There's nothing about this problem that demands you use inheritance; in practice I would modify my existing class.

Figure 4.23. `browser/todo_list/lib/todo_editor.js`

```
var TodoEditor = function(selector, model) {
  TodoForm.apply(this, arguments)
  this._model = model

  var self = this

  this._model.on("load", function(item) {
    self.loadItem(item)
  })
}
util.inherits(TodoEditor, TodoForm)

TodoEditor.prototype.saveItem = function(item) {
  this._model.save(this._todo)
  this._element.find("[name=id]").val(this._todo.id)
}
```

Now we're in a position to test `TodoEditor` in isolation from other UI components. All we need is a fake `model` object to interact with, which needs to respond to the `save()` method and be capable of emitting a `"load"` event. The `save()` method can be put in place using stubbing or mocking when needed, but for emitting events I find it easier to use an actual event emitter[11]. There are many, many implementations of event emitters for the browser, and it doesn't particularly matter which one you pick, but I'm going with the widely-used `Backbone.Events` implementation from Backbone[12]. `Backbone.Events` is an object rather than a class, and we shouldn't attach event listeners directly to it since that creates state we'll need to clean up between tests. It's easier to use `_.clone(Backbone.Events)` to make a copy of it, giving us a fresh event emitter in each test.

We need to consider which things to write tests for. When testing UI code, you'll often be writing two basic kinds of test: first, when the user does something, what changes in the UI and what does the UI tell the model to do; and second, when the model changes in some way, what changes does the user see in the UI. You can think of the UI as a bridge between the user's intent and the model, and we need to test that it wires those together correctly.

In the case of `TodoEditor`, a few things spring to mind. First, we should check that if the user enters invalid data, the editor does not tell the `model` to save the item, but displays an error instead. We can use mocking to check that `model.save()` is never called in this scenario. When the data *is* valid, we can again use mocking to check that `model.save()` is called with the data from the form, when the `Save` button is clicked.

We also need to check that the editor can change an *existing* item; when the `model` emits a `"load"` event, and then the user changes one of the fields in the form and submits it, then `model.save()` should be called with the same data emitted by `"load"`, but incorporating the change the user made. I've written separate tests for changing the `title` and changing the `body` below.

And finally, we need to check that the `Create item` link works. That is, if the `model` emits `"load"`, and the user then clicks `Create item`, adds a `title` and clicks `Save`, then the data supplied by the `"load"` event should have been discarded and `model.save()` should receive only the new data the user typed in, including a blank `id` value.

The tests for all these scenarios are listed below. They check that data emitted by `"load"` is correctly incorporated into the form, and that data the user enters is sent back to the `model` as expected.

[11]See Chapter 3, *Events and streams*.
[12]http://backbonejs.org/

Figure 4.24. *browser/todo_list/spec/todo_editor_spec.js*

```
JS.Test.describe("TodoEditor", function() { with(this) {
  extend(HtmlFixture)

  fixture(' <div class="todo-form"> \
              <p><a class="new-todo" href="/todos/new">Create item</a></p> \
              <form method="post" action="/todos"> \
                <input type="hidden" name="id"> \
                <p> \
                  <label for="todo-title">Title</label> \
                  <input type="text" name="title" id="todo-title"> \
                </p> \
                <p> \
                  <label for="todo-body">Body</label><br> \
                  <textarea name="body" id="todo-body"></textarea> \
                </p> \
                <input type="submit" value="Save"> \
              </form> \
            </div> ')

  before(function() { with(this) {
    this.model  = _.clone(Backbone.Events)
    this.editor = new TodoEditor(fixture.find(".todo-form"), model)
  }})

  it("does not save an invalid item", function(resume) { with(this) {
    expect(model, "save").exactly(0)
    syn.click(fixture.find("[type=submit]"), function() {
      resume(function() {
        assertEqual( "Title cannot be blank", fixture.find(".error").text() )
      })
    })
  }})

  it("saves a new item", function(resume) { with(this) {
    expect(model, "save").given({id: "", title: "Buy milk", body: ""})
    fixture.find("[name=title]").val("Buy milk")
    syn.click(fixture.find("[type=submit]"), resume)
  }})

  it("edits an existing item's title", function(resume) { with(this) {
    expect(model, "save").given({id: "42", title: "Discover meaning of life", body: ""})
    model.trigger("load", {id: 42, title: "Hello, world", body: ""})
    fixture.find("[name=title]").val("Discover meaning of life")
    syn.click(fixture.find("[type=submit]"), resume)
  }})

  it("edits an existing item's body", function(resume) { with(this) {
    expect(model, "save").given({id: "64", title: "Rent", body: "BY FRIDAY"})
    model.trigger("load", {id: 64, title: "Rent", body: ""})
    fixture.find("[name=body]").val("BY FRIDAY")
    syn.click(fixture.find("[type=submit]"), resume)
  }})

  it("creates a new item after editing", function(resume) { with(this) {
    expect(model, "save").given({id: "", title: "Deploy to Heroku", body: ""})
    model.trigger("load", {id: 64, title: "Rent", body: ""})
    syn.click(fixture.find(".new-todo"), function() {
      fixture.find("[name=title]").val("Deploy to Heroku")
      syn.click(fixture.find("[type=submit]"), resume)
    })
  }})
}})
```

Just as we migrated `TodoForm` to `TodoEditor`, we can migrate `TodoList` to talk to the `model` rather than to other parts of the UI. `TodoList` contained a lot of logic for storing and retrieving to-do item data, and all these concerns can be completely delegated to the `model`. Just as before, I've created a subclass of `TodoList` called `TodoDisplay` that implements this migration; just like `TodoEditor` it accepts `model` as a constructor parameter.

`TodoDisplay`, as its name suggests, is mostly responsible for displaying the current state of the to-do list. It does this *reactively*, by listening to events emitted by the `model` that tell it when to-do items are created, updated, and removed. These events allow the `model` to *tell* the `TodoDisplay` about the current state, instead of `TodoDisplay` needing to *ask*. The `model` can do this without needing to know anything about how `TodoDisplay` works, or even that it exists at all. When an item is created, we render a new table row for it. When an item is updated, we find its row and update the link text therein. And when an item is removed, we remove its row from the table. These events define the relationship between the state of the model and what is displayed to the user.

The `loadItem()` and `removeItem()` methods, which respectively handle clicks on the item title links and clicks on each row's `Delete` button, have been overridden to simply delegate their functionality to the `model`. Notice that `removeItem()` does *not* remove the item's row from the table — that happens in the `"remove"` event listener. This means that if some other part of the UI calls `model.remove()`, `TodoDisplay` will always be notified and the change will be reflected in the table. This class clearly illustrates the two directions of data flow in the app: user interaction sends commands to the `model`, and changes to the `model` update the UI. Structuring your code like this makes sure that the UI always consistently reflects the true state of the underlying data.

Figure 4.25. `browser/todo_list/lib/todo_display.js`

```
var TodoDisplay = function(selector, model) {
  TodoList.apply(this, arguments)
  this._model = model

  var self = this

  this._model.on("create", function(item) {
    var row = self.renderItem(item)
    self._rows[item.id] = row
    self._element.find("tbody").append(row)
  })

  this._model.on("update", function(item) {
    self._rows[item.id].find("a").text(item.title)
  })

  this._model.on("remove", function(item) {
    self._rows[item.id].remove()
    delete self._rows[item.id]
  })
}
util.inherits(TodoDisplay, TodoList)

TodoDisplay.prototype.loadItem = function(itemId) {
  this._model.load(itemId)
}

TodoDisplay.prototype.removeItem = function(itemId) {
  this._model.remove(itemId)
}
```

This class is actually a little simpler to test than `TodoEditor`, since most of its responsibility is reacting to data changes rather than dealing with user interaction. We can check on the state of the table by grabbing the list of link titles out of it, calling `fixture.find("a")` and invoking `text()` on each link. The tests start by making the `model` emit `"create"` and checking that that adds a single row to the table. Then we emit `"update"` with an item with the same `id`, and check there is still only one row but with different text.

Again, we use mocking to check that when the link titles and `Delete` buttons are clicked, we call `model.load()` or `model.remove()` with the right `id`. And finally, we check that when the `model` emits `"remove"` with the `id` of the loaded item, it vanishes from the table.

Figure 4.26. `browser/todo_list/spec/todo_display_spec.js`

```
JS.Test.describe("TodoDisplay", function() { with(this) {
  extend(HtmlFixture)

  fixture(' <table class="todo-list"> \
              <thead> \
                <tr> \
                  <th scope="col">Title</th> \
                  <th scope="col">Actions</th> \
                </tr> \
              </thead> \
              <tbody></tbody> \
            </table> ')

  before(function() { with(this) {
    this.model   = _.clone(Backbone.Events)
    this.display = new TodoDisplay(fixture.find(".todo-list"), model)

    model.trigger("create", {id: 37, title: "Pay rent"})
  }})

  it("displays the item", function() { with(this) {
    var titles = $.map(fixture.find("a"), function(a) { return $(a).text() })
    assertEqual( ["Pay rent"], titles )
  }})

  it("updates the item", function() { with(this) {
    model.trigger("update", {id: 37, title: "Submit tax return"})
    var titles = $.map(fixture.find("a"), function(a) { return $(a).text() })
    assertEqual( ["Submit tax return"], titles )
  }})

  it("loads an item when clicked", function(resume) { with(this) {
    expect(model, "load").given(37)
    syn.click(fixture.find("tbody a"), resume)
  }})

  it("tells the model to remove an item", function(resume) { with(this) {
    expect(model, "remove").given(37)
    syn.click(fixture.find("tbody .delete-form [type=submit]"), resume)
  }})

  it("removes a deleted item from view", function() { with(this) {
    model.trigger("remove", {id: 37})
    var titles = $.map(fixture.find("a"), function(a) { return $(a).text() })
    assertEqual( [], titles )
  }})
}})
```

Notice the two directions of control appearing again: when we test user interaction, we use mocking to check that the right model methods are called. When we trigger a change in the model, we query the DOM to check what's displayed to the user. We start every test with the DOM in a known state, and then make little changes to the model to check what effects they have. Every type of change is tested on its own, keeping the tests cleanly separated and easy to reason about.

4.2.4. Building and testing the model

This business of writing tests against fake APIs that don’t exist yet often seems strange when people first encounter it. The tests pass, but the app still doesn’t work. Isn’t that wrong? It’s not *wrong*, it’s just that our work is not finished yet. But, building the app this way, starting from the user interface and imagining what the underlying APIs should be, is a good way of discovering the requirements of those underlying APIs. Should a piece of functionality be provided as a method call or as an event? In which direction is information flowing? Which functions need to be accessed by multiple UI components? Where should validation logic live? These are all questions that you can answer by building some clients for the API you’re trying to design.

From the `TodoEditor` and `TodoDisplay` classes we’ve built, we’ve learned the following things about the `model` object:

- It needs a `save()` method that takes an object containing to-do item data. If the item is new it should be assigned a unique `id`.
- After `save()` is called, it should emit a `"create"` or `"update"` event, depending on whether the item is new or not.
- It needs a `load()` method that takes an `id`, and should emit a `"load"` event with the full data for the identified item.
- It needs a `remove()` method that also takes an `id`, and should emit a `"remove"` event to notify clients the item no longer exists.
- After a call to `remove()`, calling `load()` with the affected `id` should no longer emit a `"load"` event.

The behaviour of the `load()` method might seem a little strange, for example you might expect that `load()` would *return* the data for the given `id`, rather than emit an event. What does it mean for the data model to emit a `"load"` event anyway? Isn’t an item being ‘loaded’ a property of the UI, rather than the state of the domain data?

This is an important point that often gets overlooked. The ‘model’ does not have to only be a model of your application’s business data, its state and behaviour. You can also model a user interface: the `TodoEditor` and `TodoDisplay` are *models* of the behaviour of the interfaces they manage. But we can also model the UI in more abstract ways: the notion of an item being ‘loaded’ is a fact that we want to communicate between two components without coupling them together, and we can use a model object for this. The model object could be a simple event emitter that ferries messages between components, or it could model that state of the UI or the business data or both, depending on your needs. You can combine and layer all these approaches to best suit the problem you’re trying to solve.

Having an abstract model of the UI, whether that’s an event system or a set of ‘presenter’ objects that define what the user can see in abstract terms, without getting into the details of the DOM, can be very useful for testing the logic of a program without going through the complexity of actually simulating user interaction and DOM logic.

But returning to the problem at hand, we have a set of requirements for a `TodoModel` object that we’ve not built yet. At this point, we can turn our workflow around and write the tests before we write the code; since we already know the API we want, we can write some example use cases down and use those as our tests. Here’s a test suite that mirrors the requirements listed above in executable form. It’s not exhaustive, and takes a few short-cuts — for example an item is defined as ‘existing’ if it already has an `id` — but it will do for our purposes.

Figure 4.27. `browser/todo_list/spec/todo_model_spec.js`

```
JS.Test.describe("TodoModel", function() { with(this) {
  before(function() { with(this) {
    this.model = new TodoModel()
  }})

  describe("save()", function() { with(this) {
    it("assigns a sequential ID to new items", function() { with(this) {
      var items = [{title: "Make breakfast"}, {title: "Wash up"}]
      model.save(items[0])
      assertEqual( 1, items[0].id )
      model.save(items[1])
      assertEqual( 2, items[1].id )
    }})

    it("does not change the ID of existing items", function() { with(this) {
      var item = {id: 42, title: "Discover meaning of life"}
      model.save(item)
      assertEqual( 42, item.id )
    }})

    it("emits a 'create' event for new items", function() { with(this) {
      expect(model, "trigger").given("create", {id: 1, title: "Renew passport"})
      model.save({title: "Renew passport"})
    }})

    it("emits an 'update' event for existing items", function() { with(this) {
      expect(model, "trigger").given("update", {id: 1, title: "Renew passport"})
      model.save({id: 1, title: "Renew passport"})
    }})
  }})

  describe("load()", function() { with(this) {
    before(function() { with(this) {
      model.save({title: "Make breakfast"})
    }})

    it("emits a 'load' event with the data for the ID", function() { with(this) {
      expect(model, "trigger").given("load", {id: 1, title: "Make breakfast"})
      model.load(1)
    }})
  }})

  describe("remove()", function() { with(this) {
    before(function() { with(this) {
      model.save({title: "Make breakfast"})
    }})

    it("emits a 'remove' event with the data for the ID", function() { with(this) {
      expect(model, "trigger").given("remove", {id: 1, title: "Make breakfast"})
      model.remove(1)
    }})

    it("deletes the item from the collection", function() { with(this) {
      model.remove(1)
      expect(model, "trigger").given("load", anyArgs()).exactly(0)
      model.load(1)
    }})
  }})
}})
```

Notice how many tests reuse the same item data. When testing different scenarios like this, it can be useful to have the tests differ as little as possible, so you can be sure of the reason for the differences in behaviour. I often start with a ‘default’ scenario and then test variations of that by making only the smallest changes required to trigger a different behaviour. You may also want to test that a method deals with lots of different inputs correctly, that the inputs you’ve used in the tests are not getting ‘special treatment’, but the above suite is more concerned with the behavioural contract the API provides, rather than exhaustively testing each method. You can always add more tests later to nail down the details.

When we run the tests, they fail, since we’ve not implemented the `TodoModel` class yet. The errors the tests report will tell you the next thing you need to fix, so you can add code to make each test in turn turn green. If you write just enough code to make the tests pass, and no more, then you know your code has good test coverage, since all of it was written to make a failing test pass.

To implement `TodoModel`, I’ve again used `Backbone.Events` to provide the event emitter behaviour, but I’ve implemented all the other methods myself to provide a basic implementation that makes the tests work. The class keeps a list of items in `this._items`, indexed by `item.id`, and keeps a counter in `this._itemId` that stores the ID of the last new item. The methods manage this state and emit events as required by the spec.

Figure 4.28. `browser/todo_list/lib/todo_model.js`

```
var TodoModel = function() {
  this._items  = {}
  this._itemId = 0
}
_.extend(TodoModel.prototype, Backbone.Events)

TodoModel.prototype.save = function(item) {
  var event = item.id ? "update" : "create"
  if (!item.id) {
    item.id = ++this._itemId
  }
  this._items[item.id] = item
  this.trigger(event, item)
}

TodoModel.prototype.load = function(id) {
  var item = this._items[id]
  if (item) this.trigger("load", item)
}

TodoModel.prototype.remove = function(id) {
  var item = this._items[id]
  delete this._items[id]
  if (item) this.trigger("remove", item)
}
```

Implementing this class gets us back to having a working application with all the logic covered by tests. All that’s needed is a little glue to set the application up for real on our production pages:

Figure 4.29. `browser/todo_list/main.js`

```
(function() {
  var model   = new TodoModel(),
      editor  = new TodoEditor(".todo-form", model),
      display = new TodoDisplay(".todo-list", model)
})()
```

We no longer need any global variables — hence the `(function() { … })()` wrapper to stop these variables leaking — and all the components are well-tested. There is a small gap in the test coverage though: we know that `TodoModel` works, and we know `TodoEditor` and `TodoDisplay` work given a fake version of the model API, but we don’t know if that fake API conforms to the actual behaviour of `TodoModel`. To cover this gap, we could write a few ‘sanity check’ integration tests that wire the whole app up and check a few common scenarios. These tests don't need to go into all the edge cases of the application, since those details are covered by unit tests, which are simpler to write and often quicker to run.

The integration tests should just make sure everything's wired up correctly. If they fail, you should try to add some unit tests that reproduce the failure, and fix those, before returning to the integration tests. Integration tests are more complex and harder to maintain, due to the increased numbers of components involved and greater setup required, so always try to fix problems in simpler parts of the system rather than adding to the weight of complex, hard-to-understand code.

4.2.5. Using frameworks

There is a plethora of choices when it comes to JavaScript application frameworks these days. Angular[13], Backbone, Ember[14], Knockout[15], Meteor[16], React[17], Rendr[18], not to mention all the different template languages, modelling libraries, data binding frameworks, socket APIs, the combinations are endless. There isn't space to cover them all here, but you can apply all the principles and approaches we've seen here to most of these tools. Many of them make things we've built here implicit and automatic, for example they provide models that the UI components already know how to listen to, to detect changes and update the DOM correctly. In Angular, this is done using declarative bindings in the HTML, in Backbone you need to listen to a model's `"change"` event and call the view's `render()` method to update the DOM. Frameworks often provide more declarative ways of binding DOM events to UI logic functions, they might give you a template language, know how to sync data to the server, and so on. But all of them loosely follow the MVC design pattern of keeping data and UI separate and reactively updating the UI when the data changes.

The app we've been building actually very closely resembles Backbone's architectural style. In Backbone, a `View` is created with an element to bind to, and a reference to a `Model` or `Collection`; the view can listen to `"change"` events, which models and collections emit whenever they are modified, and invoke its `render()` method to reflect the latest state of the data. You can apply the exact same testing strategy that we've used here to a Backbone app with fairly similar results: create some fixture HTML, create a `View` bound to that HTML, pass in a fake or a real `Model`, use Syn to simulate user interaction and check that the `Model` receives the right updates.

Whichever framework you're using, try to keep UI components decoupled, and have them communicate either via a model or a shared event emitter. Some frameworks have already designed a system like this for you, others require a bit more manual work. Now that you've seen how the underlying architecture works, you can apply these ideas no matter how you're building your app.

The key thing to remember is that, as magical as some of these frameworks seem, it's all just regular JavaScript objects and functions. Similarly, code that runs in tests is not 'special', it runs the same as it would in any other web page. So long as you can inject your HTML or templates into the page using a fixture, you should be able to take the framework's start-up code and drop it into a test to get your UI running. Also remember that framework functions work just like your own JavaScript code, and you can stub or mock any interface you like.

4.3. Talking to the server

The model that we implemented in the `TodoModel` class is a little limited for real-world use, since it only stores to-do items in memory rather than actually saving them anywhere. To make something truly

[13]http://angularjs.org/
[14]http://emberjs.com/
[15]http://knockoutjs.com/
[16]https://www.meteor.com/
[17]http://facebook.github.io/react/
[18]https://www.npmjs.org/package/rendr

useful, the items will need to be permanently saved somewhere, like on a server. When we write unit tests, we only have a static web page as our environment, and our backend server is not running.

It is possible to build a testing environment where your backend *is* running, even building a JavaScript test page into the server itself, but this is usually counter-productive. Tests that integrate with the backend will run far more slowly and be harder to build; you will need to expose URLs for setting up internal infrastructure like database connections, possibly faking out services such as email sending, and resetting the state of the backend by emptying the database, simply so that you can control these things from JavaScript running in the browser. This is both costly and inadvisable on security grounds.

If you understand the API offered by your backend, and it's reasonably stable, you can build a fake version of it to run your JavaScript tests against, resulting in tests that run quickly and require minimal infrastructure to maintain. I've re-implemented the `TodoModel` API as `TodoService`, which supports similar operations but backs onto an HTTP API rather than in-memory storage. It uses several different styles of Ajax call provided by jQuery, and we'll discuss the testing trade-offs of each of them.

Figure 4.30. `browser/ajax/todo_service.js`

```
var TodoService = function() {}
_.extend(TodoService.prototype, Backbone.Events)

TodoService.prototype.load = function(id, callback) {
  $.ajax({
    type: "GET", url: "/todos/" + id, context: this,

    success: function(response) {
      this.trigger("load", response)
      callback(null, response)
    },
    error: function() {
      callback(new Error("Item #" + id + " not found"))
    }
  })
}

TodoService.prototype.save = function(item, callback) {
  var self = this

  if (item.id) {
    $.ajax({type: "PUT", url: "/todos/" + item.id, data: item}).then(function(response) {
      self.trigger("update", response)
      callback(null, response)
    }, function(error) {
      callback(new Error(error.responseJSON.error))
    })
  } else {
    $.post("/todos", item, function(response) {
      item.id = response.id
      self.trigger("create", item)
      callback(null, item)
    })
  }
}

TodoService.prototype.remove = function(id, callback) {
  $.ajax({type: "DELETE", url: "/todos/" + id, context: this}).then(function(response) {
    this.trigger("remove", response)
    callback(null, response)
  }, function() {
    callback(new Error("Item #" + id + " not found"))
  })
}
```

`TodoService` supports similar `load()`, `save()` and `remove()` methods to `TodoModel`, but since they now talk to a server asynchronously the methods each take a callback. These callbacks are Node-style 'error-first' callbacks, that is they look like `function(error, response) { … }`; if there is an error the callback is called with the error as its first argument, otherwise the first argument is `null` and the result is passed in the second argument. This pattern is adopted by most Node modules and many other JavaScript libraries, and following it makes it easy to interoperate with them via libraries like Async[19].

This class shows off some of the many ways jQuery allows you to make Ajax requests: the `GET` request is done using `jQuery.ajax()`, passing `success()` and `error()` functions in the options; the `PUT` and `DELETE` requests are done with `jQuery.ajax()` but using a promise[20] to register the callbacks; and the `POST` is done using the `jQuery.post()` short-hand function. All offer different trade-offs for faking out the server response, as we'll see.

Whichever way you make Ajax requests, remember that your code doesn't 'know' it's talking to the server. There's nothing special, from a language point of view, about a call to `jQuery.ajax()` compared to any other business logic or DOM function. As long as your stubs return something that *looks* like a real response that jQuery would emit, your code will work just the same.

4.3.1. `jQuery.ajax()` with callbacks

Let's look at the `load()` method first. It uses `$.ajax()` to make a `GET` request to `/todos/{id}`, passing in the `success` and `error` options to register callbacks for the response. This is the trickiest kind of Ajax call to stub; the `jstest` stubbing API allows for `stub($, "ajax").yields(…)`, but `yields()` only recognises callbacks passed as positional arguments. If you think about what's going on at a language level, `$.ajax()` is actually taking a single object as an argument and invoking the `success()` method on it, *or*, since we're using the `context` option, invoking the object's `success()` function as an application with `context` as the value of `this`[21].

Since the `jstest` stubbing API does not provide a nice pattern for this case, we will have to hand-write the stub implementation. We can still use the stubbing API, we just have to give it a function rather than telling it how to respond using `given()`, `yields()`, and so on. If the request is successful, jQuery calls the `success()` function with `this` bound to the `context` option and the response as the first argument. We can stub that like so:

Figure 4.31. Stubbing `jQuery.ajax()` with a hand-written implementation

```
stub($, "ajax", function(options) {
  options.success.call(options.context, response)
})
```

Stubbing in the case where the request fails would be exactly the same, but with `success` replaced by `error`. Notice that this fake implementation makes `jQuery.ajax()` invoke its callback *synchronously*, i.e. the callback is invoked before `jQuery.ajax()` returns. In many cases this is fine, but if you want it to be async then wrapping the `options.success()` call in a `setTimeout()` works as well. It's usually better to try to write your code so that it doesn't implicitly depend on the order that things happen in, without using callbacks or promises to enforce a certain order of execution.

We now know *how* to stub the `jQuery.ajax()` method, but we also need to know *what* to stub it out with: what should the value of `response` be? We should make the stub return something that looks like the real server response, or at least close enough that our app can't tell the difference. In these situations we can make requests to our server and capture example responses to use in our tests.

[19]https://npmjs.org/package/async
[20]http://promisesaplus.com/
[21]See Appendix A, *JavaScript functions*.

We can make requests using the command-line program `curl`, which is available on most Unix systems. Before we begin, let's create a shell alias for using `curl` with the header `X-Requested-With: XMLHttpRequest`; this header is set by the jQuery Ajax functions and servers may use it to distinguish Ajax calls from 'normal' requests, so they can serve an HTML or JSON fragment rather than a complete HTML page. We should send this header to make sure we're getting the same response jQuery will see.

Figure 4.32. Simulating Ajax requests with `curl`

```
$ alias ajax="curl -H 'X-Requested-With: XMLHttpRequest'"
```

Now, we can send requests to our server and see how it responds. Let's say we have a server that returns a to-do item in response to a `GET` request, like this:

Figure 4.33. A successful `GET` request

```
$ ajax -iX GET http://localhost/todos/1
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 31

{"id":1,"title":"Mow the lawn"}
```

If the server can't find the given item ID, it returns a `404 Not Found` response, like this. Instead of returning an object representing an item, it returns an error message.

Figure 4.34. A failed `GET` request

```
$ ajax -iX GET http://localhost/todos/2
HTTP/1.1 404 Not Found
Content-Type: application/json; charset=utf-8
Content-Length: 26

{"error":"Item not found"}
```

Using `curl` to talk to the server is useful for seeing what actually comes over the wire, but we also need to understand how that is exposed by the jQuery API. If we open the developer console in the browser while visiting a page on our site, we can interrogate jQuery's responses. Begin by creating the following `log()` function, which takes a `msg` parameter and returns another function that will `console.log()` whatever it's given, along with `msg`. `msg` will help us to distinguish successful responses from errors.

Figure 4.35. `log()` function for inspecting Ajax responses

```
> log = function(msg) { return function(resp) { console.log(msg, JSON.stringify(resp)) } }
```

Next, we try out the Ajax call that we want to stub out, in this case `$.ajax({type: "GET", url: "/todos/{id}"})`. If we try the URL for an item that exists, we see that the `success()` callback is invoked with the parsed JSON from the response as the first argument.

Figure 4.36. Inspecting a successful `$.ajax()` request

```
> $.ajax({type: "GET", url: "/todos/1", success: log("ITEM"), error: log("ERROR")})
ITEM {"id":1,"title":"Mow the lawn"}
```

But, if we try the URL for an item we know does not exist, we see that instead the `error()` callback is invoked, with the first argument being the jQuery XHR object[22]. This object exposes the 404 HTTP status code in the `status` property, and the parsed JSON response body in the `responseJSON` property.

[22]See http://api.jquery.com/jQuery.ajax/#jqXHR.

Figure 4.37. Inspecting a failed `$.ajax()` request

```
> $.ajax({type: "GET", url: "/todos/2", success: log("ITEM"), error: log("ERROR")})
ERROR {"readyState":4,"responseText":"{\"error\":\"Item not found\"}", ...
      "responseJSON":{"error":"Item not found"},"status":404,"statusText":"Not Found"}
```

Now we know how jQuery behaves when given a successful request and a failed one, we can use stubbing to mimic that behaviour in our tests. We're using stubbing to state some assumptions about how the server responds, and then checking how our application behaves given these server responses.

Our testing strategy needs to check a few things about the application's interaction with the server. First, it needs to check that when we call, say, `service.load(1)`, that this is translated into an HTTP call to `GET /todos/1`. That is, we need to check that `TodoService` maps the JavaScript API onto the HTTP API correctly. We can do this using mocking and pattern-matching: we can check the arguments to `jQuery.ajax()` to make sure the right request was made. But we don't need to check *all* the inputs: `jQuery.ajax()` takes some options that specify the request — `type`, `url`, `data` and so on — and some options that specify how to deal with the response — `success`, `error` and `context`. We only need to check the request data, and we can use the `jstest` pattern matchers[23] to do this. This will check for a `GET /todos/101` request:

Figure 4.38. Mocking `jQuery.ajax()` with a pattern matcher

```
expect($, "ajax").given(objectIncluding({type: "GET", url: "/todos/101"}))
service.load(101)
```

Once we've checked that the client sends the right request to the server, we should check what the client does with the server's response. As shown above, there are two situations to check: when happens in the success case and what happens in the failure case. We can stub `jQuery.ajax()` as shown in Figure 4.31, "Stubbing `jQuery.ajax()` with a hand-written implementation" to invoke either the `success()` or `error()` callback as needed to simulate each scenario, and then make assertions about the consequences.

In the success case, we want `TodoService.load()` to yield the JSON response back to its callback with no error, that is the arguments to the callback should be `null` and `{"id": 1, …}`. Having stubbed `jQuery.ajax()`, we can simply call `TodoService.load()` and make some assertions about the arguments it yields to the callback. We also want to check that `TodoService` emits a `"load"` event if the request succeeds, and as we've already seen we can achieve this using `expect(service, "trigger").given("load", …)` to mock the event API.

In the failure case, we can use very similar techniques to test the opposite of the above behaviour. We want the callback to be invoked with an error and no item data, which again we can check by calling `TodoService.load()` and placing assertions inside the callback. In this case, no `"load"` event should be emitted and we can prove this using the `exactly(0)` modifier to `expect()`.

The full `TodoService.load()` test suite is shown below. Notice how the `before()` blocks are used to express in code what each `describe()` block says in words, and pay attention to how the `resume()` function is used in each test; there are some subtleties to it that will be explained afterward.

[23]See Section 2.6.2, "Pattern-matching arguments".

Figure 4.39. `browser/ajax/load_spec.js`

```
JS.Test.describe("TodoService load", function() { with(this) {
  before(function() { with(this) {
    this.service = new TodoService()
  }})

  it("asks the server for an item by ID", function() { with(this) {
    expect($, "ajax").given(objectIncluding({type: "GET", url: "/todos/101"}))
    service.load(101)
  }})

  describe("when the server returns an item", function() { with(this) {
    before(function() { with(this) {
      stub($, "ajax", function(options) {
        options.success.call(options.context, {id: 101, title: "Mow the lawn"})
      })
    }})

    it("yields the item to the caller", function(resume) { with(this) {
      service.load(101, function(error, item) {
        resume(function() {
          assertNull( error )
          assertEqual( {id: 101, title: "Mow the lawn"}, item )
        })
      })
    }})

    it("emits a 'load' event with the item", function(resume) { with(this) {
      expect(service, "trigger").given("load", {id: 101, title: "Mow the lawn"})
      service.load(101, resume)
    }})
  }})

  describe("when the request fails", function() { with(this) {
    before(function() { with(this) {
      stub($, "ajax", function(options) {
        options.error.call(options.context, {status: 404})
      })
    }})

    it("yields an error to the caller", function(resume) { with(this) {
      service.load(101, function(error, item) {
        resume(function() {
          assertEqual( "Item #101 not found", error.message )
          assertSame( undefined, item )
        })
      })
    }})

    it("does not emit a 'load' event", function(resume) { with(this) {
      expect(service, "trigger").given("load", anyArgs()).exactly(0)
      service.load(101, function() { resume() })
    }})
  }})
}})
```

When we check the callback response from `TodoService.load()`, we use an async test, wrapping the callback's contents in `resume(function() { … })`. Since we stubbed `jQuery.ajax()` to work synchronously, this seems like unnecessary boilerplate; surely the test would work if written like this:

Figure 4.40. Potentially misleading asynchronous tests

```
it("yields the item to the caller", function() { with(this) {
  service.load(101, function(error, item) {
    assertNull( error )
    assertEqual( {id: 101, title: "Mow the lawn"}, item )
  })
}})
```

This test *would* indeed pass, but if the code were broken the test would not necessarily fail. Consider what happens if `load()` does not invoke its callback at all: in this case, the assertions are not run, no error is thrown, and the suite passes. You might notice that the assertion count is too low (and some frameworks make you state how many assertions you're expecting for just this eventuality), but this is a brittle way to detect errors. By making the test async using the `resume` parameter, `jstest` will throw a timeout error if `resume()` is never invoked, alerting us to the fact the callback is not called.

Constructing the test this way also means it will continue to work if we decide to make the stubbed `jQuery.ajax()` invoke its callbacks asynchronously. Again, if we leave out the `resume()` boilerplate, the callback won't be run before the test completes and the framework won't necessarily notice. Plus, it acts as nice documentation, telling the reader that they should expect this method to work asynchronously.

In the event tests, where we use `expect(service, "trigger")` to check whether an event was emitted, we use two different constructions to invoke `load()`: `service.load(101, resume)` and the more verbose `service.load(101, function() { resume() })`. The first construction simply says, call `service.load()` and wait for it to complete — by passing `resume` as the callback — before checking the mock expectation. However, the `jstest` `resume()` function adopts the 'error-first' callback convention popularised by Node[24]: if you call it with a string or object as the first argument, your test will show an error. This means it can be used tersely with libraries that follow this convention, but sometimes it's not what you want. In the final test case above, `load()` will yield an error but we're not concerned with that fact, we just want to wait for the call to finish. So, we use the more verbose `function() { resume() }` to make sure `resume()` is called with no arguments. If you'd rather be consistent, then the more verbose form will deal with both cases since it makes sure no arguments are implicitly passed to `resume()`.

4.3.2. `jQuery.ajax()` with promises

For saving existing items, `TodoService` uses the promise-based version of `jQuery.ajax()` to make a `PUT` request. Rather than including the `success()` and `error()` callbacks with the inputs to `jQuery.ajax()`, it attaches them to the return value using `then(success, error)`. This is possible because `jQuery.ajax()` returns an object called a 'promise' that represents the pending response and allows callbacks to be attached that will be invoked when the response actually arrives[25]. The separation of the request parameters in the inputs to `jQuery.ajax()` and the callback registration on its return value makes this type of call easier to stub, as we'll see.

As before, we start by investigating the server's response in different scenarios. First there's the 'happy path' where we update an existing item with valid data:

Figure 4.41. Using `PUT` to update an item

```
$ ajax -iX PUT http://localhost/todos/1 -d "title=Mow%20the%20lawn"
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 31

{"id":1,"title":"Mow the lawn"}
```

[24]See Section 2.7.1, "The error-first callback convention".
[25]In jQuery, as well as `then()`, you will see code that calls `success()`/`done()` and `error()`/`fail()` on the promise object to register either success or failure callbacks. jQuery's promises predate the adoption of the Promises/A+ standard that defines an interoperable promise API, consisting only of the `then()` method that takes *both* the `success()` and `error()` callbacks.

We also have to consider what happens if the data is invalid; in this case suppose the server rejects blank `title` values with a `409 Conflict` response.

Figure 4.42. Invalid item data leading to a `409 Conflict` response

```
$ ajax -iX PUT http://localhost/todos/1 -d "title="
HTTP/1.1 409 Conflict
Content-Type: application/json; charset=utf-8
Content-Length: 35

{"error":"Title must not be blank"}
```

And finally, the item might not exist, in which case the server returns a `404`.

Figure 4.43. Non-existent item ID leading to a `404 Not Found` response

```
$ ajax -iX PUT http://localhost/todos/2 -d "title=Mow%20the%20lawn"
HTTP/1.1 404 Not Found
Content-Type: application/json; charset=utf-8
Content-Length: 26

{"error":"Item not found"}
```

Returning to the browser, we can again use the `log()` function we created before to inspect the output of jQuery promises for each of these requests. I’ve stored the promise in the variable `response` to emphasise that it’s a first-class value that you can pass around; you can add callbacks with `then()` *after* the response has actually arrived and the callbacks will still execute.

Figure 4.44. Inspecting `PUT` responses with jQuery promises

```
> response = $.ajax({type: "PUT", url: "/todos/1", data: {title: "Mow the lawn"}})
> response.then(log("ITEM"), log("ERROR"))
ITEM {"id":1,"title":"Mow the lawn"}

> response = $.ajax({type: "PUT", url: "/todos/1", data: {title: ""}})
> response.then(log("ITEM"), log("ERROR"))
ERROR {"readyState":4,"responseText":"{\"error\":\"Title must not be blank\"}", ...
       "responseJSON":{"error":"Title must not be blank"},"status":409, ...
       "statusText":"Conflict"}

> response = $.ajax({type: "PUT", url: "/todos/2", data: {title: "Mow the lawn"}})
> response.then(log("ITEM"), log("ERROR"))
ERROR {"readyState":4,"responseText":"{\"error\":\"Item not found\"}", ...
       "responseJSON":{"error":"Item not found"},"status":404, ...
       "statusText":"Not Found"}
```

The responses are of the same type as what we saw when using callbacks above: on a successful response, the parsed JSON response body is yielded to the `success()` callback, while for failed responses the jQuery XHR object is yielded to the `error()` callback. Testing this will then proceed much as before; we check that calling `TodoService.save()` with an item with an existing ID makes a corresponding `PUT` request, only we don’t need `objectIncluding()` this time since the arguments to `jQuery.ajax()` are *only* the request data.

To stub out the response, we can construct an instance of `jQuery.Deferred` and resolve or reject it depending on whether we want to simulate a successful or a failed request. We then make `$.ajax()` return this object and the `TodoService` code will carry on as normal[26].

[26]In theory, you could use any promise library you like for this, since `TodoService` only needs the return value to have a `then()` method. But jQuery promises aren’t strictly conformant to the Promises/A+ standard, so to make sure that our canned responses have realistic behaviour it’s best to use jQuery’s own promises.

The test suite is shown below. Notice that when we're mocking with `expect($, "ajax").given(…)` we need to add `returns(xhr)` at the end — the code is expecting a promise to be returned and will throw an error if the mock does not specify a return value. When testing the different response cases, we pass in the minimal data need to trigger 'update' rather than 'create' behaviour — we only need to give the save item an `id` to trigger this. Omitting the `title` property means we can be sure that the data yielded to the caller on success is the data from the server response, not from the client's input.

Figure 4.45. `browser/ajax/save_existing_spec.js`

```
JS.Test.describe("TodoService.save() existing", function() { with(this) {
  before(function() { with(this) {
    this.service = new TodoService()
    this.xhr = new $.Deferred()
    stub($, "ajax").returns(xhr)
  }})

  it("sends the item data to the server", function() { with(this) {
    var params = {type: "PUT", url: "/todos/101", data: {id: 101, title: "Mow the lawn"}}
    expect($, "ajax").given(params).returns(xhr)
    service.save({id: 101, title: "Mow the lawn"})
  }})

  describe("when the request succeeds", function() { with(this) {
    before(function() { with(this) {
      xhr.resolve({id: 101, title: "Updated title"})
    }})

    it("yields the updated item to the caller", function(resume) { with(this) {
      service.save({id: 101}, function(error, item) {
        resume(function() {
          assertNull( error )
          assertEqual( {id: 101, title: "Updated title"}, item )
        })
      })
    }})

    it("emits an 'update' event with the response data", function(resume) { with(this) {
      expect(service, "trigger").given("update", {id: 101, title: "Updated title"})
      service.save({id: 101}, resume)
    }})
  }})

  describe("when the request fails", function() { with(this) {
    before(function() { with(this) {
      xhr.reject({status: 412, responseJSON: {error: "Title must not be blank"}})
    }})

    it("yield an error to the caller", function(resume) { with(this) {
      service.save({id: 101}, function(error, item) {
        resume(function() {
          assertEqual( "Title must not be blank", error.message )
          assertSame( undefined, item )
        })
      })
    }})

    it("does not emit an 'update' event", function(resume) {with(this) {
      expect(service, "trigger").given("update", anyArgs()).exactly(0)
      service.save({id: 101}, function() { resume() })
    }})
  }})
}})
```

4.3.3. `jQuery.post()` shorthand

Finally we come to the *create* behaviour, which uses the convenience function `jQuery.post()`. Calling `jQuery.post(url, data, callback)` is a shorthand for `jQuery.ajax({type: "POST", url: url, data: data, success: callback})`. As such, it provides no way to detect errors (unless you use the promise object that it returns, as we did above), and so we have fewer scenarios to test. We should again use mocking to check that calling `TodoService.save()` without an `id` should produce a `POST /todos` request, and then we can stub `$.post()` to yield a response and check that the correct things happen. The test flows much like the ones we've already seen. The difference in this case is that, since `$.post()` takes the callback as its final positional argument, we can use the `yields()` modifier to specify what it should return to the caller.

Figure 4.46. `browser/ajax/save_new_spec.js`

```
JS.Test.describe("TodoService.save() new", function() { with(this) {
  before(function() { with(this) {
    this.service = new TodoService()
  }})

  it("sends the item to the server", function() { with(this) {
    expect($, "post").given("/todos", {title: "Take out the garbage"}, instanceOf(Function))
    service.save({title: "Take out the garbage"})
  }})

  describe("when the request succeeds", function() { with(this) {
    before(function() { with(this) {
      stub($, "post").yields([{id: 29}])
    }})

    it("yields the new ID to the caller", function(resume) { with(this) {
      service.save({}, function(error, response) {
        resume(function() {
          assertNull( error )
          assertEqual( {id: 29}, response )
        })
      })
    }})

    it("emits a 'create' event with the complete item", function(resume) { with(this) {
      expect(service, "trigger").given("create", {id: 29, title: "Publish a book"})
      service.save({title: "Publish a book"}, resume)
    }})
  }})
}})
```

4.3.4. Using a fake server

All the tests we've seen so far work by stubbing the jQuery Ajax API. This tends to be fairly straightforward since we can see the Ajax calls our code is making directly, and so we can easily figure out which things we need to stub, and what they should return using the inspection techniques above. But an alternative approach is to stub out the browser API on which all these calls rely: the `XMLHttpRequest` object[27]. When making a same-origin request, all the different jQuery calls ultimately call `XMLHttpRequest` at some point, and it's possible to stub this out to completely stub out the browser's HTTP request API.

One library that does this is the Sinon[28] stubbing framework; it lets you fake out `XMLHttpRequest`, record requests, and then selectively reply to them with canned data. The API it exposes is closer to constructing

[27]https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest
[28]http://sinonjs.org/

raw HTTP responses with status codes, headers and bodies than the approach of stubbing the higher-level jQuery APIs. As an example, let's use it to test the `TodoService.remove()` method, which takes an `id` and should send a `DELETE /todos/{id}` request to the server. If the request is successful, then a `"remove"` event should be emitted with the response data.

We can test this by using `sinon.useFakeXMLHttpRequest()` to capture all the requests made by our code. Once a request has been made, we can check that is has the right `method` and `url`, as in the first test below; this accomplishes the same thing as our mock-based tests above. Once we've captured the request, we can send a response back, which comprises a status code, headers and body — in this case we send back a `200 OK` with `Content-Type: application/json` to make jQuery parse the body for us. This lets us check how our app behaves under a given response from the server.

Figure 4.47. `browser/ajax/remove_spec.js`

```
JS.Test.describe("TodoService.remove()", function() { with(this) {
  before(function() { with(this) {
    this.service  = new TodoService()
    this.xhr      = sinon.useFakeXMLHttpRequest()
    this.requests = []

    xhr.onCreate = function(request) {
      requests.push(request)
    }
  }})

  after(function() { with(this) {
    xhr.restore()
  }})

  it("tells the server to delete the item", function() { with(this) {
    service.remove(101)
    assertEqual( "DELETE", requests[0].method )
    assertEqual( "/todos/101", requests[0].url )
  }})

  it("emits a 'remove' event with the response", function(resume) { with(this) {
    expect(service, "trigger").given("remove", {id: 101})
    service.remove(101, resume)
    requests[0].respond(200, {"Content-Type": "application/json"}, '{"id":101}')
  }})
}})
```

The flow of capturing a request first and then telling Sinon how to respond to it can make for oddly ordered tests sometimes, but you can also define canned responses in advance using `sinon.fakeServer`[29]. For example, we could replace the canned response in the above example with:

Figure 4.48. Dispatching Ajax requests with Sinon's fake server

```
var server = sinon.fakeServer.create()
server.respondWith("DELETE", "/todos/101",
                   [200, {"Content-Type": "application/json"}, '{"id":101}'])
```

While working with raw HTTP data can be more appealing than stubbing the jQuery APIs in some situations, it does have some drawbacks. Firstly, it only deals with Ajax requests — those made using `XMLHttpRequest` — not JSON-P. JSON-P works by injecting `<script>` elements into the DOM and those APIs can't be stubbed in the same way as `XMLHttpRequest`, at least not without affecting all other code that needs to handle the DOM. So if you're making JSON-P requests, it's best to stub the high-level functions that wrap this functionality.

[29]See http://sinonjs.org/docs/#fakeServer.

Second, since you're working at a lower level, you'll be dealing with serialised data rather than parameter objects. For example, if you make a `GET` request with some parameters, they will end up in the `url` as a query string.

Figure 4.49. Checking serialised `GET` parameters

```
$.get("/search", {q: "Mogwai tour dates", order: "date"})
assertEqual( "/search?q=Mogwai%20tour%20dates&order=date", requests[0].url )
```

This is problematic because you're forced to check a serialised string rather than a structured object. The objects `{a: 1, b: 2}` and `{b: 2, a: 1}` are considered equal by object matchers, but the strings `"a=1&b=2"` and `"b=2&a=1"` are not equal, so it's easier to check that parameters are correct using objects than it is with strings. String-based tests will break if the order of the parameters changes in the code, which for query strings the order often doesn't matter. If you find yourself checking query strings, JSON request bodies, and so on, you should usually parse the string into an object first; this both makes sure the serialization is valid and makes the structure easier to work with.

You also can't use the default pattern-matchers, for example this would not work if the parameters were given as a query string:

Figure 4.50. Pattern-matching `GET` parameters

```
expect($, "get").given("/search", objectIncluding({order: "date"}))
$.get("/search", {q: "Mogwai tour dates", order: "date"})
```

This can be solved by inventing your own matcher[30]. For example, a matcher that matched a query string including `q=Mogwai%20tour%20dates` would look like this:

Figure 4.51. A custom matcher for query strings

```
var queryIncluding = function(params) {
  var matcher = objectIncluding(params)
  return {
    equals: function(string) {
      if (typeof string !== "string") return false

      var parts = string.split("&"), query = {}, pair

      for (var i = 0, n = parts.length; i < n; i++) {
        pair = parts[i].split("=")
        query[decodeURIComponent(pair[0])] = decodeURIComponent(pair[1])
      }
      return matcher.equals(query)
    }
  }
}
```

If we use this in a test, we see that it works as expected, and we could use this when mocking to declaratively match arguments given in serialised form.

Figure 4.52. Using a custom matcher

```
var q = queryIncluding({q: "Mogwai tour dates"})

q.equals("q=Mogwai%20tour%20dates&order=date")
// -> true

q.equals("q=&order=date")
// -> false
```

[30]See Section 2.6.2, "Pattern-matching arguments".

4.3.5. WebSocket and other protocols

Ajax isn't the only way to talk to the server, we also have tools like `WebSocket` and `EventSource` in modern browsers. Again, the trick is to remember that your code doesn't 'know' it's talking over a network, it's just calling functions and expecting to get something back. Those functions can usually be stubbed; we don't need a special `FakeWebSocketServer`, we can just use regular old stubbing and mocking.

Suppose we have a little class whose only job is to open a socket to the server, and relay any messages that arrive by emitting them as events with the parsed JSON from the message. The code looks like this:

Figure 4.53. `browser/websockets/notifier.js`

```
var Notifier = function() {}
_.extend(Notifier.prototype, Backbone.Events)

Notifier.prototype.listen = function(path) {
  var socket = new WebSocket("ws://localhost" + path),
      self   = this

  socket.onmessage = function(event) {
    self.trigger("update", JSON.parse(event.data))
  }
}
```

Just as we did with `TodoService`, we want to check two things: the class opens a connection to the right URL when instructed, and that the class interprets data from the server correctly. The first case is a mocking problem: when I call this method, what other functions does it call? We can use mocking for this: `expect("new", "WebSocket").given(url)` states that the class must make a call to `new WebSocket(url)` at some point.

To check how data from the server is handled, we need to observe how the code works at a language level, ignoring what the code really means. When we call `new WebSocket()`, we expect to get some object back and we assign a method called `onmessage()` to that object. When `onmessage()` is called, we expect it to receive some object with a `data` property, which is a string of JSON. That's all the information we need to stub it correctly: note what types of values the code deals with, then find a way to hand it those values.

For WebSocket, we can make `new WebSocket()` return a fake object that we control from our tests. After `Notifier` has had a chance to open the socket — at which point our fake gets into the system — we just call `socket.onmessage({data: "..."})` and the code will behave like it got a real WebSocket message.

The tests are shown below. The second test might look a little complicated, but all it's doing is setting up a fake socket and calling `notifier.listen()` so that the `Notifier` gets hold of the fake socket. Then it sets up a mock expectation about the event that should be emitted, before invoking `onmessage()` to pass fake data in.

Figure 4.54. `browser/websockets/notifier_spec.js`

```
JS.Test.describe("Notifier", function() { with(this) {
  before(function() { with(this) {
    this.notifier = new Notifier()
  }})

  it("opens a socket to the given URL", function() { with(this) {
    expect("new", "WebSocket").given("ws://localhost/updates").returns({})
    notifier.listen("/updates")
  }})

  it("emits an event with data received from the socket", function() { with(this) {
    var socket = {}
    stub("new", "WebSocket").returns(socket)
    notifier.listen("/updates")

    expect(notifier, "trigger").given("update", {hello: "world"})
    socket.onmessage({data: '{"hello":"world"}'})
  }})
}})
```

This is a minimal example, and in real life there's a lot more complexity to dealing with socket connections — making sure we reconnect when we lose the connection, retrying dropped messages, transport negotiation — and many apps use an abstraction like Socket.IO[31] rather than deal with sockets directly. But whatever you're using, follow the strategy above and figure out how you can mimic the library's API sufficiently well that your code can't tell the difference.

When designing apps around sockets, don't couple too much of your app to them. Don't put large amounts of business logic inside the message-handling functions; put business logic in your *own* objects with APIs you can feed messages into to test them, then write a little bit of glue to hook those objects up to listen to a socket connection. You want as little of your codebase as possible being tightly coupled to networking concerns.

4.4. Storing data

Talking to a server is not the only way to store the user's data. These days, we have client-side storage APIs like `sessionStorage` and `localStorage`[32], WebSQL and IndexedDB[33] to play with. Some frameworks use `localStorage` as the API the client-side app sits on while they sync the contents of `localStorage` with your server behind the scenes, producing two broad classes of testing scenarios: testing code that's built on top of `localStorage`, and testing code that takes things *from* `localStorage` and moves or copies them somewhere else.

`localStorage` is a singleton, meaning there's only one `localStorage` object available in each page; it's not a class you can create multiple instances of. This is both an advantage, since the tests can access the same `localStorage` as the code and inspect what's going on, and a disadvantage, since it becomes harder to isolate the state of multiple components that all want to store things in there. It requires a little more thought to make sure different components don't clobber one another's data.

As an example, let's rewrite our `TodoModel` class[34] again to keep its state in `localStorage` rather than in memory. The changes are fairly simple: replacing an index kept in a JavaScript object with one kept in `localStorage`. The one subtlety is that `localStorage` only stores string values, and will implicitly call `toString()` on anything you put in there. For this reason, we use `JSON` to encode the objects we want to store.

[31]http://socket.io/
[32]https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Storage
[33]https://developer.mozilla.org/en/docs/IndexedDB
[34]See Figure 4.28, "`browser/todo_list/lib/todo_model.js`".

Also, `localStorage` supports two APIs for manipulating its data. You can use it just like any other JavaScript object, using `localStorage.key = "value"` to store an item, `localStorage.key` to read it and `delete localStorage.key` to remove it. But, it also has methods for each of these: `localStorage.setItem("key", "value")`, `localStorage.getItem("key")` and `localStorage.removeItem("key")`. I've used a mixture of both below, since each presents different testing constraints.

Figure 4.55. `browser/local_storage/todo_store.js`

```
var TodoStore = function() {
  var itemId = localStorage.itemId
  if (!itemId) localStorage.itemId = "0"
}
_.extend(TodoStore.prototype, Backbone.Events)

TodoStore.prototype.save = function(item) {
  var itemId = parseInt(localStorage.itemId, 10)
  if (item.id) {
    this.trigger("update", item)
  } else {
    item.id = ++itemId
    localStorage.itemId = itemId.toString()
    this.trigger("create", item)
  }
  localStorage["items:" + item.id] = JSON.stringify(item)
}

TodoStore.prototype.load = function(id) {
  var item = localStorage["items:" + id]
  if (item) this.trigger("load", JSON.parse(item))
}

TodoStore.prototype.remove = function(id) {
  var item = localStorage.getItem("items:" + id)
  if (item) this.trigger("remove", JSON.parse(item))
  localStorage.removeItem("items:" + id)
}
```

4.4.1. Black-box testing

To test this, we could adopt the same approach that we used for `TodoModel`, where we treat it like a black box, don't inspect its internal state and just look at we get out of its public API. This works fine, only because we're now dealing with a singleton rather than creating completely new objects in each test case, there's going to be global state lingering around. We don't want items stored in one test to affect the outcome of subsequent tests; we want every tests to run from a clean slate, and this means we need to clean up at the end of each test, setting the world back to its original state. In the case of `localStorage`, this simply means calling `clear()` in an `after()` block to remove any state we created.

We do this at the end of each test, rather than at the start, to make sure the slate is clean after we've finished this suite. If the clean-up was done in a `before()` block, then at the end of this test suite we could have data sitting around in `localStorage` that was created by the last test. This means that whatever we run next, whether it be the next test suite in the run, or some manual debugging, could be affected by data we left lying around. So, we always put clean-up code at the end of the test.

The spec for `TodoStore` is then a carbon copy of that for `TodoModel`, with this `after()` block added:

Figure 4.56. *`browser/local_storage/todo_store_spec.js`*

```
JS.Test.describe("TodoStore", function() { with(this) {
  before(function() { with(this) {
    this.store = new TodoStore()
  }})

  after(function() { with(this) {
    localStorage.clear()
  }})

  describe("save()", function() { with(this) {
    it("assigns a sequential ID to new items", function() { with(this) {
      var items = [{title: "Make breakfast"}, {title: "Wash up"}]
      store.save(items[0])
      assertEqual( 1, items[0].id )
      store.save(items[1])
      assertEqual( 2, items[1].id )
    }})

    it("does not change the ID of existing items", function() { with(this) {
      var item = {id: 42, title: "Discover meaning of life"}
      store.save(item)
      assertEqual( 42, item.id )
    }})

    it("emits a 'create' event for new items", function() { with(this) {
      expect(store, "trigger").given("create", {id: 1, title: "Renew passport"})
      store.save({title: "Renew passport"})
    }})

    it("emits an 'update' event for existing items", function() { with(this) {
      expect(store, "trigger").given("update", {id: 1, title: "Renew passport"})
      store.save({id: 1, title: "Renew passport"})
    }})
  }})

  describe("load()", function() { with(this) {
    before(function() { with(this) {
      store.save({title: "Make breakfast"})
    }})

    it("emits a 'load' event with the data for the ID", function() { with(this) {
      expect(store, "trigger").given("load", {id: 1, title: "Make breakfast"})
      store.load(1)
    }})
  }})

  describe("remove()", function() { with(this) {
    before(function() { with(this) {
      store.save({title: "Make breakfast"})
    }})

    it("emits a 'remove' event with the data for the ID", function() { with(this) {
      expect(store, "trigger").given("remove", {id: 1, title: "Make breakfast"})
      store.remove(1)
    }})

    it("deletes the item from the collection", function() { with(this) {
      store.remove(1)
      expect(store, "trigger").given("load", anyArgs()).exactly(0)
      store.load(1)
    }})
  }})
}})
```

This is a fine approach, if all we want to test is the public API and not concern ourselves with how the data is persisted internally. But sometimes we *do* care about persistence details, and we need to take a different testing approach here.

4.4.2. Testing persistence contracts

You can think of `TodoStore` as having a *contract* with `localStorage` that says how `TodoStore` will save data, and how it expects the data to look when it comes back to read it. While an integration test can implicitly check that data is round-tripped from our program to the data store and back, it's sometimes useful to *explicitly* check what's going on behind the scenes. This is particularly true if multiple components want to integrate with the same data storage schema.

For example, we could write some tests that state that when a new item is saved, it should be stored as JSON with a new ID under a known key and the `itemId` counter should be assigned to the newly issued ID. When saving an existing item the data under its key should be updated and the `itemId` counter should be left unchanged.

Figure 4.57. *`browser/local_storage/todo_store_save_spec.js`*

```
JS.Test.describe("TodoStore.save()", function() { with(this) {
  before(function() { with(this) {
    this.store = new TodoStore()
  }})

  after(function() { with(this) {
    localStorage.clear()
  }})

  describe("saving a new item", function() { with(this) {
    before(function() { with(this) {
      store.save({title: "Make breakfast"})
    }})

    it("saves the item as JSON", function() { with(this) {
      var item = JSON.parse(localStorage["items:1"])
      assertEqual( {id: 1, title: "Make breakfast"}, item )
    }})

    it("records the item ID", function() { with(this) {
      assertEqual( "1", localStorage.itemId )
    }})
  }})

  describe("saving an existing item", function() { with(this) {
    before(function() { with(this) {
      localStorage.itemId = "3"
      localStorage["items:2"] = JSON.stringify({id: 2, title: "Wash up"})
      store.save({id: 2, title: "Renew passport"})
    }})

    it("updates the stored item", function() { with(this) {
      var item = JSON.parse(localStorage["items:2"])
      assertEqual( {id: 2, title: "Renew passport"}, item )
    }})

    it("does not change the item ID", function() { with(this) {
      assertEqual( "3", localStorage.itemId )
    }})
  }})
}})
```

These tests inspect the database after an API call, and set up state by hand-crafting the state of the database, bypassing the `TodoStore` API. This makes explicit our expectations about the data format and how it's interpreted. Note how they use `JSON.parse()` to check the stored data; this makes sure they are serialised correctly while leaving the tests immune to the particular order the item's properties are serialised in.

We've tested write operations by calling `TodoStore.save()` and then inspecting what's in the database. Similarly, we can put items into the database by hand and then check how `load()` behaves when trying to read that data. It's similar in spirit to stubbing out `jQuery.get()` with a canned response; we're stating an assumption about how the underlying storage appears, and then checking our code works with the canned data.

Figure 4.58. *`browser/local_storage/todo_store_load_spec.js`*

```
JS.Test.describe("TodoStore.load()", function() { with(this) {
  before(function() { with(this) {
    this.store = new TodoStore()
    localStorage["items:5"] = JSON.stringify({id: "fake-id", title: "Buy cough syrup"})
  }})

  after(function() { with(this) {
    localStorage.clear()
  }})

  it("emits a 'load' event with an existing item", function() { with(this) {
    expect(store, "trigger").given("load", {id: "fake-id", title: "Buy cough syrup"})
    store.load(5)
  }})

  it("does not emit a 'load' event for unknown IDS", function() { with(this) {
    expect(store, "trigger").given("load", anyArgs()).exactly(0)
    store.load(50)
  }})
}})
```

These tests use a different `id` in the `localStorage` key (`5`) from that which actually appears in the data (`"fake-id"`). This accomplishes two things; first it makes sure that the data emitted with the `"load"` event comes from the database value, not from the key or from the `id` the user passed to the `load()` method, and second it proves that `load()` looks the item up using the `id` as a key, rather than by inspecting each record to see if its ID matches. This might seem silly in this example but in practice it can be worth enforcing details like this to make sure the system's workings are understood.

If you were testing a component that took data from `localStorage` and synchronised it with a server, you could take much the same approach: you would write tests that dumped some data into `localStorage`, run your component and check what it did with the data you'd set up. These checks chould use mocking to assert certain calls to `jQuery.post()`, for example.

4.4.3. Mocking database interactions

The above examples inspect what `TodoStore` is doing internally, but they still go through the process of actually storing and retrieving information from the database. This might be undesirable; the tests might become slow, it might be hard to keep the environment's global state clean, and so on. While it's certainly worth having a few tests that really hit the database to check things are wired up properly, if your app interacts with the database via method calls then you can also use mocking to check these interactions. For example, the `TodoStore.remove()` method should call `localStorage.getItem()` to look up the item it's been told to delete. If `getItem()` returns an object, it should emit a `"remove"` event. And finally, it should call `localStorage.removeItem()` to delete the data.

Testing like this is all about checking what our code *tells* the database to do: when we call a `TodoStore` method, what commands does it send to the layer below it? Using mocking also means that less data is really hitting the database, meaning your tests won't slow down as the volume of data in there increases.

At the top of the following test suite, we stub out `localStorage` completely with a fake object, and stub the methods used by the class we're testing. This makes sure no data ends up in the actual `localStorage`, removing the need for a clean-up step at the end. Note that some browsers throw an error if you stub out methods on the *real* `localStorage`, and some complain if you replace the `localStorage` object entirely as we do here (for example, Firefox throws `TypeError: setting a property that has only a getter`, which means you're allowed to read the `localStorage` variable but not reassign it to another value). If you need to target these browsers, consider having the `TodoStore` constructor take a reference to `localStorage` as a parameter; in your tests you could then inject a completely fake object while using the real `localStorage` at runtime.

Figure 4.59. `browser/local_storage/todo_store_remove_spec.js`

```
JS.Test.describe("TodoStore.remove()", function() { with(this) {
  before(function() { with(this) {
    stub("localStorage", {})
    stub(localStorage, "getItem")
    stub(localStorage, "removeItem")

    this.store = new TodoStore()
  }})

  it("tells localStorage to look up an item", function() { with(this) {
    expect(localStorage, "getItem").given("items:99")
    store.remove(99)
  }})

  it("emits a 'remove' event if the item exists", function() { with(this) {
    stub(localStorage, "getItem").returns(JSON.stringify({id: "fake-id"}))
    expect(store, "trigger").given("remove", {id: "fake-id"})
    store.remove(99)
  }})

  it("does not emit a 'remove' event if the item does not exist", function() { with(this) {
    stub(localStorage, "getItem").returns(null)
    expect(store, "trigger").given("remove", anyArgs()).exactly(0)
    store.remove(99)
  }})

  it("tells localStorage to remove an item", function() { with(this) {
    expect(localStorage, "removeItem").given("items:99")
    store.remove(99)
  }})
}})
```

There is one flaw with these tests: it's important that `getItem()` is called *before* `removeItem()`, otherwise we would lose the data we're supposed to emit via the `"remove"` event. There are tools that allow you to check what order things were called in — Sinon's mocking framework supports it, for example — but you can also fix it by making the `removeItem()` call conditional on the result of `getItem()`, so that it's only called if `getItem()` returns something. Then the tests could use basic mocking to check that `removeItem()` is or is not called, depending on the result of `getItem()`, thus proving the tasks are done in the right order.

If order matters, I usually prefer to enforce this using dependencies in the code that are stronger than 'this statement follows that one'. `if` statements provide one way to achieve this, by making things conditional on previous tasks. You can also make sure one function follows another by having the output of one be the input of the next, so that you cannot run the second function without previously getting the result of the first. Callbacks and promises can also solve this problem; putting code in a callback is a way of saying that it depends on the result an operation, and therefore must be run *after* said operation.

4.5. `location`, `pushState` and routers

URLs are central to navigation on the web. We use them to identify pieces of content, to link them together, to bookmark them. JavaScript apps, however, have had a somewhat turbulent relationship with URLs, stemming in part from the behaviour of the browser's `location` object.

The browser environment provides a global variable called `location`[35], which represents the current URL as a structured object. The following diagram illustrates the properties of the `location` object and which part of the URL it identifies.

Figure 4.60. Structure of the `location` object

```
                              href
                               |
 /------------------------------------------------------------\

                host        pathname                     hash
                  |            |                          |
        /------------------\/-----\                    /------\

 http://www.example.com:4567/search?q=concert%20tickets#results

 \---/  \-------------/ \--/       \------------------/
   |           |         |                  |
protocol    hostname    port              search

 \-------------------------/
              |
           origin
```

So for example, if we were visiting the above URL, the value of `location.host` would be `"www.example.com:4567"` and `location.hash` would be `"#results"`. When the browser makes a request, `hash` is not sent to the server; for example the above URL would cause the browser to make a TCP connection to `www.example.com:4567` and then send a request beginning with `GET /search?q=concert%20tickets HTTP/1.1`. `hash` only exists on the client, but JavaScript apps can read any property of `location` and use it to determine which content to display to the user.

We can also use `location` to change the current URL: if you assign a new value to any property of `location` then the browser will update the current URL with that change[36]. If you change `location.hash` then the URL is updated but the page is not reloaded (since `hash` is not sent to the server); the current page and all its JavaScript continues running. For all other parts of the URL, changing it causes the browser to reload with the new URL, shutting down the currently active page.

For a long time, this was all we had to work with; if we wanted to 'bookmark' different states of a UI run using JavaScript, we had no choice but to use `location.hash`. Changing any other property would unload the page. Various 'history management' libraries sprang up that allowed JavaScript apps to use `location.hash` to encode state information; on page load, and whenever `location.hash` was changed, the history manager would read `location.hash` and determine what to render.

Eventually we began using `hash` on the client in the same way that `pathname` is used on the server, migrating the whole resource routing system onto the client. The *hashbang URL* convention, wherein

[35]See https://developer.mozilla.org/en-US/docs/Web/API/Window.location.

[36]If you set `location.href = url` or simply `location = url`, then `url` is resolved relative to the current URL. For example, given the above URL, `location.href = "/foo"` would update the URL to `http://www.example.com:4567/foo`.

URLs like `https://twitter.com/jack` became `https://twitter.com/#!/jack`, was widely adopted as a method to expose JavaScript-rendered views to search engines[37]. These URLs send only the pathname `/` to the server, leaving the hash `#!/jack` to be interpreted, routed and rendered on the client side.

But this hack quickly led to rather a lot of problems for performance and crawlability; because `location.hash` is *only* available on the client, it's not possible to make the server render a meaningful response for a hashbang URL. So, more powerful URL manipulation was added to browsers in the form of the `history.pushState()` API[38]. This lets developers change `location` in JavaScript *without* unloading the page, allowing JavaScript to maintain state using 'traditional' URLs that can be rendered by the server.

Nowadays, many application frameworks provide a 'router' that abstracts over `location.hash` and `history.pushState()` depending on the capabilities of the browser. Different ways of manipulating `location` produce different testing challenges; let's examine a few of them.

4.5.1. Testing `location` changes

Two common scenarios present themselves when testing URL-related concerns: we often want to test that performing some action makes the browser redirect to a new URL, and we want to test what our app's response to a given URL should be. Both scenarios are tricky since any modification to `location` will make the browser unload the page, aborting our test suite. So, we need to be able to test things without touching `location` itself.

For example, suppose we have a function whose job is to search for things and redirect to the first result found. It does this by making an Ajax request with a search query, receiving a URL from the server and setting `location.href` to this URL.

Figure 4.61. `browser/routing/untestable_search.js`

```
var search = function(query) {
  $.get("/search", {q: query}, function(response) {
    location.href = response.url
  })
}
```

This code cannot be tested; if we invoke `search()` and stub out `$.get()` to return a canned response, the `location.href` change will reload the page. If we introduce some indirection, we can use mocking to solve this problem. Say we invent a very simple function that acts as a wrapper for setting `location.href`:

Figure 4.62. `browser/routing/redirect.js`

```
var redirect = function(url) {
  location.href = url
}
```

Then, we replace the `location.href` setting in our code with a call to this function:

Figure 4.63. `browser/routing/search.js`

```
var search = function(query) {
  $.get("/search", {q: query}, function(response) {
    redirect(response.url)
  })
}
```

[37]See https://developers.google.com/webmasters/ajax-crawling/docs/specification.
[38]https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Manipulating_the_browser_history

This allows us to test the code using straightforward mocking: we say that `redirect()` should be called with the URL returned by the server.

Figure 4.64. `browser/routing/search_spec.js`

```
JS.Test.describe("search()", function() { with(this) {
  before(function() { with(this) {
    stub($, "get").given("/search", {q: "mogwai"}).yields([{url: "/artists/70202-mogwai"}])
  }})

  it("redirects to the given URL", function() { with(this) {
    expect("redirect").given("/artists/70202-mogwai")
    search("mogwai")
  }})
}})
```

This little bit of indirection means we can test all the code that should trigger a redirect, without the redirect actually happening. The `redirect()` function is so simple that it can be tested manually, if at all.

What about situations where we need to read the current `location`, rather than change it? Consider this routing function, which renders a different template based on the current value of `location.pathname`:

Figure 4.65. `browser/routing/untestable_router.js`

```
var route = function() {
  var match
  if (location.pathname === "/") {
    render("timeline.html")
  } else if (match = location.pathname.match(/^\/users\/([a-z]+)$/)) {
    render("profile.html", {username: match[1]})
  }
}
```

Just as before, we cannot write a test that sets `location.pathname` to some value before calling `route()`, since that will reload the page. We also can't replace the `location` object itself with a fake, since setting the value of `location` itself also reloads the page. The only solution is therefore to write code that doesn't need to touch the `location` object directly.

Remember that `location` looks just like any other JavaScript object, when we think about it at the language level. The `route()` function doesn't really care that it's talking to the real `location` object, it just wants to talk to *some* object with a `pathname` property. Let's change `route()` so that instead of referring to the global variable `location`, it has a parameter called `location` instead.

Figure 4.66. `browser/routing/router.js`

```
var route = function(location) {
  var match
  if (location.pathname === "/") {
    render("timeline.html")
  } else if (match = location.pathname.match(/^\/users\/([a-z]+)$/)) {
    render("profile.html", {username: match[1]})
  }
}
```

This lets us pass in any object we like when testing the function, without getting the real `location` object involved. We can test all the routing cases by passing in an inert object in and checking which template is rendered. When using `route()` in production, we would pass it the real `location` object to make it respond to the user's current URL.

Figure 4.67. `browser/routing/router_spec.js`

```
JS.Test.describe("route()", function() { with(this) {
  it("routes / to timeline.html", function() { with(this) {
    expect("render").given("timeline.html")
    route({pathname: "/"})
  }})

  it("routes /users/jack to profile.html with username=jack", function() { with(this) {
    expect("render").given("profile.html", {username: "jack"})
    route({pathname: "/users/jack"})
  }})
}})
```

You can use this approach to check location changes as well — if you write a function that takes the `location` object as a parameter rather than referring to the global variable, you can pass in a fake object in your tests and check which properties have been set on it after the function has run. The function won't care that it's not talking to the real `location` object; as far as it's concerned it's just assigning some properties to an object, and it's only the browser that cares about the `location` object being special.

4.5.2. Testing `pushState` and routers

These days, most application frameworks include a router of some kind, and developers use this to manage URLs rather than deal directly with `location.hash` or `history.pushState()`. The framework will often decide which one to use based on the capabilities of the browser, meaning the same app will use `location.hash` for some users and `history.pushState()` for others. For example, Backbone provides a `Router` class that maps URL patterns onto method names; when the router is instantiated it begins monitoring the current URL and will invoke the named methods when the URL matches one of the patterns.

The following Backbone router reproduces the effect of the `route()` function above using pattern-matching instead of `if` statements.

Figure 4.68. `browser/routing/backbone_router.js`

```
var Router = Backbone.Router.extend({
  routes: {
    "home":             "timeline",
    "users/:username": "profile"
  },

  timeline: function() {
    render("timeline.html")
  },

  profile: function(username) {
    render("profile.html", {username: username})
  }
})
```

The code above only *declares* a router class, it does not instantiate it. For the router to become active, it must be instantiated by calling `new Router()` and Backbone's `History` module must be started using `Backbone.history.start()`. These calls are problematic because they both introduce global state: `Backbone.history.start()` can only be called once per page, and once you instantiate a router it will begin reacting to URL changes and cannot be switched off or detached. This means that once you instantiate a router, it remains active throughout all the other tests you run, and therefore we cannot instantiate a new router for each test case — this would result in many copies of the same router running at once, and each URL change would trigger the same dispatch method in all of them, producing a performance problem at best and state conflicts at worst.

So, our routers must be set up once and remain active for the whole test run. For this reason, I've set them up in the `test.html` page rather than in an individual test suite. I've instantiated the router *after* starting the `History` module so that the initial URL of the test page does not trigger any behaviour in the router.

Figure 4.69. Starting a Backbone router

```
<script>
  Backbone.history.start()
  new Router()
</script>
```

With that setup in place, the tests for the Backbone router proceed almost as they did in Figure 4.67, "`browser/routing/router_spec.js`", the only major difference being that we use the Backbone API for making URL changes: `Backbone.history.navigate({trigger: true})`. The effect of each navigation change is tested using mocking, as before. Since we are now dealing with the real `location` object rather than passing fake locations into a function, we also have some global state to deal with: `location` is a singleton and should be left in the same state at the end of each test so that the tests do not interfere with one another. So, we note the current value of `location.hash` in a `before()` block, and navigate back to that hash in an `after()` block to put the router back at the 'default' URL.

Figure 4.70. `browser/routing/backbone_router_spec.js`

```
JS.Test.describe("Router", function() { with(this) {
  before(function() { with(this) {
    this.initialLocation = location.hash
  }})

  after(function() { with(this) {
    Backbone.history.navigate(initialLocation)
  }})

  it("routes / to timeline.html", function() { with(this) {
    expect("render").given("timeline.html")
    Backbone.history.navigate("/home", {trigger: true})
  }})

  it("routes /users/jack to profile.html with username=jack", function() { with(this) {
    expect("render").given("profile.html", {username: "jack"})
    Backbone.history.navigate("/users/jack", {trigger: true})
  }})
}})
```

What about `pushState`? We're using the Backbone router so we can use `pushState` in browsers that support it, but our tests seem to assume we're using `location.hash`. And indeed we are: if we call `Backbone.history.start()` with no arguments, that's what we get — you have to opt in to using `pushState` by calling `Backbone.history.start({pushState: true})`. But this will probably not work with the way we're running the tests.

If you open `browser/routing/test.html` by double-clicking on it, or by running the `open` command in a terminal, you will have opened it using the `file:` protocol. The browser's address bar probably displays something like `file:///home/jcoglan/jstr/Code/browser/routing/test.html`. If you try to use `history.pushState()` in such a page, Chrome throws the following error: `SecurityError: A history state object with URL 'file:///home' cannot be created in a document with origin 'null'`. The current URL does not have an *origin*, which is a combination of a protocol, hostname and port as shown in Figure 4.60, "Structure of the `location` object". For security reasons, the history API can only be used on pages with an origin, meaning they're served from a hostname and accessed with the `http:` or `https:` protocols.

If you want to test `pushState` explicitly, you need to run the tests via a hostname. You can start up a local HTTP server to do this; there are many ways to start a local static file server but this is one of the easiest;

open a terminal in the `Code` directory and run `python -m SimpleHTTPServer`. This starts an HTTP server on `localhost:8000` that serves the contents of the current directory. You can now access the tests using the URL `http://localhost:8000/browser/routing/test.html`. You can then change the command shown in Figure 4.69, “Starting a Backbone router” to `Backbone.history.start({pushState: true})` to force Backbone to use `history.pushState()` in supporting browsers. You will also need to replace `location.hash` with `location.pathname` in the `before()` block in Figure 4.70, “`browser/routing/backbone_router_spec.js`”, otherwise the `after()` block will set the URL to `http://localhost:8000/`, which won’t give you the test suite again when you refresh the page.

Although it is possible to test location-based logic as we’ve shown here, the fact that it’s tricky to set up correctly while avoiding global state problems should give you a hint that you shouldn’t couple too much application logic to it. Your routers should only contain very short methods that extract data from the current location and call some other function in the application to process that data. Those parts of the app responsible for actually displaying the resulting view can then be tested separately, without getting the `location` and `history` APIs involved. Where those UIs need to make navigation calls, those can be tested by setting mock expectations on `Backbone.history.navigate()`.

4.6. Using the timer functions

Throughout this chapter, we have seen that most of the APIs we use to interact with the browser environment can be stubbed out in order for us to run tests. Most of the APIs we would like to stub out are simple function calls that return a single piece of data, and most of the mock expectations we set up are designed to check a single well-defined command being sent to some object. Although you will find customised fakes for various things, like Sinon’s fake Ajax server, I would encourage you to use the generic stubbing APIs built into your chosen test framework to deal with such problems.

However, some systems your code interacts with will be complex enough to merit using a custom fake implementation, rather than relying on generic stubs. The timer functions — `setInterval()`, `clearInterval()`, `setTimeout()` and `clearTimeout()` — are a good example of these; their role is to schedule your code to run at specific times and in a specific order. They deal with control flow rather than simple data exchange, often invoking their callbacks multiple times, and are thus quite hard to write correct stubs for.

Let’s look at a simple example: a class called `Ticker` whose job is to take a list of messages and render them to the page one at a time using a `setInterval()` loop.

Figure 4.71. `browser/timers/ticker.js`

```
var Ticker = function(selector, messages) {
  this._element  = $(selector)
  this._messages = messages
  this._index    = 0

  var self = this
  this._timer = setInterval(function() { self.update() }, 1000)
}

Ticker.prototype.update = function() {
  if (this._index === this._messages.length) return this.stop()

  this._element.append("<li>" + this._messages[this._index] + "</li>")
  this._index += 1
}

Ticker.prototype.stop = function() {
  clearInterval(this._timer)
  delete this._timer
}
```

Now, we could of course test this by writing some code that uses `setTimeout()` to check the state of the page once per second, checking that the correct messages are shown. `jstest` lets us have multiple asynchronous steps per test; just like an `it()` block, the function we pass to `resume()` can also take a parameter that tells the framework the function generates some async work. Just as for `it()` blocks, the parameter is a `resume()` function that we should invoke when the async work is done. So, we can write a sequence of `setTimeout()` blocks to check that `Ticker` is behaving correctly. We need an `after()` block that stops the ticker, so it doesn't continue to modify the page after each test has finished, and we leave the first timeout a little later than we expect the first message to show up, to allow the browser enough time to display it.

Figure 4.72. `browser/timers/ticker_slow_spec.js`

```
JS.Test.describe("Ticker (slow)", function() { with(this) {
  extend(HtmlFixture)

  fixture(' <ul class="test-list"></ul> ')

  before(function() { with(this) {
    this.ticker = new Ticker(".test-list", ["Uno", "Dos", "Tres"])
  }})

  after(function() { with(this) {
    ticker.stop()
  }})

  it("shows no messages at first", function() { with(this) {
    var messages = $.map(fixture.find("li"), function(li) { return $(li).text() })
    assertEqual( [], messages )
  }})

  it("reveals one message each second", function(resume) { with(this) {
    setTimeout(function() {
      resume(function(resume) {
        var messages = $.map(fixture.find("li"), function(li) { return $(li).text() })
        assertEqual( ["Uno"], messages )

        setTimeout(function() {
          resume(function(resume) {
            messages = $.map(fixture.find("li"), function(li) { return $(li).text() })
            assertEqual( ["Uno", "Dos"], messages )

            setTimeout(function() {
              resume(function() {
                messages = $.map(fixture.find("li"), function(li) { return $(li).text() })
                assertEqual( ["Uno", "Dos", "Tres"], messages )
              })
            }, 1000)
          })
        }, 1000)
      })
    }, 1100)
  }})
}})
```

This is some fairly ugly code, and it doesn't get any better when we test more complex time-based logic. But an even worse problem is that the tests are slow: if we must wait a certain amount of time for some logic to trigger, our tests will take at least that much time for each test we write for that logic. We don't want to spend all day waiting for slow tests, that'll ruin our productivity! We need a better way to test time-based logic that doesn't slow us down.

Unfortunately, generic stubbing techniques are not usually enough here. Time-based logic usually relies on things happening in a certain order, and it can be hard to write stubs for the scheduling functions that correctly simulate what will happen when we run the code for real. For these situations, `jstest` and Sinon provide a ‘fake clock’, or to be more precise, a fake scheduler. They replace the timer functions with fake versions that, instead of scheduling callbacks based on the real clock time, are scheduled by a fake clock that the user can control. Rather than waiting for the right amount of time to physically pass, our tests can call a function that manually moves the scheduler’s clock forward, triggering any scheduled callbacks along the way.

In `jstest`, this functionality is available by adding `include(JS.Test.FakeClock)` to a test suite. This makes an object called `clock` available in the tests; call `clock.stub()` to stub out the timer functions, `clock.tick(n)` to go forward `n` milliseconds, and `clock.reset()` to reinstate the real timer functions. We can rewrite the above test using fake clock, as follows:

Figure 4.73. *`browser/timers/ticker_spec.js`*

```
JS.Test.describe("Ticker", function() { with(this) {
  extend(HtmlFixture)

  fixture(' <ul class="test-list"></ul> ')

  include(JS.Test.FakeClock)

  before(function() { this.clock.stub() })
  after(function() { this.clock.reset() })

  before(function() { with(this) {
    this.ticker = new Ticker(".test-list", ["Uno", "Dos", "Tres"])
  }})

  it("shows no messages at first", function() { with(this) {
    var messages = $.map(fixture.find("li"), function(li) { return $(li).text() })
    assertEqual( [], messages )
  }})

  it("reveals one message each second", function() { with(this) {
    clock.tick(1100)
    var messages = $.map(fixture.find("li"), function(li) { return $(li).text() })
    assertEqual( ["Uno"], messages )

    clock.tick(1000)
    messages = $.map(fixture.find("li"), function(li) { return $(li).text() })
    assertEqual( ["Uno", "Dos"], messages )

    clock.tick(1000)
    messages = $.map(fixture.find("li"), function(li) { return $(li).text() })
    assertEqual( ["Uno", "Dos", "Tres"], messages )
  }})
}})
```

You’ll notice that the `setTimeout()` calls have been replaced by `clock.tick()` and that the tests are no longer asynchronous. Although they trick the code into believing that three seconds have passed, in reality the test runs almost instantaneously. Using a fake clock makes it much easier and faster to test timer-based code, knowing that the code is still executing in the order it would in real life.

However, you may run into problems, as you will whenever you replace a global API like `setTimeout()` with a fake. Activating a fake clock does not only affect the component you’re testing, it affects *all* code that depends on these functions. For example, `jstest` uses `setTimeout()` to detect async tests that have become unresponsive, and to loop through all the tests without blocking for long periods of time.

It has been specially constructed to make sure it always uses the real version, so that activating the fake clock does not affect it. Most code in the wild will not take such precautions, and you may find your system becomes unresponsive.

To avoid such problems, try to have as little code in play as possible while the fake clock is active. Keep the responsibilities of timer-based code small, and don't put large amounts of business logic inside timer callbacks. If a timer calls out to a function that encapsulates business logic, then that function can be tested on its own without invoking the timer, and this decoupling can make a system much easier to test.

4.7. Working with build tools

So far, the examples in this chapter have been based almost entirely on static files; static HTML pages using `<script>` tags to load static JavaScript source code. I built the examples this way so you don't need to understand any additional tools in order to see how the tests work. But real-world projects don't typically look like this: the code we serve to the public is usually generated from source code via a number of steps. Templates need to be precompiled, maybe we need to turn our CoffeeScript[39] into JavaScript, we need to bundle all our modules into a single file, the code needs minifying so it loads quickly over the internet. Fortunately, these concerns are mostly orthogonal to the business of writing and running tests, but it is worth mentioning a few common concerns when it comes to build processes.

4.7.1. Precompiling templates

In Section 4.1.2, "Extracting markup for tests" we discussed using template languages like Handlebars in your application. For performance reasons it's often useful to turn our template files into executable JavaScript ahead of time rather that loading and parsing the template source at runtime. You'll need to make sure any compiled code is up to date before executing your tests, and to make this easy it's often worth putting the steps to compile your templates into an automated build system.

If you'd prefer to write tasks in JavaScript, Grunt[40] is very popular, but since this is a simple shell operation Make[41] works just fine. Here's the Makefile for the to-do list app we explored earlier.

Figure 4.74. `browser/todo_list/Makefile`

```
PATH := ../../node_modules/.bin:$(PATH)

templates/templates.js: templates/*.handlebars
	handlebars templates/*.handlebars > templates/templates.js
```

This `Makefile` puts the executables from npm on the `PATH`, and says that `templates/templates.js` is derived from `templates/*.handlebars`, using the `handlebars` program to convert the templates to JavaScript. Running `make` in a directory will run the first task in that directory's `Makefile`, which in this case will build the templates, or do nothing if the compiled JavaScript is up to date.

Makefiles can become quite repetitive, especially as more file dependencies are added, so Make lets you use variables to DRY things up a bit. It's common to have a task called `all`, which builds all your project files and is the first (and therefore default) task; we define it by making `all` depend on all the files that need to be built. We also have a task called `test` that runs whatever commands are necessary to test the project[42]; this depends on `all` so that running `make test` will make sure all generated files are up to date before running the tests. Here's a `Makefile` that uses all these features:

[39]http://coffeescript.org/
[40]http://gruntjs.com/
[41]http://www.gnu.org/software/make/
[42]For information on the `phantomjs` command, see Section 4.8.1, "Running tests on PhantomJS".

Figure 4.75. `browser/form_validation/Makefile`

```
PATH := ../../node_modules/.bin:$(PATH)

template_js     := templates/templates.js
template_source := templates/*.handlebars

.PHONY: all test

all: $(template_js)

test: all
	phantomjs ../../phantom.js test.html

$(template_js): $(template_source)
	handlebars $(template_source) > $(template_js)
```

The `all` and `test` tasks are listed as ‘phony’; this simply means that the task names do not refer to files, so if there is a file called `all` or `test` then Make will still run these tasks even though there are up-to-date files for them.

You should think of the `make` command as doing whatever it takes to prepare your code for testing, from the state it exists in when you clone it from Git up to the point it’s completely ready to execute. By describing the relationships between files in your project and how they are derived from one another, you make it easy for Make to figure out which files are already up to date, skipping compiler steps that don’t need to be done and saving you time.

4.7.2. CommonJS modules and Browserify

Many modern apps are organised using CommonJS modules and compiled for the browser using Browserify[43]. CommonJS is the module system adopted (and extended) by Node[44], and it allows components to explicitly define both their dependencies and their public API, rather than relying on global variables. Dependencies are loaded using `require(filename)`, which evaluates `filename` and returns the object or function that `filename` assigns to `module.exports`. Here’s a typical module:

Figure 4.76. `browser/commonjs/lib/concert_view.js`

```
var handlebars = require("handlebars"),
    templates  = require("../templates/templates")

var ConcertView = function(selector, concert) {
  this._element = $(selector)
  this._concert = concert

  var self = this
  concert.on("change", function() { self.render() })

  this.render()
}

ConcertView.prototype.render = function() {
  var html = handlebars.templates.concert(this._concert.attributes)
  this._element.html(html)
}

module.exports = ConcertView
```

If you load this file using `require()` you will get a reference to `ConcertView`, since that’s what the file assigns to `module.exports`. This module system works natively if you’re running Node, but it doesn’t

[43]http://browserify.org/
[44]http://nodejs.org/

work in the browser. Instead, you need to compile your app using Browserify, which resolves all the dependencies in your application and creates one big JavaScript file containing the whole app, which is what you load in the browser. A typical projcect defines a ‘main’ file that acts as the entry point for the app, wiring all the components together and rendering the initial view. For example, here’s a main script that creates a concert and renders it on the page:

Figure 4.77. `browser/commonjs/lib/main.js`

```
var Concert = require("./concert"),
    View    = require("./concert_view")

var model = new Concert({
  artist:     "Mogwai",
  venueName: "Royal Festival Hall",
  cityName:   "London",
  country:    "UK"
})

new View(".concert", model)
```

Compiling this script using Browserify will then resolve all the dependencies referenced in it and generate a single bundle file, ready to be loaded into the browser.

Figure 4.78. Building an app using Browserify

```
$ browserify lib/main.js > app.js
```

If you want to test code that’s managed using Browserify, you need your tests to use the same module system: since none of the components you need to refer to are globals any more, the tests will need to use `require()` to load them, and will therefore need to go through the same build system. Here’s a simple test suite for the above `ConcertView` module:

Figure 4.79. `browser/commonjs/spec/concert_view_spec.js`

```
var JS      = require("jstest"),
    Concert = require("../lib/concert"),
    View    = require("../lib/concert_view")

JS.Test.describe("ConcertView", function() { with(this) {
  extend(HtmlFixture)
  fixture(' <div class="concert"></div> ')

  before(function() { with(this) {
    this.concert = new Concert({
      artist:     "Boredoms",
      venueName: "The Forum",
      cityName:   "Kentish Town",
      country:    "UK"
    })
    new View(".concert", concert)
  }})

  it("renders the artist name", function() { with(this) {
    assertEqual( "Boredoms", fixture.find(".artist").text() )
  }})

  it("updates the artist name if it changes", function() { with(this) {
    concert.set("artist", "Low")
    assertEqual( "Low", fixture.find(".artist").text() )
  }})
}})
```

Just as we have a `main.js` for our app, we also need a `main.js` for our tests, which loads all the test suites and executes them:

Figure 4.80. `browser/commonjs/spec/main.js`

```
var JS = require("jstest")

require("./concert_template_spec")
require("./concert_view_spec")

JS.Test.autorun()
```

Running Browserify on this script will then produce a file that loads and runs all the tests[45].

Figure 4.81. Building a test suite using Browserify

```
$ browserify spec/main.js --noparse node_modules/jstest/jstest.js > test.js
```

This script can be loaded into a web page to view the results of the tests. Remember that it contains all the application code as well as the tests, and will need to be regenerated whenever any of the project files change. We can use Make to enforce this for us: by making a `test` task that depends on `test.js`, which in turn depends on `lib/*.js`, `spec/*.js` and the compiled templates, we can make sure the test bundle is rebuilt when any source, test or template file changes. Every rule in this `Makefile` says which files each target is generated from, so Make knows which targets need updating when we change any project file.

In the following examples we've used a few special Make variables: `$@` means 'the file the current task generates', and `$^` means 'the list of files this task depends on', and `$<` means 'the first file this task depends on'. They provide a terse way to refer to files in common use cases.

Figure 4.82. `browser/commonjs/Makefile`

```
SHELL := /bin/bash
PATH  := ../../node_modules/.bin:$(PATH)

app_bundle      := app.js
lib_files       := lib/*.js
template_js     := templates/templates.js
template_source := templates/*.handlebars
spec_files      := spec/*.js
test_bundle     := test.js
jstest          := $(shell readlink -f ../../node_modules/jstest/jstest.js)

.PHONY: all test

all: $(app_bundle)

$(app_bundle): $(lib_files) $(template_js)
	browserify lib/main.js > $@

$(template_js): $(template_source)
	handlebars -c handlebars $^ > $@

test: $(test_bundle)
	phantomjs ../../phantom.js test.html

$(test_bundle): $(lib_files) $(template_js) $(spec_files)
	browserify spec/main.js --noparse $(jstest) > $@
```

Using a module system can pose problems when we want to stub things: because components now refer to one another using `require()`, rather than global variables, we cannot stub out global variables in order to make components use our fake versions. However, two `require()` calls for the same file will return the same object, so if your tests can refer to a module, they *can* replace the methods it exports by using stubbing, for example:

[45]We use `--noparse` to stop Browserify spidering the `require()` calls in `jstest`; for historical reasons it contains references to modules from other platforms that Browserify doesn't recognise.

Figure 4.83. Stubbing the API of a CommonJS module

```
var MyModel = require("./path/to/model")
stub(Model, "create").returns(myFakeModel)
```

The code under test will now receive `myFakeModel` if it calls `MyModel.create()`, but `MyModel` *itself* cannot be replaced with something else. In practice this is not a big problem if you design your modules with this use case in mind, but if you find yourself needing to replace modules, consider using dependency injection[46] to pass modules in, rather than using hard-coded `require()` dependencies in your code.

4.7.3. Minification and other build tasks

You can mix any other build steps into your Makefile, including compiling CoffeeScript, converting ECMAScript 6 modules to present-day JavaScript, using Uglify[47] to concatenate and compress your source code, and so on. If you use Make to describe how your build files depend on the source files, and which commands to use to regenerate out-of-date files, you should be able to come up with a pleasant workflow for making sure your files are up to date before testing them.

The examples contain a directory called `browser/coffee_uglify`, which is a copy of `browser/commonjs` done in CoffeeScript. Its source code is in `lib/*.coffee`, its templates are in `templates/*.handlebars`, and its tests in `spec/*.coffee`. The following `Makefile` compiles all the CoffeeScript and Handlebars into the `build` directory, generates a single app bundle using Uglify, and runs the tests using PhantomJS. Notice how the dependencies between files are set up so that the code being tested is always up to date[48].

Figure 4.84. `browser/coffee_uglify/Makefile`

```
PATH := ../../node_modules/.bin:$(PATH)

source_files    := $(wildcard lib/*.coffee)
build_files     := $(patsubst lib/%.coffee,build/lib/%.js,$(source_files))
template_source := templates/*.handlebars
template_js     := build/templates.js
app_bundle      := build/app.js
spec_coffee     := $(wildcard spec/*.coffee)
spec_js         := $(patsubst spec/%.coffee,build/spec/%.js,$(spec_coffee))

.PHONY: all clean test

all: $(app_bundle)

build/%.js: %.coffee
        coffee -co $(dir $@) $<

$(template_js): $(template_source)
        mkdir -p $(dir $@)
        handlebars $^ > $@

$(app_bundle): $(build_files) $(template_js)
        uglifyjs -cmo $@ $^

test: all $(spec_js)
        phantomjs ../../phantom.js test.html

clean:
        rm -rf build
```

[46]See Section 4.2.2, “Dependency injection”.
[47]https://npmjs.org/package/uglify-js
[48]For a full explanation of the Make features used here, see http://blog.jcoglan.com/2014/02/05/building-javascript-projects-with-make/.

Whichever module system you use, whether it be CommonJS, AMD, ECMAScript 6, or the modules in a compile-to-JS language, remember that hard-coded dependencies can cause headaches when testing. Design your components to allow collaborating objects to be passed in as arguments rather than trying to stub things out through the module system; this avoids the need for specialised stubbing tools that hack the module system into doing things it never intended.

The minifying tools we have these days — especially Uglify — are very good and will usually not introduce bugs into your code unless you explicitly ask them to use unsafe optimisations. That said, I usually run my tests against my minified build files, to make sure that the code I'm actually shipping to users has been tested. This is not a hard-and-fast rule, and it can be relaxed if it removes the need for extra build steps needed to run the tests, but I find it's worth the extra effort to get this pipeline set up.

4.8. Running browser tests

We've spent plenty of time discussing how to write tests for the browser, but very little time discussing how to run them. While a full treatment of browser test runners is beyond the scope of this book, it is worth briefly mentioning a few useful tools.

Many people find the process of manually running their tests in all the browsers they want to support to be frustrating. This becomes even more true when it requires browsers that aren't on your development machine; indeed the code for this book is tested on over a dozen browsers across Windows, OS X, Linux and Android to make sure the examples work everywhere. I don't do this by hand, I have tools to help me, and you can save yourself a lot of time by knowing some of these.

4.8.1. Running tests on PhantomJS

The biggest time-saver you can get while working on your code is to make it as fast as possible to run your tests. One way to achieve that is to run your tests via the terminal rather than switching over to your browser, and PhantomJS[49] makes this possible. PhantomJS is a real WebKit-based web browser, only it doesn't have a graphical user interface. Instead, you run it by writing scripts in JavaScript that tell it what to do. For example, the examples that come with this book include this script that opens any of the `test.html` example pages:

Figure 4.85. `phantom.js`

```
// This is a PhantomJS script that can run any of the browser-based examples
// and display the results. You can run it like so:
//
//     $ phantomjs phantom.js browser/hello_world/test.html
//
// Run it with any of the `test.html` files in this tree. You can download
// PhantomJS from http://phantomjs.org/.

var system = require("system"),
    JS     = require("./node_modules/jstest/jstest"),
    file   = system.args[1]

if (file === undefined) {
  console.error("You need to give me an HTML file to run!")
  phantom.exit(1)
}

var reporter = new JS.Test.Reporters.Headless({})
reporter.open(file)
```

[49]http://phantomjs.org/

`jstest` has support for PhantomJS built in, meaning that it can detect when your page is running on PhantomJS and print the test results as text in the terminal using various output formats. For example, here's what happens when we run the tests from Section 4.4, "Storing data" using the `phantomjs` command, setting different output formats via the `jstest` `FORMAT` setting:

Figure 4.86. Running tests via PhantomJS with different output formats

```
$ phantomjs phantom.js browser/local_storage/test.html
Loaded suite: TodoStore, TodoStore.save(), TodoStore.load(), TodoStore.remove()

.................

Finished in 0.046 seconds
17 tests, 18 assertions, 0 failures, 0 errors

$ FORMAT=tap phantomjs phantom.js browser/local_storage/test.html
1..17
ok 1 - TodoStore save() assigns a sequential ID to new items
ok 2 - TodoStore save() does not change the ID of existing items
ok 3 - TodoStore save() emits a 'create' event for new items
ok 4 - TodoStore save() emits an 'update' event for existing items
ok 5 - TodoStore load() emits a 'load' event with the data for the ID
ok 6 - TodoStore remove() deletes the item from the collection
ok 7 - TodoStore remove() emits a 'remove' event with the data for the ID
ok 8 - TodoStore.save() saving a new item records the item ID
ok 9 - TodoStore.save() saving a new item saves the item as JSON
ok 10 - TodoStore.save() saving an existing item does not change the item ID
ok 11 - TodoStore.save() saving an existing item updates the stored item
ok 12 - TodoStore.load() does not emit a 'load' event for unknown IDS
ok 13 - TodoStore.load() emits a 'load' event with an existing item
ok 14 - TodoStore.remove() does not emit a 'remove' event if the item does not exist
ok 15 - TodoStore.remove() emits a 'remove' event if the item exists
ok 16 - TodoStore.remove() tells localStorage to look up an item
ok 17 - TodoStore.remove() tells localStorage to remove an item
```

If we go and deliberately break some of the code in this example, we'll see an error when we run the tests and PhantomJS will exit with a non-zero status; this is the Unix way for a program to indicate that there was an error during its execution.

Figure 4.87. Running a broken test suite with PhantomJS

```
$ phantomjs phantom.js browser/local_storage/test.html
Loaded suite: TodoStore, TodoStore.save(), TodoStore.load(), TodoStore.remove()

...F.............

1) Failure: TodoStore save() emits an 'update' event for existing items
Mock expectation not met
<{ "trigger": #function }> expected to receive call
trigger( "update", { "id": 1, "title": "Renew passport" } )

Finished in 0.044 seconds
17 tests, 18 assertions, 1 failure, 0 errors

$ echo $?
1
```

While `jstest` wires all this up automatically for you, it's quite simple to do using other frameworks. What's going on behind the scenes is that `jstest` is opening your HTML page and telling PhantomJS to pass through anything the page writes to the console, using code like this:

Figure 4.88. Listening to console output from PhantomJS

```
var page = new WebPage()

page.onConsoleMessage = function(message) {
  // process message
}
page.open("browser/local_storage/test.html")
```

The copy of `jstest` that's running on that web page detects PhantomJS using the value of `navigator.userAgent`, and uses `console.log()` to write JSON messages describing the progress of the tests as they happen. The `onConsoleMessage()` listener picks these messages up, reformats them to readable text, or TAP, or XML, or whatever format you've asked for, and finally uses `phantom.exit()` to set the exit status. You can use exactly the same approach with any other framework that lets you plug in new reporters.

Using PhantomJS makes it quicker to run tests while you're working on the code, and also allows you to run JavaScript tests on your CI server[50], where there is typipcally no desktop environment and test commands must be able to run in a shell.

Before PhantomJS came along, the only way to run browser tests in the shell was to use one of the many 'fake browser' alternatives, like HtmlUnit[51], Envjs[52] or Harmony[53]. These are all essentially brand new browser and DOM implementations based on an embedded JavaScript engine like SpiderMonkey or Rhino. They all have fairly obvious bugs, and while useful as a sanity check for your code, they provide little utility for cross-browser testing since they're all just another browser engine with their own set of issues. PhantomJS is at least based on WebKit, a real browser engine, but it is possible to find quirks in it that you won't see in Chrome or Safari. It's a great tool but it doesn't quite replace true cross-browser testing. For that, we need different tools.

4.8.2. TestSwarm: cross-browser CI

A few years ago, frustrated with the pain of testing their code on dozens of different browsers, the jQuery team developed TestSwarm[54]. TestSwarm works like many other CI servers but is tailored to running JavaScript. When you submit a job, you include a set of URLs — these are the pages containing the tests you want to run — and a list of browsers you want the tests to run against. TestSwarm then runs your tests on the given browsers whenever they are connected to the server — anyone whose browser can access the server can run tests for you, letting you crowd-source the platforms you don't have in-house.

Setting up TestSwarm is a little tricky and beyond the scope of this book, but their docs are complete enough to get a working server going if you follow them closely. `jstest` integrates with TestSwarm out of the box, but writing your own reporter for another framework is relatively easy.

Because it's based around submitting jobs using lists of test page URLs, TestSwarm is suited to after-commit testing rather than the on-the-fly tests you run during development. A typical workflow is to poll your Git repo for new commits, check each commit out into its own directory on a public webserver, run any build steps you need to get the code ready, then submit a list of the public URLs of your test pages to your TestSwarm server. Although it's less interactive, it provides a permanent record of which platforms each commit's tests have run on, and lets you run tests on browsers you don't have on your local machine during development. I am using it to check this book's examples across four different operating systems, and it's greatly reduced the effort required to make sure everything works.

[50]CI stands for *continuous integration*. Many teams have a server that runs all the tests for a project when anyone pushes a commit. Jenkins (http://jenkins-ci.org/) is a popular self-hosted CI server, while for open source Travis (https://travis-ci.org/) offers an excellent hosted solution.
[51]http://htmlunit.sourceforge.net/
[52]http://www.envjs.com/
[53]https://github.com/mynyml/harmony
[54]https://github.com/jquery/testswarm

4.8.3. Local cross-browser testing: Buster, Karma, Testem

When you’re developing front-end code, you probably have a few different browsers open to check your code in each one. You might even run your code on a local server and have a Windows VM or an iPhone connected to check your changes on those platforms too. It becomes tedious going into each browser and pressing ‘refresh’ for each change you make, so various tools were invented to make your job easier.

There are many projects that provide a *test runner*, a tool that automatcially runs your tests on a multitude of browsers for you. These tools include Buster[55], Karma[56] and Testem[57], and they mostly work by starting a local server that you connect your various browsers to. The server lets you run a single command in your terminal to run your tests on all the connected browsers and see the results all in one place, streamlining your testing workflow.

These tools all have their strengths and weaknesses and you should give them all a spin to see which one suits you. They all require some integration between the runner and your test framework of choice, to feed the test results to the runner in its standard format. `jstest` supports all of them out of the box[58] but adapters for other frameworks are a google search away.

Setting up these tools is beyond the scope of this book but let’s have a look at Testem by way of example. It’s very easy to set up, in fact here’s the config file for the tests from Section 4.4, “Storing data”:

Figure 4.89. `browser/local_storage/testem.json`

```
{"test_page": "browser/local_storage/test.html#testem"}
```

To run Testem, open a terminal in the `Code` directory and type the following command:

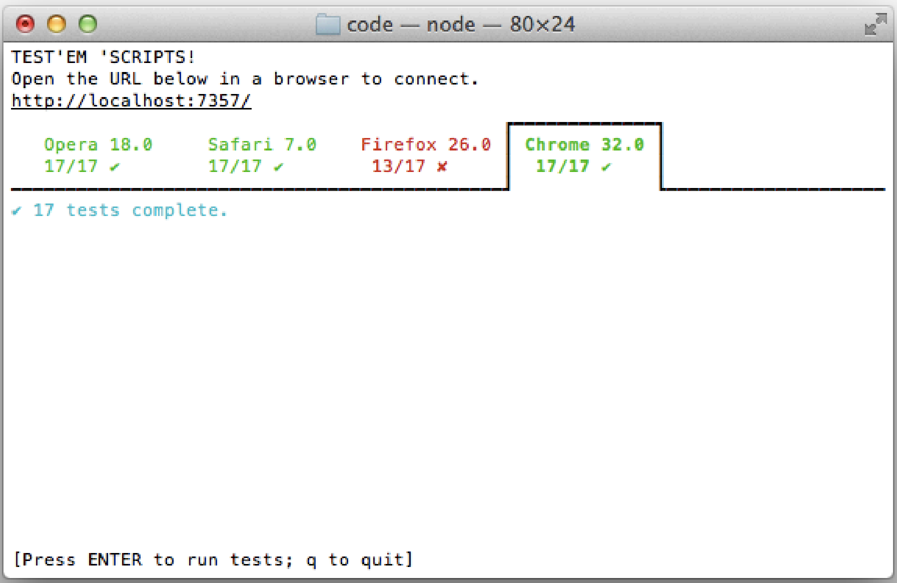
Figure 4.90. Running Testem

```
$ ./node_modules/.bin/testem -f browser/local_storage/testem.json
```

Testem will take over your terminal window and tell you to open `http://localhost:7357/`. Once you’ve done this in a few browsers, you’ll see the results in a tabbed view that you can navigate with the cursor keys. Most browsers display a passing report for these examples:

[55]http://busterjs.org/
[56]http://karma-runner.github.io/
[57]https://www.npmjs.org/package/testem/
[58]See http://jstest.jcoglan.com/browser.html for setup instructions.

Figure 4.91. A passing Testem report

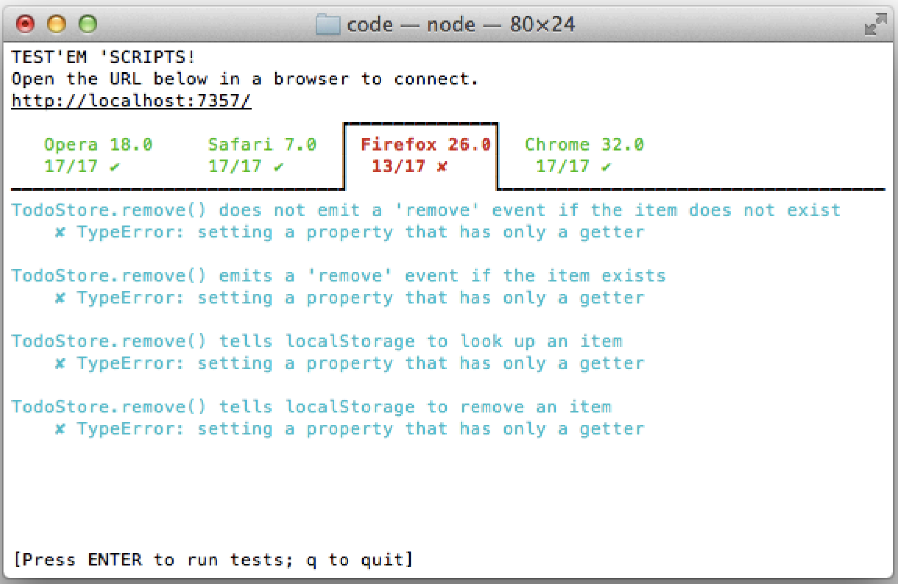


```
TEST'EM 'SCRIPTS!
Open the URL below in a browser to connect.
http://localhost:7357/

   Opera 18.0      Safari 7.0      Firefox 26.0    Chrome 32.0
   17/17 ✔         17/17 ✔          13/17 ✘         17/17 ✔

✔ 17 tests complete.

[Press ENTER to run tests; q to quit]
```

But, one browser is showing up red, and if you navigate over to its tab you can see the errors it's reporting — see Figure 4.92, “A failed Testem report”. In this case, Firefox doesn't allow us to stub out the `localStorage` object, making Figure 4.59, “`browser/local_storage/todo_store_remove_spec.js`” fail. When you change your tests and source code, or press `<enter>` in the terminal, Testem will re-run your tests in all the connected browsers, and you can see at a glance which browsers still have a problem running your code.

Since Testem starts a local HTTP server, you can connect any device on your local network to the server using the local machine's IP address. This lets you run tests on a Windows VM or iOS device in real time, with the results showing up in the terminal as you edit your code. This rapid feedback can really boost your productivity, letting you focus on writing code without the interruption of manually running your tests. I often leave Testem sitting open next to my Vim session and rarely need to run my tests by hand.

Figure 4.92. A failed Testem report



```
TEST'EM 'SCRIPTS!
Open the URL below in a browser to connect.
http://localhost:7357/

   Opera 18.0     Safari 7.0     Firefox 26.0     Chrome 32.0
   17/17 ✔        17/17 ✔        13/17 ✘          17/17 ✔

TodoStore.remove() does not emit a 'remove' event if the item does not exist
    ✘ TypeError: setting a property that has only a getter

TodoStore.remove() emits a 'remove' event if the item exists
    ✘ TypeError: setting a property that has only a getter

TodoStore.remove() tells localStorage to look up an item
    ✘ TypeError: setting a property that has only a getter

TodoStore.remove() tells localStorage to remove an item
    ✘ TypeError: setting a property that has only a getter



[Press ENTER to run tests; q to quit]
```

Testem is unusual among these tools in that it lets you point it at an existing web page, while most other tools ask you for a list of scripts you want to load and then construct the host page themselves. This is why I keep all the fixtures, templates and so on in JavaScript, since I cannot be sure I'll have access to edit the page where the tests are running. In some cases, I will even deal with loading the code and tests using JavaScript so I don't need to duplicate the `<script>` tags from my `test.html` in the test runner's config.

For example, for this set of tests I've stripped `test.html` down to:

Figure 4.93. `browser/local_storage/test.html`

```html
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>jstest</title>
  </head>
  <body>

    <script src="../../node_modules/jstest/jstest.js"></script>
    <script src="./runner.js"></script>

  </body>
</html>
```

while the rest of the code and tests are loaded in `runner.js` using `JS.load()`. You can use any script loader you like, `jstest` happens to have one built in since I do this so often. When I decide to use a runner to execute my tests, I can just point it at this file rather than telling it each individual file I want to load:

Figure 4.94. *`browser/local_storage/runner.js`*

```
JS.load(  "../../node_modules/underscore/underscore.js",
          "../../node_modules/backbone/backbone.js",
          "../../vendor/json2.js",

          "./todo_store.js",
          "./todo_store_spec.js",
          "./todo_store_save_spec.js",
          "./todo_store_load_spec.js",
          "./todo_store_remove_spec.js",

          function() { JS.Test.autorun() })
```

Loading your scripts dynamically has the added advantage that the callback function will not run if any of the scripts fails to load. If a `<script>` tag references a non-existent file, the browser just ignores it, while a script loader will usually not succeed if asked to load a missing file. This makes it more obvious when your test setup is not correct since the whole test suite will fail to start, rather than running and silently skipping over a suite whose name you misspelled.

4.9. Limitations of browser testing

In this chapter we've seen plenty of ways to test various browser components, and how to manage interacting with the browser environment. While many browser UIs are mainly responsible for displaying textual information and updating the structure of the DOM, a lot of what we do in the browser is visual in nature. While it's easy to mechanically check that some text is displayed or that a class name was added to a list item, it's not so easy to check that a UI is layed out correctly or that an animation runs smoothly.

There are visual diff tools available to automate some of these tasks but they tend to serve as warning beacons rather than detailed tests. They're quite good at telling you a page *might* have broken, and you should take a look at it, but not so great at saying if a page is definitely right or wrong. You'll get the best mileage out of your browser tests by trying to keep content and presentation separate: rather than having some code run `element.css({background: "yellow"})` to highlight a selected row, run `element.addClass("selected")` and implement the visual change in CSS. This keeps the state of UI easy to inspect via jQuery, and therefore easier to test.

Remember at all times that testing should be about effectively managing risk and reducing costs. If you're spending a huge amount of time trying to automate some testing and you're finding it hard to write effective tests, consider whether the automation is really saving you time or preventing significant defects. If it's easier to test by hand, make that part of your pre-release manual QA process rather than sinking too much time into confusing and brittle tests.

But also let the pain of writing tests act as feedback: are you finding testing hard because the problem is inherently hard to write tests for (visualisation and animation code are good examples of this), or is it hard because your architecture is getting in the way? Have a go at prototyping a new design from time to time to see if you could improve matters, take what you learn from those prototypes and fold it back into your main project. Testing is most effective when the team keeps adding tests for their changes and keeps the build green, and keeping your architecture easy to test is a big factor in making this happen.

5. Server-side applications

Since the release of Node[1] a few years ago, JavaScript has taken over the server. Developers who've been using JavaScript on the client side for years can now apply their knowledge to building servers, letting them share code between the server and the client.

While Node's design and philosophy differs in many ways from that of other popular dynamic languages such as Python and Ruby, Node developers can still learn a lot from the problems those communities have already faced, especially when it comes to testing. Server-side programming is a relatively new problem domain for many JavaScript developers, and being a very different environment from the browser it comes with its own set of challenges. But as in the browser, we can use a few techniques to make our code easy to test: it all comes down to writing small, modular components with well-defined interfaces, that can be tested independently and glued together easily.

5.1. A basic Node server

Node is designed to make it easy to create servers with very little code. The 'hello world' Node server looks like this; if you run this script with `node` then run `curl http://localhost:1337/`, you'll see it prints `Hello World`.

Figure 5.1. The Node 'Hello World' server

```
var http = require("http")

var server = http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"})
  response.end("Hello World\n")
})

server.listen(1337)
```

We start out by using `require()` to import the `http` module; in Node, we do not use global variables for libraries like we would in the browser, we use `require()` to import a module and assign it to a local variable. Top-level local variables are scoped to the current file, they do not leak into the global namespace.

Then, we use `http.createServer()` to create an HTTP server object. We pass in a function that takes a `request` and a `response`; this function is actually an event listener for the server's `"request"` event. Whenever the server receives a request, this function is invoked to process it. To be more explicit, we could write the server like this:

Figure 5.2. A Node server with an explicit `"request"` event listener

```
var http = require("http")

var server = http.createServer()

server.on("request", function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"})
  response.end("Hello World\n")
})

server.listen(1337)
```

Finally, we call `server.listen()` to start the server listening on a port. This opens the given TCP port on the local machine so the server can start receiving requests; until we call `server.listen()` the server

[1]http://nodejs.org/

is just a value in memory that the outside world cannot connect to. Calling `server.close()` makes the server stop listening on the port, and no more connections can be made to it after that.

This already shows a few interesting points about Node servers: they are regular JavaScript values that can be configured, started and stopped using regular JavaScript code. Rather than being external processes with config files like Apache[2] or nginx[3], they are JavaScript objects that we can easily manipulate in our test suites. They rely on normal JavaScript functions to process requests and emit responses, which are also regular JavaScript objects. This means we can easily fake requests and make assertions about responses using simple objects and function calls, maybe throwing in some stubbing or mocking. Or, we can start the server up for real and use an HTTP client to inspect it from the outside, while also being able to stub out internal systems. All this gives us a lot of power when it comes to server-side testing.

Let's examine a few different testing approaches using a simple server. This server is marginally more complex than the 'hello world' one; it parses the request's query string parameters, and returns them in JSON format using the appropriate Content-Type header. The call `JSON.stringify(data, null, 2)` returns `data` as pretty-printed JSON with 2-space indents.

Figure 5.3. `node/basic_server/server.js`

```
var http = require("http"),
    url  = require("url")

var server = http.createServer(function(request, response) {
  var query = url.parse(request.url, true).query
  response.writeHead(200, {"Content-Type": "application/json"})
  response.end(JSON.stringify(query, true, 2))
})

module.exports = server
```

Notice that instead of starting the server using `server.listen()` here, we're using `module.exports = server` to export the server as a module. That's because we want to be able to load the server as a value into our testing scripts, and don't necessarily want the server to be started up. Having things listening on ports is global system state, which is best avoided for testing purposes. By doing this, we can load the server from other scripts and make the decision about when to start it up there.

`module.exports` is Node's way of defining module interfaces. When you use `require()` to import a module, the value of that `require()` expression is whatever that file assigns to `module.exports`. So in this case, `require("server")` would return the `server` object.

5.1.1. Black-box server testing

To get started, let's write a black-box test. This means that, instead of testing the server by invoking its request handler function directly, we'll start the server listening on a port and then use an HTTP client to make requests to it. This lets us check the server's behaviour from the point of view of the outside world, which is often where we want to start when specifying what it should do.

For this section we're going to use the techniques described in Section 2.7.5, "Async stories with `jstest`" to write terse asynchronous tests. Black-box server testing involves a lot of async steps: starting and stopping the server, making one or more requests, checking the output of each one, and possibly dealing with streams. Since the purpose of tests is to reduce risk, it's important that they are easy to read and understand. Code with lots of nested callbacks is a pain to read and write, and it's worth using

[2]http://httpd.apache.org/
[3]http://nginx.org/

some tools to simplify things. There are many different approaches to this described in Section 2.7, “Asynchronous tests”, but I will be using the one that’s baked into `jstest`.

We start by writing a test suite that’s a list of high-level steps we want to run. We need to start the server before each test, and shut it down afterward. For our test, we want to send a request with a query string, check that we get a `200 OK` response, and check that the response is a JSON document containing the data we passed in.

Figure 5.4. `node/basic_server/server_spec.js`

```
var JS     = require("jstest"),
    server = require("./server"),
    steps  = require("./server_steps")

JS.Test.describe("server", function() { with(this) {
  include(steps)

  before(function() { with(this) {
    startServer(server)
  }})

  after(function() { with(this) {
    stopServer()
  }})

  it("returns the query parameters as JSON", function() { with(this) {
    get("/", {hello: "world", something: "else"})
    checkStatus(200)
    checkJSON({hello: "world", something: "else"})
  }})
}})
```

Once we know the steps we need for our test, we can implement them. We need to implement the `server_steps` module that we’ve used in this test suite via `include(steps)`. In `jstest`, this works by defining a module using `JS.Test.asyncSteps()` containing a method named after each step in the tests. The methods will take the arguments used in the tests, as well as a callback so each method can indicate when it’s complete. `jstest` deals with sequencing the steps in the tests correctly without us needing to use callbacks explicitly.

We know we need these five methods: `startServer(server)`, `stopServer()`, `get(path, params)`, `checkStatus(status)`, and `checkJSON(data)`, each with a callback. We’re going to start the server afresh for each test; Node servers start up very quickly so that’s not going to pose a problem for execution speed. Rather than hard-coding a port for the tests to run on, I tend to choose one dynamically; this avoids problems where I happen to have started some other server running on the same port the tests want to use. We can use `server.listen(0)` to tell Node to pick an available port for us; we can then discover which port was chosen using `server.address().port`. The `startServer()` method needs to store this port number so that the requests we make are sent to the right place.

We’re using the `request`[4] module to make our HTTP requests; you could use the `http`[5] module that’s bundled with Node, but `request` has a more terse API. `http` also exposes request/response bodies as streams, while `request` buffers the body into a single string for you. If you use the `http` module, consider using `concat-stream` to collect the response body as discussed in Section 3.4.2, “Checking a readable stream’s output”.

Note how the `get()` method stores the response it gets for use by later steps. Separating the request logic from the assertion logic means that these steps can be composed in various different ways in later tests.

[4]https://www.npmjs.org/package/request
[5]http://nodejs.org/api/http.html

Also, notice how the Content-Type assertion is not too coupled to the server implementation; we know the server actually returns `Content-Type: application/json`, but under HTTP conventions it could also return `Content-Type: application/json; charset=utf-8` and remain valid. The test doesn't really mind which one is used as long as the MIME type is JSON.

Finally it is worth noting that the assertion on the body data does not use string comparison, it uses `JSON.parse()` and checks the result against a data structure. This is done so that we know the response is actually a valid JSON document, and the test is not sensitive to the order in which the document's properties happen to be serialised.

Figure 5.5. `node/basic_server/server_steps.js`

```
var JS      = require("jstest"),
    request = require("request")

var ServerSteps = JS.Test.asyncSteps({
  startServer: function(server, callback) {
    this.server = server
    this.server.listen(0, callback)

    this.port   = this.server.address().port
    this.origin = "http://localhost:" + this.port
  },

  stopServer: function(callback) {
    this.server.close(callback)
  },

  get: function(path, params, callback) {
    var url  = this.origin + path,
        self = this

    request.get(url, {qs: params}, function(error, response, body) {
      self.error         = error
      self.response      = response
      self.response.body = body

      callback()
    })
  },

  checkStatus: function(status, callback) {
    this.assertEqual(status, this.response.statusCode)
    callback()
  },

  checkJSON: function(data, callback) {
    var type = this.response.headers["content-type"].split(";")[0]
    this.assertEqual("application/json", type)
    this.assertEqual(data, JSON.parse(this.response.body))
    callback()
  }
})

module.exports = ServerSteps
```

This is already a fairly complete set of steps for doing arbitrary testing of any server we like. We have methods to start and stop the server, make GET requests, and check some details of the response. Add in some methods to make other kinds of requests, and to set and check request and response headers, and you've got all you need to interact with any kind of HTTP service. This is how I commonly test the HTTP interface to apps I build, but there are other approaches.

5.1.2. Mock-based server testing

Although black-box testing is reasonably fast, in some scenarios the cost of starting and stopping the server, sending requests over the wire and waiting for responses can begin to slow your tests down. If you find yourself in such a situation, you might consider mock-based testing.

In our example we know that the `"request"` event listener is a simple JavaScript function; it takes a `request` object whose `url` property is a string, and a `response` object with `writeHead()` and `end()` methods. The code doesn't 'know' that these represent an HTTP request cycle, it only cares that the objects given to the function have interfaces that it knows how to talk to. You could pass in *any* object with a `url` property as the `request` parameter and the function would still work. If you can identify the interfaces the function you're testing expects to talk to, you can usually provide fake versions of those interfaces and use them to drive the code.

It's fairly easy to construct a fake request object with the properties we want; `method` and `url` are just strings, and `headers` is an object with string keys and values. To test the response, we can mock the `response` methods to check the server sends the right values to them.

In the following test, we make a fake `request` object that only has a `url` property, since that's the only `request` property this server actually uses. We also make a fake `response` object, and set some mock expectations on it that express the status, headers and body that should be sent back to the client. Finally, we invoke the request-handling function by making the `server` emit a `"request"` event with our fake inputs; you could also store the request-handling function in its own module and invoke it directly, but this approach involves little overhead and I prefer having the code signal my intent by wrapping the handler in `http.createServer()`.

Figure 5.6. `node/basic_server/server_mock_spec.js`

```
var JS     = require("jstest"),
    server = require("./server")

JS.Test.describe("server (mocking)", function() { with(this) {
  it("returns the query parameters as JSON", function() { with(this) {
    var request  = {url: "/?hello=world&something=else"},
        response = {}

    expect(response, "writeHead").given(200, {"Content-Type": "application/json"})
    expect(response, "end").given('{\n  "hello": "world",\n  "something": "else"\n}')

    server.emit("request", request, response)
  }})
}})
```

Although correct, this test has some potential maintenance problems. It's very highly coupled to the exact way the server emits its response; for example if the server sent other headers apart from `Content-Type`, or sent `content-type` in lowercase[6], this test would fail even though the server would still be working correctly for all practical purposes. We'd really like to specify that the server should emit a header with a name equivalent to `Content-Type`, and may emit other headers we don't care about.

The test is also highly coupled to the serialisation of the JSON emitted by the server; if the order of properties in the document changes, or the indentation or other formatting is changed, the client talking to this server probably won't care but this test would fail. There's little value in tests that fail for reasons we don't care about in practice, so we should make the test more forgiving.

[6]HTTP header names are case-insensitive; Content-Type is the canonical format but content-type, Content-type and CONTENT-TYPE are all equivalent. Node exposes the headers on requests and responses it has parsed in lowercase format.

5.1.3. Mocking and pattern-matching

As discussed in Section 2.6.2, “Pattern-matching arguments”, we can pass parameters to the `given()` mock modifier that, instead of being exactly equal to the expected input, act as pattern matchers that check the input fulfills some condition. In our example, we want to say that `response.writeHead()` should be called with headers that include a match for `Content-Type`, with a value beginning with `"application/json"` — we can use the `match()` matcher built into `jstest` to represent the required header value. And, we want `response.end()` to be called with a string that encodes the query parameters as some valid JSON string. Let’s rewrite our test to try and express these requirements using matchers in place of exact values; don’t worry about the matcher implementations yet, for now just focus on expressing the test’s requirements as succinctly as possible, avoiding the coupling problems we’ve identified.

Figure 5.7. `node/basic_server/server_matcher_spec.js`

```
var JS      = require("jstest"),
    server  = require("./server"),
    headers = require("./header_matcher"),
    json    = require("./json_matcher")

JS.Test.describe("server (matchers)", function() { with(this) {
  it("returns the query parameters as JSON", function() { with(this) {
    var request  = {url: "/?hello=world&something=else"},
        response = {},
        header   = headers({"Content-Type": match(/^application\/json(;|$)/)})

    expect(response, "writeHead").given(200, header)
    expect(response, "end").given(json({hello: "world", something: "else"}))

    server.emit("request", request, response)
  }})
}})
```

A matcher is an object with an `equals()` method that takes a value and returns `true` or `false` depending on whether the value fulfills the requirements. We’ve introduced two matchers here: `headers()` and `json()`. The first, `headers()`, takes a set of key-value pairs representing the desired headers, and should return a matcher that accepts any object that contains case-insensitive key matches for all the desired header names, with the correct values. We can do this by making the matcher normalise both the expected and actual headers so all their keys are lowercase, and then iterating over the expected headers to make sure each one appears in the actual headers.

This matcher uses `JS.Enumerable.areEqual()`, an internal method from `jstest`[7] that implements the notion of equality used by `assertEqual()`. In particular, it knows how to deal with objects with an `equals()` method, which is what our matchers are, and using it means that our matcher will correctly compose with other matchers, such as the `match(/^application\/json(;|$)/)` matcher we’ve used to specify a header value. If you’re writing custom matchers, you should usually use this rather than the === operator so that everything you’re using in your tests has the same idea of what equality means.

[7]Strictly speaking, `Enumerable.areEqual()` comes from a project called `jsclass`, which `jstest` is derived from. The method implements the notion of equality for the whole `jsclass` object system, its data structures, and so on. For more information, see http://jsclass.jcoglan.com/equality.html.

Figure 5.8. *node/basic_server/header_matcher.js*

```
var JS = require("jstest")

var normalise = function(headers) {
  var copy = {}
  for (var key in headers) {
    copy[key.toLowerCase()] = headers[key]
  }
  return copy
}

var headerMatcher = function(expected) {
  return {
    equals: function(actual) {
      expected = normalise(expected)
      actual   = normalise(actual)

      for (var key in expected) {
        if (!JS.Enumerable.areEqual(expected[key], actual[key])) {
          return false
        }
      }
      return true
    }
  }
}

module.exports = headerMatcher
```

The `json()` matcher takes an object, and should return a matcher that takes a string and checks that the string is a valid JSON representation of the object. To do this, we use `JSON.parse()` to parse the string, returning `false` if any errors are caught, and returning `true` only if `JS.Enumerable.areEqual()` says the expected data and the parsed JSON are equivalent.

Figure 5.9. *node/basic_server/json_matcher.js*

```
var JS = require("jstest")

var jsonMatcher = function(data) {
  return {
    equals: function(string) {
      try {
        var parsed = JSON.parse(string.toString())
        return JS.Enumerable.areEqual(data, parsed)
      } catch (e) {
        return false
      }
    }
  }
}

module.exports = jsonMatcher
```

If you’re writing matchers with significant amounts of logic in them, you might want to test the matchers themselves. This is usually a case of enumerating a set of inputs and specifying which ones the matcher matches and which it does not. For example, the `headers()` matcher matches headers that contain the given headers with the right values, it matches if the header names are a case-insensitive match, it matches if the headers contain additional fields, but it does not match if the required headers are missing or have the wrong values.

I’ve included a few tests for the matchers we’ve used below. Remember that `assertEqual()` understands values with `equals()` methods, so we can use that to test our matchers.

Figure 5.10. `node/basic_server/header_matcher_spec.js`

```
var JS      = require("jstest"),
    headers = require("./header_matcher")

JS.Test.describe("headers()", function() { with(this) {
  before(function() { with(this) {
    this.cacheControl = headers({"Cache-Control": "no-store"})
    this.contentType  = headers({"Content-Type":  match(/^application\/json(;|$)/)})
  }})

  it("matches if the headers contain an exact match", function() { with(this) {
    assertEqual( cacheControl, {"Cache-Control": "no-store"} )
  }})

  it("matches if the headers contain a case-insensitive match", function() { with(this) {
    assertEqual( cacheControl, {"cache-control": "no-store"} )
  }})

  it("matches if the headers contain additional fields", function() { with(this) {
    assertEqual( cacheControl, {"Cache-Control": "no-store", "Host": "localhost"} )
  }})

  it("does not match if the header is missing", function() { with(this) {
    assertNotEqual( cacheControl, {} )
  }})

  it("does not match if the header has the wrong value", function() { with(this) {
    assertNotEqual( cacheControl, {"Cache-Control": "no-cache"} )
  }})

  it("matches correctly if the value is a matcher", function() { with(this) {
    assertEqual(    contentType, {"Content-Type": "application/json"} )
    assertEqual(    contentType, {"Content-Type": "application/json; charset=utf-8"} )
    assertNotEqual( contentType, {"Content-Type": "text/html"} )
  }})
}})
```

Figure 5.11. `node/basic_server/json_matcher_spec.js`

```
var JS   = require("jstest"),
    json = require("./json_matcher")

JS.Test.describe("json()", function() { with(this) {
  before(function() { with(this) {
    this.matcher = json({numbers: [1, 2, 3]})
  }})

  it("matches a JSON representation of the object", function() { with(this) {
    assertEqual( matcher, '{"numbers":[1,2,3]}' )
  }})

  it("matches a valid JSON representation with whitespace", function() { with(this) {
    assertEqual( matcher, '{\n  "numbers": [1, 2, 3]\n}' )
  }})

  it("does not match invalid JSON", function() { with(this) {
    assertNotEqual( matcher, '{"numbers":[1,2,3}' )
  }})

  it("does not match JSON that does not represent the data", function() { with(this) {
    assertNotEqual( matcher, '{"numbers":[1,9,3]}' )
  }})
}})
```

5.1.4. Limitations of mock-based server testing

While it can make your tests run faster, mock-testing Node servers has a few pitfalls. As we've already seen, it can be tricky to write mock-based tests that are strict enough to represent the requirements of the client, but lenient enough to allow for non-breaking changes to the server's implementation. It's frustrating to have tests that break in response to changes that no real-world client would actually care about; it wastes time, reduces productivity and tends to make people feel like the tests aren't worth having. It's crucial to find the right balance that means you can be confident your code works, but you don't get false positives from your error-detection equipment.

One scenario where mocking breaks down very badly is when there are multiple ways of doing exactly the same thing, that is, a different set of method calls produces the same result from the end user's point of view. The Node HTTP API also allows us to write our example server like this:

Figure 5.12. `node/basic_server/alternate_server.js`

```
var http = require("http"),
    url  = require("url")

var server = http.createServer(function(request, response) {
  var query = url.parse(request.url, true).query
  response.statusCode = 200
  response.setHeader("Content-Type", "application/json")
  response.write(JSON.stringify(query, true, 2))
  response.end()
})

module.exports = server
```

We've replaced the `response.writeHead()` call with an assignment to `response.statusCode` and a call to `response.setHeader()`, and we've replaced `response.end(json)` with `response.write(json)` followed by `response.end()`[8]. This approach is just as amenable to mock-based testing as our previous version, with the slight modification that the assigned `statusCode` can't be checked using mocking since it's not a function call, and must be checked after the fact using an assertion:

Figure 5.13. `node/basic_server/alternate_server_mock_spec.js`

```
var JS     = require("jstest"),
    server = require("./alternate_server")

JS.Test.describe("alternate server (mocking)", function() { with(this) {
  it("returns the query parameters as JSON", function() { with(this) {
    var request  = {url: "/?hello=world&something=else"},
        response = {}

    expect(response, "setHeader").given("Content-Type", "application/json")
    expect(response, "write").given('{\n  "hello": "world",\n  "something": "else"\n}')
    expect(response, "end")

    server.emit("request", request, response)
    assertEqual( 200, response.statusCode )
  }})
}})
```

Again, we could use matchers to improve this test in various ways, but that's not the point. We've made some changes to the server that have no observable effect from the point of view of an external client, and yet our tests in Figure 5.7, "`node/basic_server/server_matcher_spec.js`" would break given

[8]In the Node stream API, `stream.end(data)` is syntactic sugar for `stream.write(data)` followed by `stream.end()`. HTTP requests and responses are streams, where writing to the stream appends data to the request/response body.

the new code. We have had to change our tests to cope with the different APIs being used, even though the end result is the same.

This sort of problem becomes even harder when you're dealing with web frameworks that hide the inner HTTP workings from you. For example, we could write our little example app using Express[9]:

Figure 5.14. *node/basic_express/app.js*

```
var express = require("express"),
    http    = require("http")

var app = express()

app.get("/", function(request, response) {
  response.json(request.query)
})

module.exports = http.createServer(app)
```

Calling `express()` returns a function of the form `function(request, response) { … }`, which can be passed to `http.createServer()`; that is to say `app` is a Node request-handling function. But Express adds a set of methods to that function to help you describe how it should work; our example says that if `app()` is called with a `GET` request whose path is `/` then it should return the query parameters in JSON format in the response. Whether you export `app` itself or wrap it in an HTTP server is up to you; I put the `http.createServer(app)` in the module itself since everything that loads this module will probably want a server, not just the `app()` function.

We can test this Express app in exactly the same way as in Figure 5.4, "`node/basic_server/server_spec.js`", starting up a server and firing requests at it, and that works rather well. However, attempting to mock-test it as in Figure 5.6, "`node/basic_server/server_mock_spec.js`" proves much harder; since Express layers functionality and APIs on top of the Node HTTP system, the fake request and response objects we'd pass into `app()` are not the same objects that are yielded to the `app.get("/")` request handler. Node's HTTP APIs don't include `request.query` or `response.json()`, those are provided by Express.

When you invoke `app()` with a request and response, there's a lot of framework code sitting between that call and the request handlers you actually wrote. Predicting how that framework code is going to interact with the Node HTTP objects is hard. I tried to write a mock-based test for this app, starting with blank `request` and `response` objects, adding `request` properties and adding mock expectations to `response` until everything worked correctly. The result is shown in Figure 5.15, "`node/basic_express/app_mock_spec.js`", but before we get there I'll explain how I wrote the test.

What we really care about testing is that if we send a request like `GET /?hello=world&something=else` to the server, then it returns a `200 OK` response with `Content-Type: application/json` and the JSON-formatted query parameters in the body. So, in my test, I made a `request` object with `method` and `url` properties, and a blank `response` object, and passed them into the app. Immediately I got an error: `response` didn't have a `setHeader()` method. So, I set a mock expectation that `setHeader()` should be called with the `Content-Type`. Since mocking a method makes it throw an error if given unexpected arguments, I now had a lot of other errors from the other headers Express was trying to set, none of which I cared about. And, those errors made Express follow unexpected code paths, resulting in even more unexpected headers; in some cases it would go into an infinite loop. Then there were errors due to `request` and `response` not heaving a `headers` property, and the errors that were totally uninformative. Before I mocked out `response.end()` (i.e. when `response` still had no `end()` method), rather than getting errors about *that* missing method I instead got errors about `request.socket` having no

[9]http://expressjs.com/

`destroy()` method, and `response.output` and `response.outputEncodings` not responding to `push()`. Only, the errors were more like `Cannot call method 'push' of undefined`, so I had to go digging in the source code of Express, its dependencies and Node itself to figure out what was missing from my fake objects.

To cut a long story short, I spent nearly an hour writing the following test, which includes a lot of details we don't care about, particularly when it comes to headers. Note how Express initially sets `Content-Type` to `text/html`, before later changing it to `application/json`, although the test doesn't actually express this since mock expectations are order-independent. We have had to add lots of mock expectations for things we don't care about, simply to make the application stop throwing errors.

Figure 5.15. `node/basic_express/app_mock_spec.js`

```
var JS   = require("jstest"),
    app  = require("./app"),
    json = require("../basic_server/json_matcher")

JS.Test.describe("Express app (mocking)", function() { with(this) {
  it("returns the query parameters as JSON", function() { with(this) {
    var request = {
      method:  "GET",
      url:     "/?hello=world&something=else",
      headers: {}
    }

    var response = {_headers: {}}

    expect(response, "setHeader").given("Content-Type",   "text/html; charset=utf-8")
    expect(response, "setHeader").given("Content-Type",   "application/json")
    expect(response, "setHeader").given("Content-Length", "36")
    expect(response, "setHeader").given("ETag",           'W/"24-L+oct+pdJBs3gXph1NRCaA"')
    expect(response, "setHeader").given("X-Powered-By",   "Express")

    expect(response, "end").given(json({hello: "world", something: "else"}), undefined)

    app.emit("request", request, response)

    assertEqual( 200, response.statusCode )
  }})
}})
```

Here we've reused the pattern-matcher for the JSON string as discussed in Section 5.1.3, "Mocking and pattern-matching" because Express calls `end()` with a `Buffer`. But, the test could be improved by simplifying the mocks; we could instead stub out `response.setHeader()` without specifying any arguments, before setting an expectation for the one value we care about:

Figure 5.16. Accepting any arguments with a stub, while checking for one particular input

```
stub(response, "setHeader")
expect(response, "setHeader").given("Content-Type", "application/json")
```

This would improve the quality of the test, but the time and effort it took to produce the test in the first place indicates we shouldn't be using mocking here. Mocking is supposed to be used to specify a clear contract between two components; if you're digging around in framework internals trying to find out what to stub, you aren't checking a well-defined boundary, you're testing implementation details. This test tells us more about the inner workings of Express than about what an HTTP client can expect from our server, and so it isn't very useful.

In general this tends to come up whenever you try to mock APIs that you didn't design and don't control. Mocking works best with APIs that you understand very well, usually because you designed, built and tested them yourself, but that's not to say you should never use it with third-party APIs. It's a balancing act, and you need to experiment to find out what works for your team.

Personally, due to the fast start-up times and request turn-around you get with Node, I usually write these kinds of tests as black boxes, from the point of view of an external HTTP client. I try to minimise the number of these tests by putting very little logic in my HTTP handlers themselves; instead the handlers will usually delegate to other internal APIs and I focus the bulk of my testing on those internal modules. Since the server is just another JavaScript object, you are still free to stub out the APIs it talks to internally, and this will prove very useful as we get onto more complex examples.

5.2. Dealing with databases

The servers we've looked at so far are extremely simple, toy examples. Most apps, in order to do anything useful, need to store data so that users can save and retrieve their work, communicate with one another, or receive personalised alerts from the application. In this section we'll look at an application built on top of Redis[10], and examine some approaches to structuring and testing it. The initial design will be quite poor, but by analysing its structure we'll be able to refactor it into a better shape.

5.2.1. Designing an HTTP API

To begin with, let's lay out the requirements for what we're about to build. We'd like an API to support a chatroom application. This application will have a very small surface area but will provide us with plenty of code to discuss. When using the app, a user will need to be able to pick a username, see all the messages for each chatroom, including polling for new messages since the last one they saw, and post new messages to a chatroom. The API will, at first, support only these three kinds of requests:

- `POST /users` with a `username` parameter, that will allow people to register to use the service. The chat app will initially not have any authentication; you can pick a username and start chatting, like on IRC.
- `POST /chat/:roomName` with `userId` and `message` parameters, that will allow users to post a text message to the named room. There is no access control on rooms, and users may post to whichever room they like without explicitly creating or registering for it first.
- `GET /chat/:roomName` with a `since` parameter, that will return all the messages posted to the named room since the given timestamp.

Let's see a few examples of this API in action. The registration endpoint takes a `username` and returns a new user record in JSON format, telling the user what their ID is. Each registered user is issued with a sequential ID. Here's what a successful request looks like:

Figure 5.17. A successful registration request

```
$ curl -iX POST http://localhost:1337/users -d "username=alice"
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Content-Length: 36

{
  "username": "alice",
  "id": 1
}
```

If the `username` is invalid, then the server returns a `409 Conflict` response with a list of errors:

[10]http://redis.io/

Figure 5.18. An invalid registation request with no username

```
$ curl -iX POST http://localhost:1337/users -d "username="
HTTP/1.1 409 Conflict
Content-Type: application/json; charset=utf-8
Content-Length: 87

{
  "errors": [
    "Usernames may only contain letters, numbers and underscores"
  ]
}
```

To post a message to a chatroom, we send a `POST` to the named room, passing in our `userId` and the `message` we want to post. We get back a JSON representation of the message, including its ID and timestamp. Just like users, messages have sequential IDs. If any of the input is invalid, a `409 Conflict` response is returned as above.

Figure 5.19. Posting messages to a chatroom

```
$ curl -iX POST http://localhost:1337/chat/lounge \
       -d userId=1 -d message=First

HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Content-Length: 65

{
  "id": 1,
  "timestamp": 1392755172018,
  "message": "First"
}

$ curl -iX POST http://localhost:1337/chat/lounge \
       -d userId=1 -d message=Second

HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Content-Length: 66

{
  "id": 2,
  "timestamp": 1392755182531,
  "message": "Second"
}
```

To retrieve the messages for a chatroom, we send a `GET` to the name of the room we want, passing in a parameter called `since`, which is the timestamp of the last message we saw (we pick a timestamp from years ago if we want to see all the messages). For example, this gets all the messages we posted in the above example:

Figure 5.20. Requesting all the messages in a chatroom going back a few years

```
$ curl -iX GET http://localhost:1337/chat/lounge?since=1000000000000
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 251

{
  "messages": [
    {
      "id": 1,
      "timestamp": 1392755172018,
      "message": "First",
      "username": "alice"
    },
    {
      "id": 2,
      "timestamp": 1392755182531,
      "message": "Second",
      "username": "alice"
    }
  ]
}
```

And this only gets the final message, because `since` is later than the timestamp of the first message.

Figure 5.21. Requesting only the recent messages from a chatroom

```
$ curl -iX GET http://localhost:1337/chat/lounge?since=1392755172019
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 137

{
  "messages": [
    {
      "id": 2,
      "timestamp": 1392755182531,
      "message": "Second",
      "username": "alice"
    }
  ]
}
```

Now we know the API we need, we have to figure out how to model this in Redis[11]. Redis is commonly referred to as a ‘NoSQL database’ and a ‘key-value store’, but really it’s more accurate to call it a ‘data-structure server’. It is a key-value store in the sense that you store pieces of data indexed by a string key, but those data can be of various types. They can be strings, or lists, or hashes (themselves a key-value map structure), sets or sorted sets. Using Redis effectively comes down to picking the right data structures and operations to model your data and make changes to it atomically.

5.2.2. Creating a database schema

Although many people call NoSQL databases ‘schemaless’, they absolutely do have a schema, it’s just that the database software doesn’t enforce one for you. When you change the way your data store represents things, you need to make sure all the server processes you have running at that time can cope with the change gracefully, without crashing or introducing consistency errors. Although the lack of schema enforcement certainly helps while prototyping, it pays to put some thought into it rather than rushing in and committing to a design that can’t change as needed later.

[11]I’m not claiming the following design is the best possible, or that Redis is the best choice for this application, but this app is a reasonable prototype and gives us plenty of design questions to consider.

For our chat application, we’re going to use the following design:

- For each kind of record that needs sequential IDs, we’ll have a counter. There will be two keys, `counters:user` and `counters:message` and our app will use the Redis `INCR`[12] command to atomically produce IDs for new records.
- Each user will be stored in a hash containing all the user’s information. In the above example, the key `users:1` contains a hash with `id=1` and `username=alice`.
- So that we can look up users by the username, we will index each user by mapping their username to their ID. The value of the key `index:users:alice` is `1`.
- So that we don’t create duplicate usernames, we keep a set of all registered usernames in the key `users`. This lets us atomically check if a username is taken and claim it if not, using the `SADD`[13] command.
- Every message will be stored under a key like `messages:1` as a hash with `id`, `timestamp`, `userId` and `message` fields. When producing API responses, the `userId` of a message will be used to look up the `username` and include it in the message.
- A room will be represented as a sorted set of message IDs, under a key like `rooms:lounge:messages`, where the IDs are sorted by the message’s timestamp. This will let us use the `ZRANGEBYSCORE`[14] command to efficiently grab all the messages since a given time.

5.2.3. Building the application

Having decided on an initial feature set, let’s start building the app. The first pass at a server for this application is shown in Figure 5.23, “`node/chat_api/app.js`”. There are a number of design differences from the servers we’ve seen before, the first of which is that rather than include all the logic for each request handler inline, I’ve separated each request handler function out into its own file. This is simply because there is going to be quite a lot of logic in each request handler, and it’s useful to be able to see the types of requests the server responds to in one short file, without all the processing logic getting in the way.

The second major difference is that the value of this module is not an HTTP server, i.e. `http.createServer(app)` as we had before. Instead, it’s a function that *creates* such a server; the function beginning `module.exports = function(config) { …` is what we export from the module. This is because the app is no longer self-contained, but needs to connect to external entities, in this case a Redis database. It needs to know the host, port and other connection settings in order to make this connection, but we don’t want to hard-code those settings; we should use a different database for testing from the one we use for production data. It’s common to use environment variables for configuration data, for example we could set up the database connection like this:

Figure 5.22. Using environment variables directly

```
var db = redis.createClient(process.env.REDIS_PORT, process.env.REDIS_HOST)
db.select(process.env.REDIS_DB)
```

However, it’s bad practice to put code that reads environment variables inside your application and library code, for several reasons. First, it means that to test it you will have to change global variables; `process.env` is a global variable and changing its properties could affect parts of the system aside from the one you’re trying to test, so it’s easier if your components accept their configuration as function arguments that don’t have global side-effects. Second, peppering references to environment variables all over the codebase makes it hard to see which variables you should use to configure the app, and

[12]http://redis.io/commands/incr
[13]http://redis.io/commands/sadd
[14]http://redis.io/commands/zrangebyscore

also makes it hard to change the config mechanism; suppose you decide config should now live in files instead of environment variables, you now have to change lots of lines of code all over your system. It's best if the application code is mostly agnostic to how its configuration is managed; if you want to use environment variables then you can write a start-up script that reads the config from the environment and passes the config values into the app. Finally, if libraries and application modules hard-code environment variables they want to use, then since `process.env` is a global variable there's a chance two modules will pick the same name for a config value and a single environment variable will end up affecting both modules. Therefore, it's best if modules take configuration as function arguments, and configuration management is left up to the end user who's glueing everything together.

Here's the application module, whose main job is to wire everything up. It creates a database connection, sets up a router to handle the API endpoints, and delegates all the processing to other modules.

Figure 5.23. `node/chat_api/app.js`

```
var connect = require("connect"),
    express = require("express"),
    http    = require("http"),
    redis   = require("redis")

var register    = require("./routes/register"),
    postMessage = require("./routes/post_message"),
    getMessages = require("./routes/get_messages")

module.exports = function(config) {
  var db = redis.createClient(config.redis.port, config.redis.host, config.redis)
  db.select(config.redis.database || 0)

  var app = express()
  app.use(connect.urlencoded())

  app.post("/users", function(request, response) {
    register(request, response, db)
  })

  app.post("/chat/:roomName", function(request, response) {
    postMessage(request, response, db)
  })

  app.get("/chat/:roomName", function(request, response) {
    getMessages(request, response, db)
  })

  return http.createServer(app)
}
```

We can start the app up in production using a simple boot script, which reads the configuration from the environment and creates a copy of the app with that configuration:

Figure 5.24. `node/chat_api/main.js`

```
var app = require("./app"),
    env = process.env

var server = app({
  redis: {
    host:     env.REDIS_HOST,
    port:     env.REDIS_PORT,
    database: env.REDIS_DB
  }
})

server.listen(env.PORT)
```

Running this script like so will connect the app to the database and allow it to start receiving requests:

Figure 5.25. Starting an app in a configuration environment

```
$ REDIS_HOST=redis.example.com REDIS_DB=1 PORT=1337 node main.js
```

Before we get to the request-handling code itself, I should mention that it makes heavy use of the Async[15] module, particularly `async.waterfall()`. Most I/O operations in Node, which includes all database commands, is asynchronous, and operations must be sequenced using callbacks. By convention, the first parameter to a Node callback is for errors, so the first line of a callback function is often something like `if (error) return handleError(error)`, i.e. deal with the error and don't run the following commands. This results in code that looks like this, with much nesting and repetitive error handling.

Figure 5.26. Sequencing Redis operations in a 'callback pyramid'

```
var handleError = function(error) {
  response.json(error.status || 500, {errors: [error.message]})
}

redis.incr("counters:message", function(error, id) {
  if (error) return handleError(error)

  var data = {id: id, timestamp: timestamp, userId: userId, message: message}
  redis.hmset("messages:" + id, data, function(error, ok) {
    if (error) return handleError(error)

    redis.zadd("rooms:" + roomName, timestamp, data.id, function(error, added) {
      if (error) return handleError(error)

      response.end(added)
    })
  })
})
```

Extreme cases of this sort of code, especially once conditionals and loops get involved, are affectionately known as 'callback hell'. `async.waterfall()` attempts to improve things by making the sequencing and error handling more implicit. It takes an array of functions, and runs them one at a time, feeding the output of one command into the input of the next via callbacks. It uses the 'error-first' callback convention to short-circuit the list; if any command in the sequence produces an error, then the rest of the functions in the array are not invoked and an error-handling callback is executed, making it more like using `try/catch` in synchronous code. With `async.waterfall()`, we can rewrite the above code as follows, where I have used `cb` as a short-hand for `callback` to reduce line noise:

[15]https://www.npmjs.org/package/async

Figure 5.27. Sequencing Redis operations using Async

```
var data

async.waterfall([
  function(cb) {
    redis.incr("counters:message", cb)
  }, function(id, cb) {
    data = {id: id, timestamp: timestamp, userId: userId, message: message}
    redis.hmset("messages:" + id, data, cb)
  }, function(ok, cb) {
    redis.zadd("rooms:" + roomName, timestamp, data.id, cb)
  }
], function(error, added) {
  if (error) {
    response.json(error.status || 500, {errors: [error.message]})
  } else {
    response.end(added)
  }
})
```

5.2.4. Monolithic request handlers

In our initial prototype, each of the request handlers used by Figure 5.23, “`node/chat_api/app.js`” is a ‘monolithic’ function. They each take a `request` and `response` object and a `redis` database client, and contain all the logic needed to process the request. Let’s look at the request handler for the `POST /users` request first; its source code is shown in Figure 5.28, “`node/chat_api/routes/register.js`”.

This function follows a pattern that we will see reproduced in the other request handlers. First, it extracts the pieces of data it needs from the request. In this case, all it needs is the value of the `username` parameter. It must then validate the inputs to make sure they conform to our requirements; we will accept usernames that contain only letters, numbers and underscores[16]. If validation fails, we must return a `409 Conflict` response without talking to the database.

If the data is valid, we can begin talking to Redis. Say that `username` is `"bob"`. We first run `SADD users bob`, which will return `1` if `bob` was added to the set and `0` if it’s already present. If `bob` was added, then we run `INCR counters:user` to get a new user ID, and add the ID to the user record we’re creating. Say the new ID is `33`, we store the record using `HMSET users:33`, then store the username-to-ID mapping with `SET index:users:bob 33`. If all these steps succeed we return a `201 Created` response.

If `bob` was already in the `users` set, then `GET index:users:bob` returns his user ID, and `HGETALL users:33` returns all his data, which we then return in a `200 OK` response. If there was an error running any of these steps, a `500 Internal Server Error` response is sent to the client.

The full code for this procedure is shown below.

[16]While this is a somewhat arbitrary restriction, it is important that usernames are not allowed to include the `:` character. We use it as a key delimiter in Redis and allowing it to appear in data can create opportunities for key manipulation attacks. For example, if we stored user data under `users:bob` and user messages under `users:bob:messages`, then allowing someone to pick the username `bob:messages` would clobber `bob`'s message list.

Figure 5.28. `node/chat_api/routes/register.js`

```
var async = require("async")

module.exports = function(request, response, redis) {
  var username = request.body.username || ""

  if (!username.match(/^[a-z0-9_]+$/i)) {
    return response.json(409, {
      errors: ["Usernames may only contain letters, numbers and underscores"]
    })
  }

  var userData = {username: username}

  async.waterfall([
    function(cb) {
      redis.sadd("users", username, cb)
    }, function(added, cb) {
      if (added === 1) {
        async.waterfall([
          function(cb) {
            redis.incr("counters:user", cb)
          }, function(id, cb) {
            userData.id = id
            redis.hmset("users:" + id, userData, cb)
          }, function(ok, cb) {
            redis.set("index:users:" + username, userData.id, cb)
          }, function(ok, cb) {
            cb(null, 201, userData)
          }
        ], cb)
      } else {
        async.waterfall([
          function(cb) {
            redis.get("index:users:" + username, cb)
          }, function(id, cb) {
            redis.hgetall("users:" + id, cb)
          }, function(userData, cb) {
            userData.id = parseInt(userData.id, 10)
            cb(null, 200, userData)
          }
        ], cb)
      }
    }
  ], function(error, status, data) {
    if (error) {
      response.json(500, {errors: [error.message]})
    } else {
      response.json(status, data)
    }
  })
}
```

To test this code, we could invoke the `register` module directly with fake `request`, `response` and `redis` objects, and carry out mock-based testing as described in Section 5.1.2, “Mock-based server testing”. However, as we’ve already seen, this becomes very ugly when we’re dealing with complex interactions. Mocking/stubbing the sequence of Redis actions and their return values is likely to be a time-consuming and error-prone process, plus this one function deals with all sorts of things: HTTP details, data validation, and database interaction. A mock-based test for this will be verbose, complex and hard to read, as you can see in the following example. Note that even though the stubbing makes all the Redis calls synchronous, `async.waterfall()` runs its function list asynchronously, so we need the `setTimeout()` call to allow time for all the commands to execute.

Figure 5.29. `node/chat_api/spec/register_mock_spec.js`

```
var JS       = require("jstest"),
    register = require("../routes/register")

JS.Test.describe("register()", function() { with(this) {
  before(function() { with(this) {
    this.request  = {body: {username: "alice"}}
    this.response = {}
    this.redis    = {}
    this.userData = {id: 33, username: "alice"}
  }})

  after(function(resume) { with(this) {
    register(request, response, redis)
    setTimeout(resume, 10)
  }})

  describe("with an invalid username", function() { with(this) {
    before(function() { with(this) {
      request.body.username = ""
    }})

    it("returns a 409 Conflict response", function() { with(this) {
      expect(response, "json").given(409, {
        errors: ["Usernames may only contain letters, numbers and underscores"]
      })
    }})
  }})

  describe("when the username does not already exist", function() { with(this) {
    before(function() { with(this) {
      stub(redis, "sadd").given("users", "alice").yields([null, 1])
    }})

    it("adds the user to the database", function() { with(this) {
      expect(redis, "incr").given("counters:user").yields([null, 33])
      expect(redis, "hmset").given("users:33", userData).yields([null, "OK"])
      expect(redis, "set").given("index:users:alice", 33).yields([null, "OK"])

      expect(response, "json").given(201, userData)
    }})
  }})

  describe("when the username exists", function() { with(this) {
    before(function() { with(this) {
      stub(redis, "sadd").given("users", "alice").yields([null, 0])
    }})

    it("returns the existing user", function() { with(this) {
      expect(redis, "get").given("index:users:alice").yields([null, "33"])
      expect(redis, "hgetall").given("users:33").yields([null, userData])

      expect(response, "json").given(200, userData)
    }})
  }})
}})
```

These tests are verbose, and are really just verbatim copies of the commands in the implementation code; they say what the code does but they don't prove the code actually fulfills its requirements correctly.

So, we will instead take the approach of Section 5.1.1, “Black-box server testing”; this will proceed pretty much as we saw before, starting a server, firing requests at it and checking the responses, but now we also need to worry about the database. We need to create a copy of the app with database config pointing to our test database, one we don’t mind getting wiped. At the end of each test, we must make sure we clean up any data we left in the database since it could affect the outcome of later tests; we can accomplish this using the `FLUSHDB` command. Although these are full-stack tests that go right from the HTTP frontend down to the database, they run reasonably quickly and give us confidence that everything is wired up correctly, without us needing to worry about whether we’ve stubbed the Express or Redis APIs correctly.

Figure 5.30. `node/chat_api/spec/register_spec.js`

```
var JS    = require("jstest"),
    app   = require("../app"),
    steps = require("./server_steps")

var redisConfig = {host: "127.0.0.1", port: 6379, database: 15},
    server      = app({redis: redisConfig})

JS.Test.describe("POST /users", function() { with(this) {
  include(steps)

  before(function() { with(this) {
    startServer(server)
  }})

  after(function() { with(this) {
    stopServer()
    cleanDatabase(redisConfig)
  }})

  it("rejects missing usernames", function() { with(this) {
    post("/users", {})
    checkStatus(409)
    checkJSON({errors: ["Usernames may only contain letters, numbers and underscores"]})
  }})

  it("rejects invalid usernames", function() { with(this) {
    post("/users", {username: "$%^&"})
    checkStatus(409)
    checkJSON({errors: ["Usernames may only contain letters, numbers and underscores"]})
  }})

  it("accepts new usernames and returns a new ID", function() { with(this) {
    post("/users", {username: "alice"})
    checkStatus(201)
    checkJSON({id: 1, username: "alice"})

    post("/users", {username: "bob"})
    checkStatus(201)
    checkJSON({id: 2, username: "bob"})
  }})

  it("returns the existing ID for registered usernames", function() { with(this) {
    post("/users", {username: "alice"})
    post("/users", {username: "alice"})
    checkStatus(200)
    checkJSON({id: 1, username: "alice"})
  }})
}})
```

The one upside of mock-based testing in this scenario is that it allows us to simulate what happens when there's an error while talking to Redis, since we can use the `yields()` modifier to make our fake Redis client emit an error.

By Node convention, errors are always communicated via the first parameter to a callback. In Figure 5.29, "`node/chat_api/spec/register_mock_spec.js`", all the `yields()` modifiers set the first callback argument to `null`, meaning there was no error, but if we instead pass an `Error` object as the first callback argument to the `SADD` command, then no further Redis commands should be run and the server should return a `500 Internal Server Error`. We can represent the fact that no more Redis commands should be run by simply omitting the mock expectations for them; since `redis` is a plain old object rather then a real client, we'll get errors if the `register()` function tries to call any methods we've not set expectations for.

Figure 5.31. `node/chat_api/spec/redis_error_spec.js`

```
var JS       = require("jstest"),
    register = require("../routes/register")

JS.Test.describe("register() errors", function() { with(this) {
  before(function() { with(this) {
    this.request  = {body: {username: "alice"}}
    this.response = {}
    this.redis    = {}

    stub(redis, "sadd").yields([new Error("E_ACCESSDENIED")])
  }})

  it("returns a 500 Internal Server Error response", function(resume) { with(this) {
    expect(response, "json").given(500, {errors: ["E_ACCESSDENIED"]})

    register(request, response, redis)
    setTimeout(resume, 10)
  }})
}})
```

5.2.5. Working with time

Now we come to the second request handler in our chatroom app, the `POST /chat/:roomName` handler. This will follow a similar pattern to the `register` module; it will extract some values from the request, validate them, run some commands against Redis, and send back a response containing the message's generated ID, timestamp and other data. Once validation is complete, it will check the given `userId` actually exists in the database, emitting a `409` error if not. Then, it will use `INCR counters:message` to generate a new message ID, then use `ZADD rooms:name:messages timestamp id` to add the message's ID to the set of messages for the given room. Sorting the messages by their timestamp allows Redis to efficiently tell us the messages in a room since a given time.

There is one new kind of interaction in this handler: it needs to know the current time so it can attach that timestamp to the message. To do this, we've created a module called `time` with a `time.current()` method, that returns the current Unix epoch time in milliseconds. This is the entire module:

Figure 5.32. `node/chat_api/lib/time.js`

```
var Time = {
  current: function() {
    return Date.now()
  }
}

module.exports = Time
```

It probably seems silly to wrap `Date.now()` in another function, but it’s a necessary step to making the code testable. We want to write tests that check that the app grabs the current time and attaches it to the message, but we can’t put a fixed value for the timestamps in our tests since time is always marching forward. We can’t even say `now = Date.now()` in our tests and check that the message’s timestamp is set to `now`, because time ellapses between us setting that variable and the handler being executed.

We can ‘freeze’ time by stubbing it out. However, stubbing out the `Date.now()` method messes with Node’s internal scheduling, and will stop the code and tests from running, so we can’t do that. Instead, we put a layer of indirection between our app and `Date.now()`, in the form of the `time.current()` method. We can stub this out, and our app will see the frozen test time, but code that depends on `Date.now()` will continue unaffected.

Figure 5.33. *`node/chat_api/routes/post_message.js`*

```
var async = require("async"),
    time  = require("../lib/time")

module.exports = function(request, response, redis) {
  var roomName  = request.params.roomName || "",
      userId    = request.body.userId     || "",
      message   = request.body.message    || "",
      timestamp = time.current(),
      errors    = [],
      messageData

  if (!roomName.match(/^[a-z0-9_]+$/i)) {
    errors.push("Rooms may only contain letters, numbers and underscores")
  }
  if (message.match(/^ *$/)) {
    errors.push("Message must not be blank")
  }
  if (errors.length > 0) return response.json(409, {errors: errors})

  async.waterfall([
    function(cb) {
      redis.exists("users:" + userId, cb)
    }, function(exists, cb) {
      var error = (exists === 0) ? "User #" + userId + " does not exist" : null
      cb(error && {message: error, status: 409})
    }, function(cb) {
      redis.incr("counters:message", cb)
    }, function(id, cb) {
      messageData = {id: id, timestamp: timestamp, userId: userId, message: message}
      redis.hmset("messages:" + id, messageData, cb)
    }, function(ok, cb) {
      redis.zadd("rooms:" + roomName + ":messages", timestamp, messageData.id, cb)
    }
  ], function(error) {
    if (error) {
      response.json(error.status || 500, {errors: [error.message]})
    } else {
      delete messageData.userId
      response.json(201, messageData)
    }
  })
}
```

There is an alternate construction for this request handler that would make it easier to test without stubbing out time: rather than the handler reaching out into the outside world to *ask* for the current time, the outside world *tells* the handler what the current time is by passing it in as an argument. With this design, a mock-based test *could* create a time value using `Date.now()` (or `new Date()` if you need an object rather than a numeric timestamp), pass it into the handler, and check that the same value comes out in the message response. In production, the Express app would grab the current time, also using `Date.now()`, and pass it into the handler function. In other words, the handler's initial data-gathering phase would look like this; notice the absence of the `time` module here:

Figure 5.34. Capturing the current time at the edge of the system

```
app.post("/chat/:roomName", function(request, response) {
  var timestamp = Date.now()
  postMessage(request, response, db, timestamp)
})
```

Figure 5.35. Taking the current time as a parameter

```
var async = require("async")

module.exports = function(request, response, redis, timestamp) {
  var roomName = request.params.roomName || "",
      userId   = request.body.userId     || "",
      message  = request.body.message    || "",
      // ...
}
```

With this design, the request handler doesn't really care what `timestamp` *means*, it's just a number that gets handed off to Redis and put into a JSON response. Capturing the current time once and passing it down into the lower layers can also solve the problem of multiple components needing to know the current time, and needing to agree on what the current time is. For example, a payroll processor might need to find all the contractors that did work this week so we can pay them; but if the value of 'this week' ticks over while the request is being processed, some people might not get paid! Rather than all the components asking for the time with their own `Date.now()` calls and getting slightly different values[17], the time is captured once and given to all the components that need it. This makes it easier to create temporally consistent logic, where you can pretend that all your logic executes at the same instant.

Although mock-based testing has some advantages, our experience with the previous request handler showed it was not preferable to black-box testing due to all the other complexity in the module. Although it makes the question of time easier to deal with, it's a bad fit for almost all the other concerns we need to test, so on balance we're going to stick with black-box testing for this handler. This means invoking the code via an HTTP request rather than as a function call, and it doesn't make much sense for the client to pass in the current time: the server should use its own clock. So, we're forced to use stubbing in this situation.

This test suite is very similar to the last one, making requests and checking the responses. Note how we stub `time.current()` to return a fixed value, then check for that value in the response to confirm the app works as expected. Also notice that we're checking messages were posted by using the `GET /chat/:roomName` endpoint, rather than inspecting the state of Redis. This suite is all about the user-facing API; we want to know that if the user posts a message, it shows up when they query for it, and we're not concerned with the implementation details of how that happens.

[17]V8 might be fast, but it's not instantaneous! A typical web request takes at least a few milliseconds so the value of `Date.now()` will change while a request is being processed.

Figure 5.36. *node/chat_api/spec/post_message_spec.js*

```
var JS   = require("jstest"),
    app  = require("../app"),
    time = require("../lib/time")

var redisConfig = {host: "127.0.0.1", port: 6379, database: 15},
    server      = app({redis: redisConfig})

JS.Test.describe("POST /chat/:roomName", function() { with(this) {
  include(require("./server_steps"))

  before(function() { with(this) {
    startServer(server)
    post("/users", {username: "alice"})
    this.payload = {userId: 1, message: "Hi there"}
  }})

  after(function() { with(this) {
    stopServer()
    cleanDatabase(redisConfig)
  }})

  it("rejects invalid room names", function() { with(this) {
    post("/chat/not-ok", payload)
    checkStatus(409)
    checkJSON({errors: ["Rooms may only contain letters, numbers and underscores"]})
  }})

  it("rejects non-existent user IDs", function() { with(this) {
    payload.userId = 2
    post("/chat/lounge", payload)
    checkStatus(409)
    checkJSON({errors: ["User #2 does not exist"]})
  }})

  it("accepts valid messages", function() { with(this) {
    stub(time, "current").returns(1392747231870)
    post("/chat/lounge", payload)
    checkStatus(201)
    checkJSON({id: 1, timestamp: 1392747231870, message: "Hi there"})
  }})

  it("posts acceptable messages", function() { with(this) {
    stub(time, "current").returns(1392747231870)
    post("/chat/lounge", payload)
    get("/chat/lounge", {since: 1000000000000})
    checkJSON({
      messages: [{
        id:        1,
        timestamp: 1392747231870,
        username:  "alice",
        message:   "Hi there"
      }]
    })
  }})

  it("does not post invalid messages", function() { with(this) {
    payload.message = "    "
    post("/chat/lounge", payload)
    get("/chat/lounge", {since: 1000000000000})
    checkJSON({messages: []})
  }})
}})
```

Finally, we come to the `GET /chat/:roomName` endpoint, whose job is to return a room's messages since a certain time, with their timestamps, usernames and content. We'll begin again by using `time.current()` to capture the time, and validating the inputs as before. We'll use `ZRANGEBYSCORE rooms:name:messages startTime endTime` to get the IDs of the messages since the given time, then for each ID (that's the `async.map()` bit) we'll call `HGETALL messages:id` to get the message data, followed by `HGET users:id username` to find the message's username. The test suite and further discussion follow immediately afterward.

Figure 5.37. *`node/chat_api/routes/get_messages.js`*

```
var async = require("async"),
    time  = require("../lib/time")

module.exports = function(request, response, redis) {
  var roomName = request.params.roomName,
      since    = request.query.since || "",
      now      = time.current(),
      errors   = []

  if (!roomName.match(/^[a-z0-9_]+$/i)) {
    errors.push("Rooms may only contain letters, numbers and underscores")
  }
  if (!since.match(/^[1-9][0-9]{12}$/)) {
    errors.push("The 'since' parameter must be a valid timestamp")
  }
  if (errors.length > 0) return response.json(409, {errors: errors})

  since = parseInt(since, 10)

  async.waterfall([
    function(cb) {
      redis.zrangebyscore("rooms:" + roomName + ":messages", since, now, cb)
    }, function(messageIds, cb) {
      async.map(messageIds, function(id, cb) {
        var messageData
        async.waterfall([
          function(cb) {
            redis.hgetall("messages:" + id, cb)
          }, function(data, cb) {
            messageData = data
            messageData.id = parseInt(messageData.id, 10)
            messageData.timestamp = parseInt(messageData.timestamp, 10)
            redis.hget("users:" + messageData.userId, "username", cb)
          }, function(username, cb) {
            messageData.username = username
            delete messageData.userId
            cb(null, messageData)
          }
        ], cb)
      }, cb)
    }
  ], function(error, messages) {
    if (error) {
      response.json(500, {error: error.message})
    } else {
      response.json(200, {messages: messages})
    }
  })
}
```

Figure 5.38. `node/chat_api/spec/get_messages_spec.js`

```
var JS   = require("jstest"),
    app  = require("../app"),
    time = require("../lib/time")

var redisConfig = {host: "127.0.0.1", port: 6379, database: 15},
    server      = app({redis: redisConfig})

JS.Test.describe("GET /chat/:roomName", function() { with(this) {
  include(require("./server_steps"))

  before(function() { with(this) {
    startServer(server)
    post("/users", {username: "alice"})

    this.oldTime = 1000000000000
  }})

  after(function() { with(this) {
    stopServer()
    cleanDatabase(redisConfig)
  }})

  it("rejects requests with an invalid timestamp", function() { with(this) {
    get("/chat/kitchen", {since: 1234})
    checkStatus(409)
    checkJSON({errors: ["The 'since' parameter must be a valid timestamp"]})
  }})

  describe("with messages in one room", function() { with(this) {
    before(function() { with(this) {
      stub(time, "current").returns(1300000001000, 1300000002000, 1399999999999)
      post("/chat/kitchen", {userId: 1, message: "First"})
      post("/chat/kitchen", {userId: 1, message: "Second"})
    }})

    it("returns all the messages given an old timestamp", function() { with(this) {
      get("/chat/kitchen", {since: oldTime})
      checkStatus(200)
      checkJSON({
        messages: [
          {id: 1, timestamp: 1300000001000, username: "alice", message: "First"},
          {id: 2, timestamp: 1300000002000, username: "alice", message: "Second"}
        ]
      })
    }})

    it("returns messages since the given timestamp", function() { with(this) {
      get("/chat/kitchen", {since: 1300000001500})
      checkStatus(200)
      checkJSON({
        messages: [
          {id: 2, timestamp: 1300000002000, username: "alice", message: "Second"}
        ]
      })
    }})

    it("returns no messages for other rooms", function() { with(this) {
      get("/chat/garage", {since: oldTime})
      checkJSON({messages: []})
    }})
  }})
}})
```

Again, this is test suite follows the pattern of starting the server, making some requests, checking the responses, stopping the server and cleaning the database. What makes this test interesting is how it deals with time-based queries. A good way to deal with this would be to add some messages to Redis directly, with fixed timestamps, and then make a request with another fixed time to see what we got back. However, since this is a black-box test that is only supposed to express what the outside world can see, we're not allowed to go poking around in the database without going via the HTTP API[18]. This means we can't pass in any timestamps ourselves, since they are all generated inside the application. So instead, we use a feature of the `jstest` stubbing library, which lets us set a sequence of return values for a method:

Figure 5.39. Setting a sequence of return values for a stubbed method

```
    stub(time, "current").returns(1300000001000, 1300000002000, 1399999999999)
```

This stubs `time.current()` to return one of three timestamps: an early one, one a little later, and one far in the future. Each call to `time.current()` will return the next timestamp in the sequence. We need to put some messages into the database so we can check we can get them out again, and to do that we need to use the `POST` endpoint. We insert two messages by making two `POST` requests, and those will use up the first two timestamps in the sequence. For the `GET` request, we need a value of the current time that's later than the timestamps on the messages, so that they are within the range of the `ZRANGEBYSCORE` query, so as long as the third timestamp is later than the first two, that's fine.

Stubbing `time.current()` in this way effectively lets us inject fixed times into the system so we can predict what we should get back. Using a time in the far past (before either of the messages was posted) should return all the messages, while using a time between the two messages' timestamps should only return the later one. It works, but it tends to be a bit brittle. Rather than passing in fixed values directly to functions where they are needed, we are making assumptions about how many times `time.current()` is called, and in what order various values should appear.

The most common failure mode of this approach is that either `time.current()` is stubbed incorrectly, or it's not stubbed at all. People write tests with timestamps a few years in the future on the assumption the tests won't survive that long, and then are inevitably surprised when the build for some old code breaks because one of their 'future' dates slipped into the past. This is a really frustrating kind of bug, since it often takes years between the bug being written and it being discovered, making it expensive to fix. If your code relies on the concepts of 'past' and 'future', try to make sure that all time values in the system are strictly controlled by the tests, and that the code isn't using the actual time anywhere during testing. Again, this is best achieved by having APIs that take time as an argument, rather than the code inside that API asking for the current time itself.

5.3. Extracting services

In the previous section, we built a working chatroom application but its design had some problems. Since each request handler function contains all the HTTP, data validation and persistence logic, none of those things can be tested on their own. If we want to test any part of this logic, we can only do so by making an HTTP request. And since the validation logic is in the same function as the database access code, we can't test the validation rules without hitting the database or providing a fake implementation of one.

For our chatroom, this isn't a big problem; the app is tiny and the tests are reasonably fast. But as applications grow, these things become a huge bottleneck: business rules and persistence logic tend to become more complicated, and need to be reused across multiple parts of a project, more things become coupled together, and the added complexity slows request processing down. If we want to test all the edge cases of a validation rule, for example, it is far more productive to be able to call that logic directly, without having to spin up a server and stub out all the other APIs that a request will invoke.

[18]Remember that this first design is deliberately problematic, we'll get to a better solution in Section 5.3, "Extracting services".

If all your tests cut through the full stack, then a change to any part of your system can break a lot of tests, often for reasons that are hard to discern. What you end up with is a set of tests that are hard to write and maintain due to the amount of setup involved, run very slowly and are poor indicators of the source of bugs. This can destroy a team's productivity if left unchecked; I've seen test suites that took hours to run even when parallelised on a cluster of EC2 machines, and that were impossible for developers to run locally before committing. This results in lots of broken builds that take days to fix, and can ultimately stop teams from shipping entirely. Long story short, you *really* don't want to get into this situation.

Fortunately, we can improve things by splitting the application into smaller components and testing them individually. By spotting patterns in the structure of the app, we can see boundaries between things that suggest how we might split things up. You'll notice that all our request handlers go through four distinct phases: they gather pieces of data from the HTTP request, they validate that data, they communicate with the database, and finally they emit an HTTP response. It makes sense for all the HTTP logic to remain in the Express app: its entire responsibility is to handle HTTP requests. But the validation and database logic can be extracted into *services*, objects that provide some focussed set of functionality that can be reused across the application.

Let's start by putting all the validation rules into a single module, separated from all the other logic.

Figure 5.40. `node/services/lib/validation.js`

```
module.exports = {
  VALID_NAME:     /^[a-z0-9_]+$/i,

  USERNAME_ERROR: "Usernames may only contain letters, numbers and underscores",
  ROOMNAME_ERROR: "Rooms may only contain letters, numbers and underscores",
  MESSAGE_ERROR:  "Message must not be blank",
  SINCE_ERROR:    "The 'since' parameter must be a valid timestamp",

  checkUser: function(userData) {
    var username = userData.username || "",
        errors   = []

    if (!username.match(this.VALID_NAME)) errors.push(this.USERNAME_ERROR)

    return (errors.length === 0) ? null : errors
  },

  checkMessage: function(messageData) {
    var roomName = messageData.roomName || "",
        message  = messageData.message  || "",
        errors   = []

    if (!roomName.match(this.VALID_NAME)) errors.push(this.ROOMNAME_ERROR)
    if (message.match(/^ *$/)) errors.push(this.MESSAGE_ERROR)

    return (errors.length === 0) ? null : errors
  },

  checkPoll: function(query) {
    var roomName = query.roomName || "",
        since    = query.since,
        errors   = []

    if (!roomName.match(this.VALID_NAME)) errors.push(this.ROOMNAME_ERROR)
    if (typeof since !== "number" || since < Math.pow(10,12)) errors.push(this.SINCE_ERROR)

    return (errors.length === 0) ? null : errors
  }
}
```

This module is very easy to test because it’s made up entirely of *pure functions*. A pure function is one whose output depends only on its arguments, not on any external state, and that does not have any side effects, i.e. it does not change the state of the outside world or of the arguments passed into it. In this case, our validation functions take an object containing some user data, a posted message, and so on, and return either an array of error messages or `null` if the data is valid. They don’t have any dependencies to stub out, and they don’t send information to a database or HTTP response. We just pass them some data and check what they return; testing them is as simple as saying `assertEqual( 4, 2 + 2 )`.

Here are a few examples. To test validation, I usually start with a valid object and check that it is indeed considered valid, and then I make small changes to it to check that they render it invalid. By only using small changes, we can be sure that the change in output is due to the single change we made; if we use completely different data in each test then it’s harder to tell the reason for the change in output.

Figure 5.41. `node/services/spec/validation_spec.js`

```
var JS         = require("jstest"),
    validation = require("../lib/validation")

JS.Test.describe("validation", function() { with(this) {
  describe("checkUser()", function() { with(this) {
    it("accepts valid usernames", function() { with(this) {
      assertNull( validation.checkUser({username: "carol"}) )
    }})

    it("rejects invalid usernames", function() { with(this) {
      assertEqual( ["Usernames may only contain letters, numbers and underscores"],
                   validation.checkUser({username: "**"}) )
    }})

    it("rejects missing usernames", function() { with(this) {
      assertEqual( ["Usernames may only contain letters, numbers and underscores"],
                   validation.checkUser({}) )
    }})
  }})

  describe("checkMessage()", function() { with(this) {
    before(function() { with(this) {
      this.message = {userId: 1, roomName: "lounge", message: "Hi"}
    }})

    it("accepts messages with a userId, room name and some text", function() { with(this) {
      assertNull( validation.checkMessage(message) )
    }})

    it("rejects messages with no room name", function() { with(this) {
      delete message.roomName
      assertEqual( ["Rooms may only contain letters, numbers and underscores"],
                   validation.checkMessage(message) )
    }})
  }})

  describe("checkPoll()", function() { with(this) {
    it("accepts a message query with a room name and timestamp", function() { with(this) {
      assertNull( validation.checkPoll({roomName: "kitchen", since: 1234567890123}) )
    }})

    it("rejects non-numeric 'since' values", function() { with(this) {
      assertEqual( ["The 'since' parameter must be a valid timestamp"],
                   validation.checkPoll({roomName: "kitchen", since: "whenever"}) )
    }})
  }})
}})
```

Because this module does not depend on the database, or interact with HTTP in any way, we gain several benefits. First, the module can be reused far more easily; because it isn't coupled to HTTP or a database, we can embed this logic in any system we like, saving us having to rewrite it for another application and keep the implementations in sync. And second, because this module does absolutely no I/O and has no asynchronous components, the tests for it run orders of magnitude faster than if we'd invoked the logic via HTTP, hitting the database and so on. We can write exhaustive tests that check every last edge case without paying a huge penalty in the time we wait for the tests to run.

5.3.1. Encapsulating the database

After validation, the other major concern we need to factor out is the database interaction. There are many ways to design an API for interacting with a database, and many frameworks are available offering different abstractions. For our example, we're going to take the small step of wrapping the existing database logic up in a few functions that present a high-level API that describes what we want to do. We'll make one module for each kind of record in our database: a `UserService` for managing user account records, and a `ChatService` for dealing with chatroom messages. This is fairly arbitrary delineation, and I'm not presenting it as the only 'right' approach; how you divide up your application will depend on the kinds of data you are handling, how those records and related, where they are stored, which parts of the application need access to them, and other factors. The important thing here is simply that we've moved the logic into functions that aren't coupled to either the HTTP request/response cycle, or the validation logic. We've separated persistence code from presentation (the HTTP interface) and business logic (validation).

The `validation` module is a single object rather than a class, because it has no state and no dependencies. There are no parameters that vary the behaviour of the module; calling its functions with the same input will *always* produce the same output. In contrast, a service object does have some state: it needs a database connection which needs to be able to connect to different databases depending on whether we're running tests or running the app in production. Similarly, if you have internal services that provide HTTP APIs, your application's service classes will need an HTTP client that's bound to the right host for the current environment.

We don't want each service class to make its own database connection; this would result in the app making multiple connections to the database, and spreading the connection logic out across the codebase. With Redis, having multiple connections per process is simply unnecessary, since it processes all commands from all clients sequentially. You don't get parallelism by adding more connections. With other databases, it may be beneficial to open multiple connections to run concurrent queries, but you shouldn't open too many; most systems use a connection pool to keep a list of available connections for the app to pick up, use and return to the pool. This logic is often complex and should be kept in one place, rather than spread out and mixed in with application logic.

There are also testability benefits; as we've seen a little already, and as we will discuss further in Section 5.3.2, "Dependency injection", passing in dependencies as arguments to functions can make it easier to apply mock-based testing to those functions, compared to having the functions construct or ask for their dependencies themselves. So, our service objects will be based on classes whose constructors take a database connection as a parameter, preventing the class from having to implement any connection-handling logic itself.

Let's start with the `UserService`. We have one user-related action in our app: we need users to be able to register. `UserService` provides the database logic for this action, and we've factored some of the internal details of this routine out into private methods whose names clarify what they do. Aside from that, this is the same logic that was originally wrapped up inside the `POST /users` request handler, but notice that it contains no HTTP-related code. `UserService.register()` yields an error (or `null`), a boolean indicating whether a new account was created, and the user's data. This is enough for the HTTP layer to pick the right status code (`500 Internal Server Error`, `201 Created` or `200 OK`) and construct a JSON response, but it also allows this API to be called by components that aren't related to HTTP.

Figure 5.42. `node/services/lib/user_service.js`

```
var async = require("async")

var UserService = function(redis) {
  this._redis = redis
}

UserService.prototype.register = function(username, callback) {
  var userData = {username: username},
      redis    = this._redis,
      self     = this

  async.waterfall([
    function(cb) {
      redis.sadd("users", username, cb)
    }, function(added, cb) {
      if (added === 1) {
        self._create(userData, cb)
      } else {
        self._findByUsername(username, cb)
      }
    }
  ], callback)
}

UserService.prototype._create = function(userData, callback) {
  var redis = this._redis

  async.waterfall([
    function(cb) {
      redis.incr("counters:user", cb)
    }, function(id, cb) {
      userData.id = id
      redis.hmset("users:" + id, userData, cb)
    }, function(ok, cb) {
      redis.set("index:users:" + userData.username, userData.id, cb)
    }, function(ok, cb) {
      cb(null, true, userData)
    }
  ], callback)
}

UserService.prototype._findByUsername = function(username, callback) {
  var redis = this._redis

  async.waterfall([
    function(cb) {
      redis.get("index:users:" + username, cb)
    }, function(id, cb) {
      redis.hgetall("users:" + id, cb)
    }, function(userData, cb) {
      userData.id = parseInt(userData.id, 10)
      cb(null, false, userData)
    }
  ], callback)
}

module.exports = UserService
```

To test this module, we'll use black-box testing. We could use mock-based testing, passing in a fake Redis client and thus avoiding any I/O and making the tests run faster, but as we've already seen, mocking a complex sequence of interactions is rather painful. Also, talking to the database is the entire responsibility of this module, so it makes sense to check that it actually talks to a real database correctly.

Putting database logic in its own module means we can reuse it across the application. Code that only needs to be written once only needs to be tested once, and by having black-box tests of this low-level component, we can fake the service APIs out in higher-level tests, avoiding the slowness of database I/O without losing any test coverage. The test suite mirrors the tests that we originally performed in Figure 5.30, "`node/chat_api/spec/register_spec.js`", only instead of checking the results via an HTTP interface we can call the service methods directly.

Figure 5.43. *`node/services/spec/user_service_spec.js`*

```
var JS          = require("jstest"),
    redis       = require("redis"),
    UserService = require("../lib/user_service")

JS.Test.describe("UserService", function() { with(this) {
  before(function() { with(this) {
    this.db = redis.createClient()
    db.select(15)
    this.service = new UserService(db)
  }})

  after(function(resume) { with(this) {
    db.flushdb(resume)
  }})

  it("registers a new user", function(resume) { with(this) {
    service.register("bob", function(error, created, userData) {
      resume(function() {
        assertNull( error )
        assert( created )
        assertEqual( {id: 1, username: "bob"}, userData )
    })})
  }})

  describe("with an existing user", function() { with(this) {
    before(function(resume) { with(this) {
      service.register("bob", resume)
    }})

    it("assigns sequential IDs", function(resume) { with(this) {
      service.register("alice", function(error, created, userData) {
        resume(function() {
          assertNull( error )
          assert( created )
          assertEqual( {id: 2, username: "alice"}, userData )
      })})
    }})

    it("returns existing users", function(resume) { with(this) {
      service.register("bob", function(error, created, userData) {
        resume(function() {
          assertNull( error )
          assertNot( created )
          assertEqual( {id: 1, username: "bob"}, userData )
      })})
    }})
  }})
}})
```

5.3.2. Dependency injection

In Figure 5.42, “`node/services/lib/user_service.js`”, we pass in a Redis client as a constructor argument, instead of having the constructor call `redis.createClient()` itself. This technique of passing dependencies into a component, rather than having that component reach out into the world to ask for its dependencies, is called *dependency injection*. It lets us change whether the service, in this case, uses a real Redis client or a fake, simply by changing the argument we pass into it. We could have written the constructor like this:

Figure 5.44. `UserServer` with a hard-coded Redis dependency

```
var async = require("async"),
    redis = require("redis")

var UserService = function(config) {
  this._redis = redis.createClient(config.port, config.host)
  this._redis.select(config.database)
}
```

However, this would mean that if we wanted to make `UserService` use a fake Redis client, we’d need to stub `redis.createClient()` in our tests:

Figure 5.45. Making the `redis` module return a fake client

```
var redis = require("redis"), client = {}
stub(redis, "createClient").returns(client)
```

This would make `UserService` receive a fake client, but it would also mean that the `redis.createClient()` call in our tests would receive a fake client, since that relies on the same `redis` module to make a connection. With this approach, we can’t create a situation where some modules get to talk to the database but the module under test is insulated. Whenever you stub out a shared API *globally*, whether that be a database API, the HTTP interface, or access to the filesystem, you affect *all* the modules that rely on that API, which can prevent your whole system from working.

We saw one such problem in Section 5.2.5, “Working with time”, where stubbing out `Date.now()` stopped the tests from running because the framework relies indirectly on `Date.now()` working properly. This should come as no surprise; `Date` is a global object and so any part of your application, your libraries or Node itself can call on it. If you stub out a function that some library you’re using depends on, the library may well stop working. Dependency injection solves this problem by letting us pass fake objects into the module we’re trying to test, without disrupting the behaviour of anything else.

If we’ve decided that `redis` should be passed in as an argument rather than fetched using `require("redis")`, what about `async`? `UserService` depends on that too, shouldn’t we inject it rather than hard-coding a module dependency? To answer this question, we should consider whether the dependency is a logically separate component of the system, where a clearly defined boundary exists that we might want to stub out. The database is clearly a major logical component of the system; we talk about *the* database or *the* user service, and we’d include them if drawing a diagram of the system. But we wouldn’t talk about *the* `async`; it’s not a logical component of the system, it’s just an implementation detail of a component’s internals. The same applies to a library like Underscore[19], or a stream utility like concat-stream[20] or through[21]. These are things we use to construct our system components, but they are not components in their own right. They are implementation details rather than architectural boundaries, and you’ll rarely if ever want to stub or mock them.

[19]http://underscorejs.org/
[20]https://www.npmjs.org/package/concat-stream
[21]https://www.npmjs.org/package/through

Let's complete our tour of the extracted persistence logic by looking at `ChatService`. This uses the same design principles we discussed above, taking a Redis database connection as a constructor argument and exposing an API that supports the operations the app needs to perform. Remember that the app needs to let a user post a message to a room, and it needs to let people poll a room for messages since a timestamp, and we can translate those requirements into two methods:

- `postMessage(room, userId, message, now, callback)`
- `pollMessages(room, since, now, callback)`

As we discussed in Section 5.2.5, "Working with time", these methods take any time values they need as arguments rather than generating their own timestamps; this is going to make testing them much easier since it lets us inject fixed timestamps into the module without having to stub out any shared APIs. They also take all the pieces of information they need — the room names, user IDs, and so on — as arguments, so this module does not need to know how to get hold of those values: it does not need to know how they are represented in an HTTP request and how to extract them. So, just like `UserService`, this module can be reused to support apps that don't use HTTP for their user interface.

Aside from that, the logic is mostly extracted from Figure 5.33, "`node/chat_api/routes/post_message.js`" and Figure 5.37, "`node/chat_api/routes/get_messages.js`", with one small change: the logic for looking up the details of each individual message during polling has been pulled out into its own method for clarity.

You'll notice that these methods need to refer to user data — checking that a given user ID exists, and looking up usernames for messages. We could have used `UserService` to encapsulate that logic rather than implementing it in the `ChatService` module, but I've chosen not to do that. I usually try to keep service classes decoupled from one another to keep things simple; one pain commonly encountered when growing a large app is the interconnections between areas of business logic become tangled and hard to reason about, making things hard to test and change. I aim for architectures where the stack is separated into distinct layers, where components can talk to things in the layer beneath them, but they can't talk to components in the same layer or above them. This results in a tree of dependencies rather than a tangled network where it's possible to go round in circles trying to trace the code paths in the application.

But of course, `ChatService` still *uses* data managed by the `UserService`, so there is some coupling there. If the complexity of interacting with the user data became greater, then I'd consider moving the logic into `UserService` and making `ChatService` depend on it, or I'd try to find a way to break the coupling entirely; this design is fine for now but you should constantly reassess whether your design needs changing to cope with new requirements.

Figure 5.46. `node/services/lib/chat_service.js`

```
var async = require("async")

var ChatService = function(redis) {
  this._redis = redis
}

ChatService.prototype.postMessage = function(room, userId, message, now, callback) {
  var redis = this._redis, messageData

  async.waterfall([
    function(cb) {
      redis.exists("users:" + userId, cb)
    }, function(exists, cb) {
      var error = (exists === 0) ? "User #" + userId + " does not exist" : null
      cb(error && new Error(error))
    }, function(cb) {
      redis.incr("counters:message", cb)
    }, function(id, cb) {
      messageData = {id: id, timestamp: now, userId: userId, message: message}
      redis.hmset("messages:" + id, messageData, cb)
    }, function(ok, cb) {
      redis.zadd("rooms:" + room + ":messages", now, messageData.id, cb)
    }, function(ok, cb) {
      delete messageData.userId
      cb(null, messageData)
    }
  ], callback)
}

ChatService.prototype.pollMessages = function(room, since, now, callback) {
  var self = this

  async.waterfall([
    function(cb) {
      self._redis.zrangebyscore("rooms:" + room + ":messages", since, now, cb)
    }, function(messageIds, cb) {
      async.map(messageIds, self.getMessage.bind(self), cb)
    }
  ], callback)
}

ChatService.prototype.getMessage = function(messageId, callback) {
  var redis = this._redis, messageData

  async.waterfall([
    function(cb) {
      redis.hgetall("messages:" + messageId, cb)
    }, function(data, cb) {
      messageData = data
      messageData.id = parseInt(messageData.id, 10)
      messageData.timestamp = parseInt(messageData.timestamp, 10)
      redis.hget("users:" + messageData.userId, "username", cb)
    }, function(username, cb) {
      messageData.username = username
      delete messageData.userId
      cb(null, messageData)
    }
  ], callback)
}

module.exports = ChatService
```

The tests for this module use the same strategy as in Figure 5.43, “`node/services/spec/user_service_spec.js`”, treating the service as a black box and testing it by interacting only with its public API. Notice that since we can now pass in fixed timestamps, we can test message posting by submitting a message with a known timestamp, then polling for messages on an interval that includes that timestamp to check whether or not we get the message back. In the example below, we see that a message with an unknown user ID is not accepted, while one with a valid ID is successful.

The one quirk with this test is that although `ChatService` doesn’t depend on `UserService`, we have used `UserService` to create user data. We’ve done this because `ChatService` does need to interact with some user data, and we want to make sure that the data it sees during testing is a realistic recreation of what it will see in production. And what better way to do that than by having the `UserService` create the data for us? We could reproduce what `UserService` does ourselves; which approach we use will depend on how complex and time-consuming it is to invoke the real `UserService` in a test, compared to maintaining another copy of its storage logic to reproduce its effects.

Figure 5.47. *`node/services/spec/chat_service_post_message_spec.js`*

```
var JS          = require("jstest"),
    redis       = require("redis"),
    ChatService = require("../lib/chat_service"),
    UserService = require("../lib/user_service")

JS.Test.describe("ChatService.postMessage()", function() { with(this) {
  before(function(resume) { with(this) {
    this.db      = redis.createClient()
    this.users   = new UserService(db)
    this.service = new ChatService(db)
    this.now     = 1393101364563

    db.select(15)
    users.register("bob", resume)
  }})

  after(function(resume) { with(this) {
    db.flushdb(resume)
  }})

  it("does not post the message given an unknown user ID", function(resume) { with(this) {
    service.postMessage("garage", 2, "Hello", now, function(error, message) {
      resume(function(resume) {
        assertEqual( "User #2 does not exist", error.message )
        assertEqual( undefined, message )

        service.pollMessages("garage", now - 10, now + 10, function(error, messages) {
          resume(function() { assertEqual( [], messages ) })
    })})})
  }})

  it("posts the message with a valid user ID", function(resume) { with(this) {
    service.postMessage("garage", 1, "Hello", now, function(error, message) {
      resume(function(resume) {
        assertNull( error )
        assertEqual( {id: 1, timestamp: now, message: "Hello"}, message )

        service.pollMessages("garage", now - 10, now + 10, function(error, messages) {
          resume(function() {
            assertEqual( [{id: 1, username: "bob", timestamp: now, message: "Hello"}],
                         messages )
    })})})})
  }})
}})
```

The tests for polling for new messages look much the same, only as well as creating some users at the start of each test we also create some messages to check what `pollMessages()` does in different situations. Again, we can pass in fixed timestamps rather than stubbing things, but note how the tests all use offsets from a fixed time rather than using absolute timestamps. I find this makes it easier to see what's going on, since it's easier to mentally compare two small offsets than to compare two full Unix timestamps.

Figure 5.48. `node/services/spec/chat_service_poll_messages_spec.js`

```
var JS          = require("jstest"),
    redis       = require("redis"),
    ChatService = require("../lib/chat_service"),
    UserService = require("../lib/user_service")

JS.Test.describe("ChatService.pollMessages()", function() { with(this) {
  before(function(resume) { with(this) {
    this.db      = redis.createClient()
    this.users   = new UserService(db)
    this.service = new ChatService(db)
    this.now     = 1393101364563

    db.select(15)
    users.register("alice", function() { users.register("bob", resume) })
  }})

  after(function(resume) { with(this) {
    db.flushdb(resume)
  }})

  describe("with messages posted in a room", function() { with(this) {
    before(function(resume) { with(this) {
      service.postMessage("attic", 1, "First post!", now - 200, function() {
        service.postMessage("attic", 2, "Anyone here?", now - 100, resume)
      })
    }})

    it("yields all the messages given an old timestamp", function(resume) { with(this) {
      service.pollMessages("attic", now - 5000, now, function(error, messages) {
        resume(function() {
          assertEqual([
            {id: 1, username: "alice", timestamp: now - 200, message: "First post!"},
            {id: 2, username: "bob", timestamp: now - 100, message: "Anyone here?"}
          ], messages)
      })})
    }})

    it("does not yield messages older than the timestamp", function(resume) { with(this) {
      service.pollMessages("attic", now - 150, now, function(error, messages) {
        resume(function() {
          assertEqual([
            {id: 2, username: "bob", timestamp: now - 100, message: "Anyone here?"}
          ], messages)
      })})
    }})

    it("yields no messages for other rooms", function(resume) { with(this) {
      service.pollMessages("basement", now - 5000, now, function(error, messages) {
        resume(function() {
          assertEqual( [], messages )
      })})
    }})
  }})
}})
```

5.3.3. Wiring everything together

We’ve pushed most of the logic in our original chatroom app down into individual modules, and now we need to glue them back together to produce a workable app. Since all the real logic now lives somewhere else, putting an Express app together is mostly uncomplicated grunt-work, mapping HTTP requests onto methods on the underlying modules. Each request handler below performs these distinct tasks: it extracts the relevant pieces of information from various parts of the HTTP request, it validates the data, and if that succeeds then it passes the extracted data off to a service method before returning an HTTP response.

Figure 5.49. `node/services/app.js`

```
var connect = require("connect"),
    express = require("express"),
    http    = require("http"),
    redis   = require("redis"),
    render  = require("./lib/render")

module.exports = function(validation, users, chats) {
  var app = express()
  app.use(connect.urlencoded())

  app.post("/users", function(request, response) {
    var username = request.body.username

    var errors = validation.checkUser({username: username})
    if (errors) return response.json(409, {errors: errors})

    users.register(username, function(error, created, userData) {
      render(response, error, created ? 201 : 200, userData)
    })
  })

  app.post("/chat/:roomName", function(request, response) {
    var roomName  = request.params.roomName,
        userId    = request.body.userId,
        message   = request.body.message,
        timestamp = Date.now()

    var errors = validation.checkMessage({roomName: roomName, message: message})
    if (errors) return response.json(409, {errors: errors})

    chats.postMessage(roomName, userId, message, timestamp, function(error, messageData) {
      render(response, error, 201, messageData)
    })
  })

  app.get("/chat/:roomName", function(request, response) {
    var roomName = request.params.roomName,
        since    = parseInt(request.query.since || "0", 10),
        now      = Date.now()

    var errors = validation.checkPoll({roomName: roomName, since: since})
    if (errors) return response.json(409, {errors: errors})

    chats.pollMessages(roomName, since, now, function(error, messages) {
      render(response, error, 200, {messages: messages})
    })
  })

  return http.createServer(app)
}
```

Just as we used dependency injection to pass a Redis connection into the service classes, here we've used dependency injection to pass the `validation` module and instances of the `UserService` and `ChatService` classes into the application. This is going to let us test the HTTP layer of the app without hitting any of the underlying logic or database interaction. The app relies on two small helper functions for sending HTTP responses; `render()` decides what type of response to send given a possible error, status and response data:

Figure 5.50. `node/services/lib/render.js`

```
module.exports = function(response, error, status, data) {
  if (error) {
    response.json(500, {errors: [error.message]})
  } else {
    response.json(status, data)
  }
}
```

All the start-up script for the application has to do is create a database connection, create instances of all the service classes based on that connection, and create a copy of the app based on those services; it builds the stack up in layers that glue together easily. Splitting the app into smaller decoupled components has not made it significantly more costly to wire together and deploy, and it has made the architectural structure of the system more explicit.

Figure 5.51. `node/services/main.js`

```
var redis = require("redis"),
    app   = require("./app"),
    env   = process.env

var validation  = require("./lib/validation"),
    UserService = require("./lib/user_service"),
    ChatService = require("./lib/chat_service")

var db = redis.createClient(env.REDIS_PORT, env.REDIS_HOST)
db.select(env.REDIS_DB || 0)

var users = new UserService(db),
    chats = new ChatService(db)

var server = app(validation, users, chats)
server.listen(env.PORT)
```

Although this app is much less complex than our original design, it's still worth writing a few 'sanity check' tests for it, if only to act as documentation for how the HTTP interface is supposed to work. But because we've tested all the behaviour of the individual components already, we don't need to write exhaustive tests at this level. All we need to do is check that the app maps its HTTP input onto the right internal service calls, and that it maps the expected responses from those services back onto the correct kinds of HTTP responses.

For example, let's take the `POST /users` request. This has four broad types of response: It can:

- Return a `409 Conflict` if the given username is invalid
- Return a `201 Created` if the username is valid and does not already exist
- Return a `200 OK` if the username is valid and *does* already exist
- Return a `500 Internal Server Error` if there's a problem while talking to the database

Actually, this description contains too much detail. Take a look at the function that handles the `POST /users` request:

Figure 5.52. `node/services/app.js`

```
app.post("/users", function(request, response) {
  var username = request.body.username

  var errors = validation.checkUser({username: username})
  if (errors) return renderValidation(response, errors)

  users.register(username, function(error, created, userData) {
    render(response, error, created ? 201 : 200, userData)
  })
})
```

This request handler will return a `409` if there are any validation errors. It doesn't care what those errors *are*; they could be about an invalid username, they could be that we don't accept sign-ups on Tuesdays, it doesn't make any difference to this function. So when we test how the HTTP interface deals with validation we only need to check that it extracts the right data from the request and passes it to the `validation` module, that it returns a `409` if `validation` returns an array of errors, and that it proceeds successfully if `validation` returns `null` or some other falsey value. We don't need to exactly replicate the behaviour of `validation` in our stubs, we only need to return the right *type* of value to trigger the HTTP behaviour we're testing.

Similarly, we don't need to *actually* generate a database error to check the server returns a `500`, we only need to make `users.register()` yield an `Error` object as the first callback argument. We can switch `created` from `true` to `false` to make sure this produces a `201` or `200`, and we don't really care what `userData` is as long as it ends up being correctly reproduced as JSON in the response body. Having these abstract interfaces means we can hide a lot of the detail and use totally made-up values in our stubs, provided they are the same type of values the real service modules return. This makes the problem of keeping your stubs in sync with your production code considerably easier.

Let's have a look at the test suite. It begins by making fake objects to stand in for the `validation`, `users` and `chats` instances, it creates an app using these fakes and starts it up to accept requests. The first thing we check is that the handler extracts the right data from the request and validates it, using `expect(validation, "checkUser").given(…)`. We make `validation.checkUser()` return an array of errors, and assert the response is a `409`. This first test makes sure that `users.register()` is not called simply by not setting an expectation for it; since we're using blank objects to stand in for services, any unexpected method call we've not set up a stub for will throw an error.

The subsequent tests proceed in a similar style, setting some mock expectations about what calls the request should cause, providing some canned return values for these calls and checking those values are mapped onto the right kind of HTTP response.

Figure 5.53. *`node/services/spec/register_spec.js`*

```
var JS    = require("jstest"),
    app   = require("../app"),
    steps = require("../../chat_api/spec/server_steps")

JS.Test.describe("POST /users", function() { with(this) {
  include(steps)

  before(function() { with(this) {
    this.validation  = {}
    this.userService = {}
    this.chatService = {}

    this.app = app(validation, userService, chatService)
    startServer(app)
  }})

  after(function() { with(this) {
    stopServer(app)
  }})

  it("returns a 409 Conflict if the input is invalid", function() { with(this) {
    expect(validation, "checkUser").given({username: "zack"}).returns(["Invalid username"])

    post("/users", {username: "zack"})
    checkStatus(409)
    checkJSON({errors: ["Invalid username"]})
  }})

  it("returns a 201 Created if the user is valid and new", function() { with(this) {
    expect(validation, "checkUser").given({username: "zack"}).returns(null)
    expect(userService, "register").given("zack").yields([null, true, {id: 1}])

    post("/users", {username: "zack"})
    checkStatus(201)
    checkJSON({id: 1})
  }})

  it("returns a 200 OK if the user is valid and not new", function() { with(this) {
    expect(validation, "checkUser").given({username: "zack"}).returns(null)
    expect(userService, "register").given("zack").yields([null, false, {id: 1}])

    post("/users", {username: "zack"})
    checkStatus(200)
    checkJSON({id: 1})
  }})

  it("returns a 500 if the user service fails", function() { with(this) {
    expect(validation, "checkUser").given({username: "zack"}).returns(null)
    expect(userService, "register").given("zack").yields([new Error("DB is offline")])

    post("/users", {username: "zack"})
    checkStatus(500)
    checkJSON({errors: ["DB is offline"]})
  }})
}})
```

Our unit tests now cover the HTTP interface, the validation logic, and the database interaction, but they do it much more efficiently than in our original design. Rather than testing each validation rule via the HTTP interface, including hitting the database, we test the validation logic directly and only test broad type differences — returning `null` versus returning an array of errors — via HTTP. Notice how there is only one test for validation failure in the above suite, while Figure 5.41, “`node/services/spec/`

validation_spec.js” contains two tests for user validation failure. As the amount of validation logic increases, we can add more unit tests for it but we’d still only have one test at the HTTP level. This means most of our tests end up running an order of magnitude faster than they would do in our original design, saving a lot of time by keeping the test suite fast.

We go through a similar process to test the POST and GET /chat/:roomName endpoints, creating fake service objects, making sure request data is correctly extracted and validated, and tweaking the response of the service objects to see how this varies the HTTP response. I’ve only shown a couple of cases here — the validation failure and success cases — but the pattern of each test is so consistent it should be clear how to extend this to cover things I’ve missed, for example cases where database errors or missing user IDs occur.

Figure 5.54. *node/services/spec/post_message_spec.js*

```
var JS    = require("jstest"),
    app   = require("../app"),
    steps = require("../../chat_api/spec/server_steps")

JS.Test.describe("POST /chat/:roomName", function() { with(this) {
  include(steps)

  before(function() { with(this) {
    this.validation  = {}
    this.userService = {}
    this.chatService = {}

    this.app = app(validation, userService, chatService)
    startServer(app)
  }})

  after(function() { with(this) {
    stopServer(app)
  }})

  it("returns a 409 Conflict if the input is invalid", function() { with(this) {
    expect(validation, "checkMessage")
        .given({roomName: "basement", message: "Hi, there!"})
        .returns(["Invalid message"])

    post("/chat/basement", {userId: "84", message: "Hi, there!"})
    checkStatus(409)
    checkJSON({errors: ["Invalid message"]})
  }})

  it("returns a 201 Created if the input is valid", function() { with(this) {
    expect(validation, "checkMessage")
        .given({roomName: "basement", message: "Hi, there!"})
        .returns(null)

    expect(chatService, "postMessage")
        .given("basement", "84", "Hi, there!", instanceOf("number"))
        .yields([null, {id: 99}])

    post("/chat/basement", {userId: "84", message: "Hi, there!"})
    checkStatus(201)
    checkJSON({id: 99})
  }})
}})
```

Figure 5.55. `node/services/spec/get_messages_spec.js`

```
var JS    = require("jstest"),
    app   = require("../app"),
    steps = require("../../chat_api/spec/server_steps")

JS.Test.describe("GET /chat/:roomName", function() { with(this) {
  include(steps)

  before(function() { with(this) {
    this.validation  = {}
    this.userService = {}
    this.chatService = {}

    this.app = app(validation, userService, chatService)
    startServer(app)
  }})

  after(function() { with(this) {
    stopServer(app)
  }})

  it("returns a 409 Conflict if the input is invalid", function() { with(this) {
    expect(validation, "checkPoll")
        .given({roomName: "basement", since: 1234})
        .returns(["Invalid request"])

    get("/chat/basement", {since: "1234"})
    checkStatus(409)
    checkJSON({errors: ["Invalid request"]})
  }})

  it("returns a 200 OK if the input is valid", function() { with(this) {
    expect(validation, "checkPoll")
        .given({roomName: "basement", since: 1234})
        .returns(null)

    expect(chatService, "pollMessages")
        .given("basement", 1234, instanceOf("number"))
        .yields([null, [{id: 1}, {id: 2}]])

    get("/chat/basement", {since: "1234"})
    checkStatus(200)
    checkJSON({messages: [{id: 1}, {id: 2}]})
  }})
}})
```

In summary, testing the HTTP interface is no longer about full-stack testing, it's about making sure the app maps user input onto the right internal function calls, and that it integrates things together properly. We already have tests that show the internal components work, and reproducing those tests here is a waste of effort and a waste of time spent running slow tests. HTTP tests are slower than tests of internal logic, so we want as few of them as we can get away with. Wherever possible, push logic embedded in HTTP request handlers down into objects you can test without going via HTTP.

5.4. Authentication

So far, we've made no attempt to secure our chatroom app; people can pick any username they like without proving their identity. Most real-world applications have authenticated user accounts, and restrict access to resources by checking the user's identity on each request. This is typically done by requiring a password to log in[22], which creates a session between the user's browser and the application

[22]Many apps choose to outsource this part of the process to a third party, for example relying on Facebook Connect to authenticate users.

server. On successfully logging in, the server issues the browser with a cookie that represents the session, and the browser includes this cookie in all future requests to the server to identify the user making the request.

In this section we'll examine various testing problems related to authentication, as well as introducing ORMs[23], a common pattern for interacting with databases.

5.4.1. Passwords

The first step in building an authentication process for many apps is to let users set a password when they sign up, and have that password verified when they log in. An app should never store passwords in plain text, but should hash them using a per-user salt and a tunable slow hashing/key-stretching function. (At time of writing, PBKDF2[24], bcrypt[25] and scrypt[26] are commonly recommended.) Below is a `User` class with two public methods: `setPassword()` takes a password, generates a random salt and hashes the password with PBKDF2, and `checkPassword()` hashes a password with the existing salt and checks the result against the stored hash. `checkPassword()` uses a constant-time string comparison to avoid timing attacks.

Figure 5.56. `node/passwords/user.js`

```
var crypto = require("crypto"),
    _      = require("underscore")

var User = function(attributes) {
  this._attributes = _.clone(attributes || {})
}

User.KEY_LENGTH  = 32
User.WORK_FACTOR = 200000

User.prototype.setPassword = function(password) {
  var attrs = this._attributes

  attrs.salt = crypto.randomBytes(User.KEY_LENGTH).toString("hex")
  attrs.hash = this._hashPassword(password).toString("hex")
}

User.prototype.checkPassword = function(password) {
  var expected = new Buffer(this._attributes.hash || "", "hex"),
      actual   = this._hashPassword(password),
      n        = Math.max(actual.length, expected.length),
      diff     = 0

  for (var i = 0; i < n; i++) {
    diff |= actual[i] ^ expected[i]
  }
  return diff === 0
}

User.prototype._hashPassword = function(password) {
  var salt = new Buffer(this._attributes.salt || "", "hex")
  return crypto.pbkdf2Sync(password, salt, User.WORK_FACTOR, User.KEY_LENGTH)
}

module.exports = User
```

[23]ORM stands for Object-Relational Mapping; see http://en.wikipedia.org/wiki/Object-relational_mapping.
[24]http://en.wikipedia.org/wiki/PBKDF2
[25]http://en.wikipedia.org/wiki/Bcrypt
[26]http://en.wikipedia.org/wiki/Scrypt

Before we go any further, I must point out this code is for demonstration purposes only; a full discussion of the cryptography and security concerns is beyond the scope of this book. It is a reasonable minimal password verification implementation, but you should consult a security expert before deploying this sort of code in production.

It is fairly simple to test a class like this, what with it having no dependencies on other application components. We can just create a `User` object, set a password, and check that it accepts the right password and rejects incorrect passwords. We should check passwords that are truncations or extensions of the correct password, or use the wrong case, to make sure only an exact match is accepted.

Figure 5.57. `node/passwords/user_spec.js`

```
var JS   = require("jstest"),
    User = require("./user")

JS.Test.describe("User (slow)", function() { with(this) {
  before(function() { with(this) {
    this.user = new User()
  }})

  describe("with no password", function() { with(this) {
    it("rejects all passwords", function() { with(this) {
      ["a", "list", "of", "password", "attempts"].forEach(function(pw) {
        assertNot( user.checkPassword(pw) )
      })
    }})
  }})

  describe("with a password", function() { with(this) {
    before(function() { with(this) {
      user.setPassword("secret")
    }})

    it("accepts the correct password", function() { with(this) {
      assert( user.checkPassword("secret") )
    }})

    it("rejects incorrect passwords", function() { with(this) {
      assertNot( user.checkPassword("secre") )
      assertNot( user.checkPassword("secrets") )
      assertNot( user.checkPassword("wrong") )
    }})

    it("rejects a password with the wrong case", function() { with(this) {
      assertNot( user.checkPassword("SeCrEt") )
    }})
  }})
}})
```

These tests correctly verify the behaviour we need, but they are slow. *Painfully* slow. This is by design; a password hash needs to run slowly to make it expensive to attempt a brute-force cracking attack, but it does make it rather tedious to wait for the tests to finish.

Figure 5.58. Running user tests with slow authentication

```
$ TEST=slow node node/passwords/test.js
Loaded suite: User (slow)

....

Finished in 7.702 seconds
4 tests, 10 assertions, 0 failures, 0 errors
```

These tests run slowly because of the work factor we've chosen. A *work factor* is a parameter to a password hashing algorithm that tells it how many iterations of its internal hashing routine to perform; this affects how long the function takes. You should set your work factor so that hashing a password takes at least half a second on your hardware. For Node's PBKDF2 function on my machine, that work factor is 200,000.

While slow hashing is the right thing to do in production, it can wreak havoc on your tests. Every test that goes through your HTTP layer and requires the user to log in to access the functionality you're testing will take *at least* as long as it takes to hash a password, probably double that since you need to create a user object at the start of the test so you can log in as that user. Having tests take over a second each is a disaster; developers don't want to wait for minutes every time they run the tests, and if no-one runs the tests people tend to stop writing them at all.

However, notice that, rather than hard-coding the work factor inside `User._hashPassword()` or keeping it in a local variable outside the module, we've set it as a property on the `User` constructor. This means we can access it from outside the module, and change it to a smaller number while the tests run. The following test suite is identical to the previous one, except for one additional line: `stub(User, "WORK_FACTOR", 1)` sets `User.WORK_FACTOR` to `1` for the duration of the tests, and speeds everything up. We could equally put `User.WORK_FACTOR = 1` in our `before()` block, but using a stub makes sure that the setting is reverted to its original value at the end of each test.

Figure 5.59. `node/passwords/fast_user_spec.js`

```
var JS   = require("jstest"),
    User = require("./user")

JS.Test.describe("User (fast)", function() { with(this) {
  before(function() { with(this) {
    stub(User, "WORK_FACTOR", 1)
    this.user = new User()
  }})

  describe("with no password", function() { with(this) {
    it("rejects all passwords", function() { with(this) {
      ["a", "list", "of", "password", "attempts"].forEach(function(pw) {
        assertNot( user.checkPassword(pw) )
      })
    }})
  }})

  describe("with a password", function() { with(this) {
    before(function() { with(this) {
      user.setPassword("secret")
    }})

    it("accepts the correct password", function() { with(this) {
      assert( user.checkPassword("secret") )
    }})

    it("rejects incorrect passwords", function() { with(this) {
      assertNot( user.checkPassword("secre") )
      assertNot( user.checkPassword("secrets") )
      assertNot( user.checkPassword("wrong") )
    }})

    it("rejects a password with the wrong case", function() { with(this) {
      assertNot( user.checkPassword("SeCrEt") )
    }})
  }})
}})
```

Reducing the work factor during testing has made our test suite run over 600 times faster. You should aim for each unit test to only take a few milliseconds so that the test suite doesn't become prohibitively slow to run over time.

Figure 5.60. Running user tests with fast authentication

```
$ TEST=fast node node/passwords/test.js
Loaded suite: User (fast)

....

Finished in 0.012 seconds
4 tests, 10 assertions, 0 failures, 0 errors
```

Although this works, there's something unsatisfying about tweaking what is essentially a global setting to achieve our goals. It's not likely to affect other parts of the system in this case, so it's not a big deal, but there are other reasons to avoid it in this case.

Over time, you will probably need to increase your work factor to cope with hardware getting faster. This is why we use tunable work factors: so we can keep making it expensive to hash passwords in the face of faster computers and dedicated cracking tools. With the code as is, changing the value of `User.WORK_FACTOR` would instantly invalidate all existing hashes; if they were generated with `WORK_FACTOR = 200` and we change `WORK_FACTOR` to `400` then a correct login password will no longer hash to the same value we have stored. So, for future-proofing, it's advisable to store the work factor (and possibly other parameters such as the key length and hashing algorithm used) as part of each user record, rather than making them global settings. The globals then serve only as default values for newly created users.

The `AdjustableUser` class shown below embodies this design by adding `workFactor` and `keyLength` properties to `this._attributes` and using those instead of `User.WORK_FACTOR` and `User.KEY_LENGTH` when hashing passwords. When a new `AdjustableUser` object is created, those constants are used to populate any attributes that are missing, but we can create an `AdjustableUser` with existing user data and that data will be used rather than the defaults.

Figure 5.61. `node/passwords/adjustable_user.js`

```
var crypto = require("crypto"),
    _      = require("underscore")

var AdjustableUser = function(attributes) {
  var attrs = this._attributes = _.clone(attributes || {})

  attrs.keyLength  = attrs.keyLength  || AdjustableUser.KEY_LENGTH
  attrs.workFactor = attrs.workFactor || AdjustableUser.WORK_FACTOR
}

AdjustableUser.KEY_LENGTH  = 32
AdjustableUser.WORK_FACTOR = 200000

AdjustableUser.prototype.setPassword = function(password) {
  var attrs = this._attributes

  attrs.salt = crypto.randomBytes(attrs.keyLength).toString("hex")
  attrs.hash = this._hashPassword(password).toString("hex")
}

AdjustableUser.prototype.checkPassword = function(password) {
  var expected = new Buffer(this._attributes.hash || "", "hex"),
      actual   = this._hashPassword(password),
      n        = Math.max(actual.length, expected.length),
      diff     = 0

  for (var i = 0; i < n; i++) {
    diff |= actual[i] ^ expected[i]
  }
  return diff === 0
}

AdjustableUser.prototype._hashPassword = function(password) {
  var attrs = this._attributes,
      salt  = new Buffer(attrs.salt, "hex")

  return crypto.pbkdf2Sync(password, salt, attrs.workFactor, attrs.keyLength)
}

module.exports = AdjustableUser
```

This design lets us configure hashing speed on a per-user basis, by passing a `workFactor` value into the constructor, *and* on a global basis by stubbing the value of `WORK_FACTOR`. Both of these approaches will come in handy in future tests, since some tests will have direct access to the user objects being created and others will not. `WORK_FACTOR` gives us a fallback to configure the system in cases where the tests don't have direct access to the user objects, for example when we are testing sign-up through the HTTP front-end.

Below is an example test suite for `AdjustableUser`, demonstrating how to set `workFactor` for a user object. It runs similarly quickly to the fast tests we saw above, in which `User.WORK_FACTOR` was stubbed to be `1`.

Figure 5.62. *`node/passwords/adjustable_user_spec.js`*

```
var JS   = require("jstest"),
    User = require("./adjustable_user")

JS.Test.describe("AdjustableUser", function() { with(this) {
  before(function() { with(this) {
    this.user = new User({workFactor: 1})
    user.setPassword("secret")
  }})

  it("accepts the correct password", function() { with(this) {
    assert( user.checkPassword("secret") )
  }})

  it("rejects incorrect passwords", function() { with(this) {
    assertNot( user.checkPassword("secre") )
    assertNot( user.checkPassword("secrets") )
    assertNot( user.checkPassword("wrong") )
  }})

  it("rejects a password with the wrong case", function() { with(this) {
    assertNot( user.checkPassword("SeCrEt") )
  }})
}})
```

To build up our authentication system, we're going to build a primitive object-relational mapper, or ORM. An ORM provides an object-based interface for working with database records; it provides convenience methods for searching the database, and it represents each record it finds as a single object. Those objects may have validation and business logic attached to them, since you often want to validate data before it gets saved to the database. So, in contrast to the services we built before, where data objects had no behaviour, and validation and persistence were kept in separate flat APIs, an ORM bundles all those things together into graphs of objects that know how to do everything your application needs them to do[27].

The final user API we're aiming for will look something like this, with operations to create new users and find existing ones, as well as validate user data and check passwords:

Figure 5.63. Overview of the `User` ORM

```
var alice = new User({username: "alice"})
alice.validate() // -> ["Password must not be blank"]

var bob = new User({username: "bob", password: "chips"})
bob.validate()              // -> null
bob.checkPassword("fish")   // -> false
bob.checkPassword("chips")  // -> true

bob.save(function(error) { ... })

User.findByUsername("bob", function(error, user) {
  user.get("username") // -> "bob"
})
```

In the database, we're going to represent things exactly as we did in Section 5.2.2, "Creating a database schema"; we're just building a different API to interact with that data. We built some of the methods in the above example — the password-related methods — in the previous section, so we can reuse that logic by subclassing `AdjustableUser`. We still need to add validation and persistence logic, and we can build those up in layers. Let's start with validation.

[27]Strictly speaking, the term ORM originally referred only to tools that map objects onto relational databases, but it is also used colloquially to refer to systems that use NoSQL data stores as well.

5.4.2. Validation

Our `AdjustableUser` class gives us a user object that we can instantiate with some attributes, and set and check a password on. We’re going to build on that class by validating the attributes we pass in, and making sure passwords are stored in hashed form rather than in plain text.

The `ValidatedUser` user class below takes a set of attributes, and if those attributes contain a `password`, the constructor calls `setPassword()` to convert the password to hashed form before deleting the plain-text `password` from the attributes. This makes sure that the plain-text password does not remain stored in the object and accidentally end up being written to the database. The class also provides `get()` and `set()` methods for interacting with the user’s attributes, and a `validate()` method. `validate()` inspects the user’s current state and returns an array of errors, or `null` if the object is valid.

Figure 5.64. `node/orm/validated_user.js`

```
var User = require("../passwords/adjustable_user"),
    util = require("util")

var ValidatedUser = function(attributes) {
  User.call(this, attributes)

  if (this._attributes.password) {
    this.setPassword(this._attributes.password)
    delete this._attributes.password
  }
}
util.inherits(ValidatedUser, User)

ValidatedUser.prototype.get = function(attribute) {
  return this._attributes[attribute]
}

ValidatedUser.prototype.set = function(attribute, value) {
  this._attributes[attribute] = value
}

ValidatedUser.prototype.validate = function() {
  var attrs  = this._attributes,
      errors = []

  if (attrs.username.length < 2) {
    errors.push("Usernames must have at least 2 characters")
  }
  if (!attrs.username.match(/^[a-z0-9_]+$/i)) {
    errors.push("Usernames may only contain letters, numbers and underscores")
  }
  if (!attrs.hash) {
    errors.push("Password must not be blank")
  }
  return errors.length > 0 ? errors : null
}

module.exports = ValidatedUser
```

When testing this class, we’re mainly concerned with the `valiate()` method. It should return `null` for a valid object, but any change to the object that renders it invalid should make `validate()` return an array containing the validation error. Note how we’re using `workFactor: 1` here; since creating a valid user involves setting (and therefore hashing) a password, we need to make sure the tests stay fast by setting a low work factor.

As I’ve mentioned before, this is my typical approach to validation testing: check that a valid object is valid, then make small perturbations to check they render it invalid. Using small changes means you can easily see what caused the `validate()` method to change its output.

Figure 5.65. *`node/orm/validated_user_spec.js`*

```
var JS            = require("jstest"),
    ValidatedUser = require("./validated_user")

JS.Test.describe("ValidatedUser", function() { with(this) {
  before(function() { with(this) {
    this.user = new ValidatedUser({username: "alice", password: "fish", workFactor: 1})
  }})

  it("returns no errors for a valid user", function() { with(this) {
    assertNull( user.validate() )
  }})

  it("returns an error for a short username", function() { with(this) {
    user.set("username", "n")
    assertEqual( ["Usernames must have at least 2 characters"], user.validate() )
  }})

  it("returns an error for an invalid username", function() { with(this) {
    user.set("username", "$%^&")
    assertEqual( ["Usernames may only contain letters, numbers and underscores"],
                 user.validate() )
  }})

  it("returns an error for a blank password", function() { with(this) {
    var user = new ValidatedUser({username: "alice"})
    assertEqual( ["Password must not be blank"], user.validate() )
  }})
}})
```

Note that there is one aspect of validation this class does not check: usernames must be unique. This constraint only makes sense if we’re actually saving users to a data store where we can apply the uniqueness constraint, and indeed uniqueness checking can only be properly implemented as part of *saving* the object. If you check validation by reading from the database before saving the object, then it’s possible for two user objects with the same name to both check the database, see that the name is not taken, and then both save themselves. In our design, we avoided this by using `SADD users` as part of the saving procedure, and we’ll do the same here.

5.4.3. Persistence

The final ‘layer’ to our user object is persistence; users must be able to save themselves to the database. In the case of existing users, this simply means using `HMSET` command to store the user’s attributes in a hash. But for new users (which we’ve modelled as any user without an ID), we need to first run `SADD users username` to check the username is available, then run `INCR counters:user` to generate an ID for the user, before we use `HMSET` to store the user keyed by its ID.

The `PersistedUser` class below has a `save()` method that implements this procedure, first calling `validate()` to make sure the object can even be saved at all. The constructor takes `id` as an extra parameter, so we can distinguish new users from those loaded via database queries, and we’ve provided a couple of convenience methods for finding existing users: `findById()` and `findByUsername()`.

Figure 5.66. `node/orm/persisted_user.js`

```
var store         = require("./store"),
    ValidatedUser = require("./validated_user"),
    util          = require("util"),
    async         = require("async")

PersistedUser = function(attributes, id) {
  ValidatedUser.call(this, attributes)
  this.id = id
}
util.inherits(PersistedUser, ValidatedUser)

PersistedUser.prototype.save = function(callback) {
  var errors = this.validate()
  if (errors) return callback(errors)

  var conn  = store.getConnection(),
      attrs = this._attributes,
      self  = this

  if (this.id) return conn.hmset("users:" + this.id, this._attributes, callback)

  async.waterfall([
    function(cb) {
      conn.sadd("users", attrs.username, cb)
    }, function(added, cb) {
      var error = (added === 0) ? ["Username already exists"] : null
      cb(error)
    }, function(cb) {
      conn.incr("counters:user", cb)
    }, function(id, cb) {
      self.id = id
      conn.set("index:users:" + attrs.username, id, cb)
    }, function(ok, cb) {
      conn.hmset("users:" + self.id, attrs, cb)
    }
  ], callback)
}

PersistedUser.findById = function(id, callback) {
  store.getConnection().hgetall("users:" + id, function(error, attributes) {
    if (error) return callback(error)
    if (!attributes) return callback(new Error("User #" + id + " not found"))

    attributes.keyLength  = parseInt(attributes.keyLength, 10)
    attributes.workFactor = parseInt(attributes.workFactor, 10)
    callback(null, new PersistedUser(attributes, id))
  })
}

PersistedUser.findByUsername = function(username, callback) {
  var conn = store.getConnection(), self = this

  async.waterfall([
    function(cb) {
      conn.get("index:users:" + username, cb)
    }, function(id, cb) {
      self.findById(id, cb)
    }
  ], callback)
}

module.exports = PersistedUser
```

You'll notice that this class needs access to the database, which it gets by calling a method called `store.getConnection()`. In previous examples, we've given classes the ability to talk to the database by passing a connection into their constructor, but in this case we have 'class methods' — methods bound to the `PersistedUser` class itself rather than to any of its instances. We *could* generate copies of the class, just as we generated copies of our Express app, using a similar technique:

Figure 5.67. Generating ORM classes bound to a database connection

```
module.exports = function(redis) {
  var PersistedUser = function(attributes, id) {
    // ...
  }

  PersistedUser.prototype.save = function(callback) {
    if (this._id) return redis.hmset("users:" + this._id, this._attributes, callback)
    // ...
  }

  PersistedUser.findById = function(id, callback) {
    redis.hgetall("users:" + id, function(error, attributes) {
      // ...
    })
  }
}
```

This is certainly a viable approach, but you don't see it often in practice when using ORM frameworks. This is because it tends to be awkward to use, forcing the user to create copies of their model classes by manually passing around database configuration. It also prevents the use of things like `instanceof` and other reflection APIs; since there are multiple copies of the class in a running process, a module may receive a user object descended from a different class to the one the module has a reference to.

So, most frameworks go with having one copy of each model class, and those classes get their database connection from a singleton. In most frameworks, there are adapters for talking to all kinds of databases but in our home-grown example, we only need support for Redis, so let's make a little module that we can configure to hand out Redis connections. This will give us enough support to bind the app to a different database in testing and in production.

Figure 5.68. `node/orm/store.js`

```
var redis = require("redis")

module.exports = {
  configure: function(config) {
    this._conn = redis.createClient(config.port, config.host)
    this._conn.select(config.database || 0)
  },

  getConnection: function() {
    return this._conn
  }
}
```

Since we now have a singleton database connection manager, it makes more sense to configure the connection at the start of the test script, rather in any of the individual test suites. Here's the `test.js` file for these examples:

Figure 5.69. *`node/orm/test.js`*

```
var store = require("./store")
store.configure({host: "127.0.0.1", port: 6379, database: 15})

var JS = require("jstest")
require("./validated_user_spec")
require("./persisted_user_spec")
JS.Test.autorun()
```

The tests below are not exhaustive but cover some of the main cases we care about. We need to know we can save a new user and retrieve them via `findByUsername()`, we need to know a new user cannot clobber the `username` of an exsiting user, and we want to be able to update the details of existing users without breaking because their `username` already exists. We create a pre-existing user at the start of the tests for use in the latter two cases. We also need to set the work factor to reduce the execution time, and since we're talking to a database we need to remember to clear it out at the end of each test.

Figure 5.70. *`node/orm/persisted_user_spec.js`*

```
var JS            = require("jstest"),
    store         = require("./store"),
    PersistedUser = require("./persisted_user")

JS.Test.describe("PersistedUser", function() { with(this) {
  before(function(resume) { with(this) {
    this.alice = new PersistedUser({username: "alice", password: "fish", workFactor: 1})
    alice.save(resume)
  }})

  after(function(resume) { with(this) {
    store.getConnection().flushdb(resume)
  }})

  it("saves a user with a new username", function(resume) { with(this) {
    var bob = new PersistedUser({username: "bob", password: "chips", workFactor: 1})
    bob.save(function() {
      PersistedUser.findByUsername("bob", function(error, user) {
        resume(function() {
          assertNull( error )
          assertEqual( "bob", user.get("username") )
    })})})
  }})

  it("does not save a new user with an existing username", function(resume) { with(this) {
    var aliceCopy = new PersistedUser({username: "alice", password: "chips", workFactor: 1})
    aliceCopy.save(function(error) {
      resume(function(resume) {
        assertEqual( ["Username already exists"], error )
        PersistedUser.findByUsername("alice", function(error, user) {
          resume(function() {
            assert( user.checkPassword("fish") )
    })})})})
  }})

  it("saves updates to an existing user", function(resume) { with(this) {
    alice.set("email", "alice@example.com")
    alice.save(function() {
      PersistedUser.findByUsername("alice", function(error, user) {
        resume(function() {
          assertNull( error )
          assertEqual( "alice@example.com", user.get("email") )
    })})})
  }})
}})
```

The nested asynchronous steps in these tests suggest we might be better off using `JS.Test.asyncSteps()` to make the tests cleaner, but in this case I think writing the tests this way makes it clearer how the APIs actually look in real-world use. Using async steps tends to involve some indirection, where an API call is made in one step and then its results checked in another, and this can make it a little unclear how the class should be used in practice.

Notice how in the second test, we check that `aliceCopy` did not clobber the details of the first `alice` user by making sure that Alice's password is still `fish` rather than `chips`. We could also have used another simple property, such as an email address as in the final example; it doesn't make much of a difference what you use as long as you prove to yourself that none of the attributes in the duplicate record end up being saved to the original `alice` user.

5.4.4. Signing up and logging in

We've now built most of the infrastructure that we need to let people log in and start using the chat app: we have code to validate, store and authenticate users in our `PersistedUser` class in Figure 5.66, "`node/orm/persisted_user.js`", and a service for storing and retrieving chat messages in the `ChatService` class from Figure 5.46, "`node/services/lib/chat_service.js`". To let people actually use this functionality, we need to tie it all together into a website they can interact with in the browser.

So far, we've been using Express to build a JSON API, which handed us a user ID on registration, and accepted this user ID as a form parameter when posting messages. Now that we're moving to build an app that people will use via the browser, we need to introduce quite a bit more complexity. Instead of returning a user ID, the app will store the ID in the session when we log in, and send a session cookie to the browser. When posting messages, the session will be used to identify the user, rather than the client submitting a user ID explicitly. This means we need to enable session support and CSRF protection[28]. We're also going to need the app to serve static files, for when we eventually add some client-side JavaScript, and we need HTML templates; I've used Jade[29] for these examples.

There is a fair bit of code to step through here to understand the app, but it is mostly boilerplate whose job is to expose the functionality we've already built via HTTP and HTML. Let's walk through each component of the app in turn, before we move on to testing it.

The top-level application file is shown below. This file is really just configuration: all the logic lives in other modules, and this file simply wires everything together, setting up middleware and mapping URLs to internal module functions.

[28]CSRF stands for 'cross-site request forgery', a type of security hole wherein JavaScript running on a web page can send requests to another site that the user is logged in to, abusing the browser's cookie policies to impersonate the user and gain fraudulent access to their account. Any site that uses cookie-based authentication needs CSRF protection, and must use `POST` requests for any action that can make changes on a user's behalf. For more information see http://en.wikipedia.org/wiki/Cross-site_request_forgery.
[29]http://jade-lang.com/

Figure 5.71. `node/chatroom/app.js`

```
var connect     = require("connect"),
    http        = require("http"),
    path        = require("path"),
    store       = require("../orm/store"),
    ChatService = require("../services/lib/chat_service"),
    loadUser    = require("./middleware/load_user"),
    home        = require("./routes/home"),
    signup      = require("./routes/signup"),
    sessions    = require("./routes/sessions"),
    Chats       = require("./routes/chats")

var chats = new Chats(new ChatService(store.getConnection())),
    app   = require("express")()

app.set("views", path.resolve(__dirname, "views"))
app.set("view engine", "jade")

app.use(connect.static(path.resolve(__dirname, "public")))
app.use(connect.urlencoded())
app.use(connect.cookieParser())
app.use(connect.session({secret: "4855bca1b9a0ae6d5752a2ee7f2b162f66219b23"}))
app.use(connect.csrf())
app.use(loadUser)

app.get("/",                home)

app.get("/signup",          signup.get)
app.post("/signup",         signup.post)

app.get("/login",           sessions.getLogin)
app.post("/login",          sessions.postLogin)
app.get("/logout",          sessions.logout)

app.get("/chat",            chats.get.bind(chats))
app.post("/chat/:roomName", chats.post.bind(chats))
app.get("/chat/:roomName",  chats.poll.bind(chats))

module.exports = http.createServer(app)
```

The `loadUser()` middleware is a function that runs at the start of every request. It generates a CSRF token for the templates to insert into forms, and it checks the session for a user ID. If an ID is found, it loads the user's data and makes it available via `request.user` and `response.locals.user`. (`response.locals` is the object that stores variables that can be referenced by the templates when calling `response.render()`.)

Figure 5.72. `node/chatroom/middleware/load_user.js`

```
var User = require("../../orm/persisted_user")

module.exports = function(request, response, next) {
  response.locals.csrf = request.csrfToken()

  var userId = request.session.userId
  if (userId === undefined) return next()

  User.findById(userId, function(error, user) {
    if (error) delete request.session.userId
    request.user = response.locals.user = user
    next()
  })
}
```

The `home()` module is a very small request handler that picks a page to redirect to based on whether the user is logged in. If there is a logged-in user, we redirect to `/chat`, otherwise we go to `/signup`.

Figure 5.73. `node/chatroom/routes/home.js`

```
module.exports = function(request, response) {
  var url = request.user ? "/chat" : "/signup"
  response.redirect(url)
}
```

The `signup` routes are where things start to get interesting. `signup.get()` is very simple, it just renders the sign-up page. `signup.post()` is more complicated; it needs to extract the form data from the request, construct a new user with that data, and attempt to save it. `user.save()` will validate the user, including making sure their name is unique, and will yield an array of errors if anything went wrong. If we get any errors, we display the sign-up page again with the errors shown to the user, otherwise we stash the new user's ID in the session and send them off to the `/chat` page.

Note that the parameters are explicitly extracted from `request.body`, rather than creating a user by calling `new User(request.body)`. If we used the latter approach, then a malicious user could add extra parameters to the request and get them into the database — remember Redis has no enforced schema and any attributes we add to a user will end up in a Redis hash. To avoid getting junk unvalidated data in our database, we must be explicit about which parameters we actually want.

Figure 5.74. `node/chatroom/routes/signup.js`

```
var _    = require("underscore"),
    User = require("../../orm/persisted_user")

module.exports = {
  get: function(request, response) {
    if (request.user) return response.redirect("/chat")
    response.render("signup")
  },

  post: function(request, response) {
    var params = {username: request.body.username, password: request.body.password},
        user   = new User(params)

    user.save(function(errors) {
      if (errors) {
        _.extend(response.locals, {username: params.username, errors: errors})
        response.render("signup")
      } else {
        request.session.userId = user.id
        response.redirect("/chat")
      }
    })
  }
}
```

The template for the sign-up page is short and sweet: if there are any errors, we display them in a list. Beneath that, we display the sign-up form, including the `_csrf` parameter that must be included with all `POST` requests, and a text input for the `username` and `password`.

Figure 5.75. *node/chatroom/views/signup.jade*

```
extends layout

block content
  h2 Sign up

  if errors
    div.error
      p There was a problem with your registration
      ul
        each error in errors
          li= error

  form(method="post", action="/signup")
    input(type="hidden", name="_csrf", value=csrf)
    p
      label(for="username") Username
      input(type="text", name="username", id="username", value=username)
    p
      label(for="password") Password
      input(type="password", name="password", id="password")
    input(type="submit", value="Sign up")
```

The `sessions` module is similarly structured; its `getLogin()` method renders the login form, while `postLogin()` handles login attempts. Rather than creating a new user, it attempts to find an existing user by their username and check their password using the API we defined in Figure 5.61, "`node/passwords/adjustable_user.js`". If the username and password are valid, the user is logged in and redirected to `/chat`, otherwise we render the login form again with the error displayed. The `logout()` method deletes the user ID from the session and redirects to the homepage.

Figure 5.76. *node/chatroom/routes/sessions.js*

```
var User = require("../../orm/persisted_user")

module.exports = {
  getLogin: function(request, response) {
    if (request.user) return response.redirect("/chat")
    response.render("login")
  },

  postLogin: function(request, response) {
    var username = request.body.username,
        password = request.body.password

    User.findByUsername(username, function(error, user) {
      if (user && user.checkPassword(password)) {
        request.session.userId = user.id
        response.redirect("/chat")
      } else {
        response.locals.username = username
        response.locals.error = "Invalid username or password"
        response.render("login")
      }
    })
  },

  logout: function(request, response) {
    delete request.session.userId
    response.redirect("/")
  }
}
```

The login template is much like the sign-up template, only with slightly different error display markup:

Figure 5.77. `node/chatroom/views/login.jade`

```
extends layout

block content
  h2 Log in

  if error
    p.error= error

  form(method="post", action="/login")
    input(type="hidden", name="_csrf", value=csrf)
    p
      label(for="username") Username
      input(type="text", name="username", id="username", value=username)
    p
      label(for="password") Password
      input(type="password", name="password", id="password")
    input(type="submit", value="Log in")
```

We will leave discussion of the `/chat/:roomName` endpoints until Section 5.5, “Client-side JavaScript”.

This whole chapter, ever since Section 5.1.1, “Black-box server testing”, we’ve been testing server applications by starting them listening on a port and sending HTTP requests to them to see what came back. This is a valid approach because we’ve spent most of our time building APIs rather than websites, and users typically interact with an API using an HTTP client that can send arbitrary requests. But once we move on to building a website that people will access via a browser, this is no longer a useful strategy, because a browser is much more than just an HTTP client; it has additional features and imposes additional constraints.

First, browsers can’t make entirely arbitrary requests; for security reasons user code cannot set the `Cookie`, `Host`, `Origin`, `Referer` and other headers, and the same-origin policy[30] prevents client-side scripts from making arbitrary requests to any origin. Second, most browser requests are performed implicitly, either via clicking a link, submitting a form, following a redirect or requesting page assets. And third, browsers usually have features that are not typically enabled by default in vanilla HTTP clients: they store cookies received via `Set-Cookie` headers in responses and implicitly attach all applicable cookies to every request they send, they have caches that all implement some slightly different subset of the HTTP caching spec along with a set of opaque heuristics. And, they do a lot of things that go beyond HTTP concerns: parsing HTML, interpreting CSS and drawing pages visually, running JavaScript with DOM bindings in a security sandbox, storing data in `localStorage`, enforcing content security policies, and so much more.

To check a website actually works, you need to stick to only doing things a browser will let a real person do: following links and redirects using `GET`, resolving relative URLs along the way; and submitting forms, making sure you only submit parameters that are named by the form’s inputs, and making sure you send *all* the form’s inputs, including hidden fields. And, you need store and send cookies or nothing that relies on sessions is going to work.

Therefore, If you want to write tests in the way we’ve been doing so far, you’ve got your work cut out. As well as requesting your initial page, you need to parse the HTML that comes back, find elements using CSS or XPath, follow links (resolving relative URLs), change the values of form inputs, serialise forms and submit them correctly, follow redirects, store cookies you receive and attach the right ones to every request you make. It’s certainly possible to do all this using modules from npm, and we will discuss this in Section 5.4.6, “Simulating a browser”, but given the vast array of behaviour a real browser represents, it’s best to test using the real thing rather than building a replica with its own set of quirks.

[30]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Same_origin_policy_for_JavaScript

5.4.5. Testing via the browser

Browser-based testing has come on in leaps and bounds in recent years. While historically many browser testing tools have focussed on letting the user record a sequence of actions — mouse clicks, keboard input and so on — so they can be replayed later, modern tools focus on providing a scripting API for the browser so we can programmatically control them in a structured way. Rather than record that a mouse click should happen 400 pixels down the page and 200 pixels left, we can write code that finds elements with CSS selectors and uses those elements to interact with the page, much like using jQuery to control the DOM on the client side. There are a multitude of tools for doing this, and broadly speaking they fall into two camps: headless browsers, and automation tools for real browsers.

A ‘headless’ browser is just like the browser you use every day, except it has no GUI: it draws nothing on the screen. It still loads pages, parses the HTML, builds a model of the page, interprets CSS to understand how the page is layed out, runs JavaScript, and everything else, it just doesn’t display the result of this process visually. This means the browser does not need a desktop environment to run, and can therefore be invoked from the terminal or used on a continuous integration server. The most commonly used headless browser today is PhantomJS[31], which is based on WebKit[32], the browser engine used in Safari and Chrome. (Chrome technically now uses a fork of WebKit called Blink[33].) There is also SlimerJS[34], which is much like PhantomJS except it’s based on Gecko[35], the rendering engine used by Firefox. Both PhantomJS and SlimerJS provide a headless browser that you can control using JavaScript, that is, you use JavaScript to control the browser from the *outside*, rather than being limited to running JavaScript only within the context of web pages.

These tools are very useful on their own, allowing you to programmatically interact with the web, visiting pages, extracting data, following links and so on. In fact `jstest` integrates with both of them out of the box, so you can run your client-side tests from the terminal[36]. But there are also systems built on top of them that are more focussed on testing rather than general scripting; CasperJS[37] is a popular example of such.

The important thing about these tools is that they are *new browsers*; they are not simply Chrome or Firefox with the GUI stripped off. As such, they have their own quirks that you might run into from time to time; for example, PhantomJS 1.9 still uses a version of the WebSocket protocol dating from 2010[38] and that is no longer used in current browsers. They’re not perfect, but they are plenty good enough for running tests as a warning beacon; I usually don’t find much discrepancy between how my tests run on PhantomJS compared to how they run in real browsers.

The other main category of tools covers programs that automate real browsers that actual people use. The main contender here is Selenium WebDriver[39], which can control most of the popular desktop browsers. This means you can run your code automatically on the same platforms where it will run in production, so you can easily catch cross-browser issues. There are lots of libraries that provide an API for Selenium for various programming languages; in Node you can choose from WD.js[40], Nightwatch.js[41] and the recently released Testium[42].

[31]http://phantomjs.org/
[32]http://www.webkit.org/
[33]http://www.chromium.org/blink
[34]http://slimerjs.org/
[35]https://developer.mozilla.org/en-US/docs/Mozilla/Gecko
[36]http://jstest.jcoglan.com/phantom.html
[37]http://casperjs.org/
[38]https://tools.ietf.org/html/draft-hixie-thewebsocketprotocol-76
[39]http://docs.seleniumhq.org/projects/webdriver/
[40]https://www.npmjs.org/package/wd
[41]http://nightwatchjs.org/
[42]https://www.npmjs.org/package/testium

There is a third category of browser testing tools, which I'm going to refer to as 'simulators'. This includes tools like HtmlUnit[43], Envjs[44] and Zombie.js[45]; they are browser implementations designed to run in-process with the app that you're testing. I call them simulators, but really they are just more browser implementations; the only difference is they are usually complete reimplementations of the browser written in the application's host language (for example JavaScript) rather than binding to production rendering engines like WebKit. Being complete reimplementations, they of course have their own set of bugs distinct from any you'll see in real browsers, and speaking from personal experience I have never used one of these tools for more than an hour without finding show-stopping bugs on very common use cases. If a simulator won't run tests that work perfectly in real browsers, and leads you to spend more time adjusting your code to make the simulator happy rather than fixing real production bugs, I don't recommend basing your testing process on it.

For these examples I've gone with Testium, for no other reason than it took me the least effort to set up. Testium is a Node testing framework based on Mocha[46] that connects to Selenium and can drive both desktop browsers and PhantomJS. To get started, it requires a little config file called `.testiumrc` that tells it how to start your application. The `launch` flag just tells Testium that you want it to start your app automatically, and `command` tells it what to run in order to do that. The `port` option tells it which port your app will be running on so that the tests can make requests to it; Testium sets the `PORT` environment variable so your app can detect it.

Figure 5.78. *`node/chatroom/.testiumrc`*

```
launch = true

[app]
command = node main.js
port = 4180
```

The `command` refers to `main.js`, which we've not written yet. Figure 5.71, "`node/chatroom/app.js`" defines an application but does not configure or start it up: that's the job of this little boot script. It connects to the database based on some environment variables (we can leave these unset to get default values), and starts the app listening on the given port.

Figure 5.79. *`node/chatroom/main.js`*

```javascript
var store = require("../orm/store"),
    env   = process.env

store.configure({host: env.REDIS_HOST, port: env.REDIS_PORT, database: env.REDIS_DB})

var app = require("./app")
app.listen(env.PORT)
```

The nice thing about this script is you can use it in production as well as for testing. All it does is configure and start the app based on your environment variables; just set different environments in production and in testing and it will work just fine.

Let's take a look at a couple of testing scenarios. First, we want to check that sign-up works correctly: people should be directed to sign up if they attempt to access the app when not logged in, and the sign-up process itself should reject invalid input and accept valid data. We can check these requirements by telling the Testium browser to visit certain pages, telling it to fill in form fields and click on buttons, and then checking which URL we end up on and what text we can see. This is enough to let us check the validation is wired up properly and that users are sent to the right pages.

[43]http://htmlunit.sourceforge.net/
[44]http://www.envjs.com/
[45]http://zombie.labnotes.org/
[46]http://visionmedia.github.io/mocha/

As with any test that creates state, we need to clean up after ourselves. This sort of test creates state in two places: data is put into the database, and cookies end up in the browser, and both need cleaning at the end of each test. Our `afterEach()` block (so called because we're now using Mocha rather than `jstest`) takes care of all this so each test starts with a clean slate.

You'll notice one quirk in this file; we call `store.configure()` despite having already set up a database connection in Figure 5.79, "`node/chatroom/main.js`". This is because Testium runs `main.js` in a separate process from your tests, because otherwise its blocking API would stop Node's event loop and prevent your app from running. Since the tests are in a new Node process, they need to reinitialise the database connection for that process so we can empty the database. You may find it useful to put the database connection settings in a shared file if you need to copy this line into many different tests.

Figure 5.80. *`node/chatroom/testium/signup_spec.js`*

```
var injectBrowser = require("testium/mocha"),
    assert        = require("assert"),
    url           = require("url"),
    store         = require("../../orm/store"),
    env           = process.env

store.configure({host: env.REDIS_HOST, port: env.REDIS_PORT, database: env.REDIS_DB})

describe("Signing up", function() {
  before(injectBrowser())

  var pathname = function(browser) {
    return url.parse(browser.getUrl()).pathname
  }

  afterEach(function(resume) {
    this.browser.navigateTo("/")
    this.browser.clearCookies()
    store.getConnection().flushdb(resume)
  })

  it("redirects to the signup page when not logged in", function() {
    this.browser.navigateTo("/")
    assert.equal(pathname(this.browser), "/signup")
  })

  it("rejects invalid usernames", function() {
    this.browser.navigateTo("/signup")
    this.browser.type("[name=username]", "$%^&")
    this.browser.click("[type=submit]")
    assert.equal(pathname(this.browser), "/signup")
    this.browser.assert.elementHasText(".error li:first-child",
        "Usernames may only contain letters, numbers and underscores")
  })

  it("rejects blank passwords", function() {
    this.browser.navigateTo("/signup")
    this.browser.click("[type=submit]")
    assert.equal(pathname(this.browser), "/signup")
    this.browser.assert.elementHasText(".error li:last-child", "Password must not be blank")
  })

  it("accepts valid sign-ups", function() {
    this.browser.navigateTo("/signup")
    this.browser.type("[name=username]", "alice")
    this.browser.type("[name=password]", "something")
    this.browser.click("[type=submit]")
    assert.equal(pathname(this.browser), "/chat")
    this.browser.assert.elementHasText(".navigation li:first-child", "Logged in as alice")
  })
})
```

The tests for logging in are much the same, only now we need to perform some initial setup: to test login, we need a user to log in as. Since the tests are just Node scripts, we can load up our `User` class and create a user in the database during a `beforeEach()` block. The first test below checks that entering the right credentials redirects the user to the `/chat` page and displays their login status, while the second test checks that the wrong credentials leave the user on the login page with an error displayed to them.

Figure 5.81. *`node/chatroom/testium/login_spec.js`*

```
var injectBrowser = require("testium/mocha"),
    assert        = require("assert"),
    url           = require("url"),
    User          = require("../../orm/persisted_user"),
    store         = require("../../orm/store"),
    env           = process.env

store.configure({host: env.REDIS_HOST, port: env.REDIS_PORT, database: env.REDIS_DB})

describe("Logging in", function() {
  before(injectBrowser())

  var pathname = function(browser) {
    return url.parse(browser.getUrl()).pathname
  }

  beforeEach(function(resume) {
    new User({username: "bob", password: "sideshow", workFactor: 1}).save(resume)
  })

  afterEach(function(resume) {
    this.browser.navigateTo("/")
    this.browser.clearCookies()
    store.getConnection().flushdb(resume)
  })

  it("redirects to the chat page on successful login", function() {
    this.browser.navigateTo("/login")
    this.browser.type("[name=username]", "bob")
    this.browser.type("[name=password]", "sideshow")
    this.browser.click("[type=submit]")
    assert.equal(pathname(this.browser), "/chat")
    this.browser.assert.elementHasText(".navigation li:first-child", "Logged in as bob")
  })

  it("renders the form with an error on failed login", function() {
    this.browser.navigateTo("/login")
    this.browser.type("[name=username]", "bob")
    this.browser.type("[name=password]", "the-wrong-word")
    this.browser.click("[type=submit]")
    assert.equal(pathname(this.browser), "/login")
    this.browser.assert.elementHasText(".error", "Invalid username or password")
  })
})
```

You can run these tests by typing the folling commands from the `Code` directory[47]. Testium will run your tests in PhantomJS by default, but you can set other browsers via the `.testiumrc` file.

Figure 5.82. Running a Testium test suite

```
$ npm install webdriver-http-sync
$ cd node/chatroom
$ ../../node_modules/.bin/mocha testium/*.js
```

One thing you will undoubtedly notice about this process it that it's very slow. We've taken steps to make the usually slow login process faster by giving the user a `workFactor` of `1`, but even that only makes a small dent in the running time. We can only make this change for the login tests; in the sign-

[47]The `webdriver-http-sync` npm package is not included in this book's distribution since it includes native code that needs to be compiled locally.

up tests the new user is created in the application, not in the tests, and since the tests are in a separate process we cannot stub `User.WORK_FACTOR` or any other aspect of the app's internal workings. There are plenty of other sources of slowness; the entire application stack is being invoked, from the web server down to the database, which introduces both latency and complexity, and there's the overhead of calling Selenium and waiting for the browser to respond.

The real lesson here is not necessarily to make all these things go faster, but to not rely so much on full-stack integration tests. We know user validation works, we tested that in Figure 5.65, "`node/orm/validated_user_spec.js`" and Figure 5.70, "`node/orm/persisted_user_spec.js`". We don't need to test every failure case again, we just need to check the different code paths the Express frontend can take: invalid data should return the user to the sign-up page with a list of errors, and valid data should log the user in and take them to `/chat`. You should really reserve this sort of testing for broad-brush assurance that the stack is wired together correctly, and leave the details to unit tests that will run much faster.

Not only do unit tests run faster, they are more informative. An integration test can fail for all sorts of reasons: the database is down, there's a bug in your validation or persistence logic, the HTTP layer is misconfigured, the templates are wrong, there was a problem with the automation tools or the browser itself. When tests like this fail they are usually very opaque about the reason for the failure, and you waste a lot of time tracking down the cause. Unit tests, since they invoke far less code by definition, are usually much more helpful for finding the source of bugs; if you get a failing integration test, find the cause and encode the bug you found as a failure unit test for the relevant component. This way you spend far less time both running your tests and figuring out why they are broken.

5.4.6. Simulating a browser

A frustrating limitation of browser testing tools is that most run your app and your tests in separate processes, completely preventing the possibility of mocking and stubbing internal APIs. We've seen one example of that with the deliberate slowness of password hashing during sign-up, but often it's useful to stub out internal components so you can unit-test the web app, and make sure it's making the right decisions at the HTTP level. Just as we did in Figure 5.53, "`node/services/spec/register_spec.js`", where we said, "when I send the request `POST /users`, it should call `userService.register()` with the given username", it's often useful to perform mock-based testing of a website, checking what gets displayed given a canned database response, or checking the right model commands result from submitting a form.

Fortunately, for apps that run entirely on the server with no client-side JavaScript — like our sign-up and login pages — this is relatively simple. Although building a full-on browser simulator is prohibitively expensive and error-prone, building a user agent that knows how to load pages, parse HTML, follow links, submit forms, follow redirects and handle cookies is enough to mechanically interact with many sites. Below is a simple `Browser` class that, while massively incomplete, does a good enough job at these tasks to let us test our app in the same process where it's running.

It uses a couple of npm modules to do most of the hard work. `request`, which we've been using for many of our tests, is used to make HTTP requests, and we're using its cookie-handling functionality: `request.jar()` creates a cookie jar that we pass along with all requests. `request` does have the ability to follow redirects, but we're doing that ourselves here because a browser must remember its current URL so it can resolve relative links. So, we need to record the `Location` header of each response rather than simply receiving the final one. The `cheerio` module knows how to parse HTML, and provides a subset of the jQuery API that we can use to manipulate and serialise forms before submitting them.

The `Browser` also passes along a few of the common request headers used by browsers, including `Accept`, `User-Agent` and `Referer`. You might need to add support for other headers if you app relies on them.

Figure 5.83. `node/chatroom/spec/browser.js`

```
var cheerio = require("cheerio"),
    path    = require("path"),
    request = require("request"),
    url     = require("url"),
    _       = require("underscore")

var Browser = function(userAgent, origin) {
  this.location = url.parse(origin)
  this._cookies = request.jar()
  this._headers = {"Accept": "text/html", "User-Agent": userAgent}
}

Browser.prototype.visit = function(uri, callback) {
  this._request("get", uri, {}, callback)
}

Browser.prototype.clickLink = function(link, callback) {
  this._request("get", link.attr("href"), {}, callback)
}

Browser.prototype.submitForm = function(form, callback) {
  var method = (form.attr("method") || "get").toLowerCase(),
      action = form.attr("action") || "",
      data   = {},
      self   = this

  form.find("input, select, textarea").each(function(i, input) {
    input = self.dom(input)
    var name = input.attr("name"), type = input.attr("type") || ""

    if (name && (!type.match(/checkbox|radio/) || input.attr("checked") !== undefined)) {
      data[name] = input.val() || ""
    }
  })
  this._request(method, action, data, callback)
}

Browser.prototype._request = function(method, uri, params, callback) {
  var headers = _.extend({"Referer": this.location.href}, this._headers),
      options = {followRedirect: false, headers: headers, jar: this._cookies},
      self    = this

  options[method === "post" ? "form" : "qs"] = params
  uri = url.resolve(this.location.href, uri)
  this.location = url.parse(uri)

  request[method](uri, options, function(error, response, body) {
    self.status  = response.statusCode
    self.headers = response.headers
    self.body    = body

    if (self.status >= 300 && self.status < 400 && self.headers.location) {
      self._request("get", self.headers.location, {}, callback)
    } else {
      self.dom = cheerio.load(self.body)
      callback()
    }
  })
}

module.exports = Browser
```

Based on this `Browser` class, we can create a set of async steps for use in tests. We’re using `JS.Test.asyncSteps()` here because much of the work is asynchronous, and the tests will be hard to read otherwise. There’s the steps for starting and stopping the server, and cleaning the database, as we’ve seen before, and the remaining steps are simple wrappers around the `Browser` API.

Figure 5.84. `node/chatroom/spec/browser_steps.js`

```
var JS      = require("jstest"),
    Browser = require("./browser"),
    store   = require("../../orm/store")

module.exports = JS.Test.asyncSteps({
  startServer: function(server, callback) {
    this.server = server
    this.server.listen(0, callback)

    this.port   = this.server.address().port
    this.origin = "http://localhost:" + this.port
  },

  stopServer: function(callback) {
    this.server.close(callback)
  },

  createBrowser: function(callback) {
    this.browser = new Browser("Mozilla/5.0 (KHTML, like Gecko)", this.origin)
    callback()
  },

  visit: function(pathname, callback) {
    this.browser.visit(pathname, callback)
  },

  clickLink: function(selector, callback) {
    var link = this.browser.dom(selector)
    this.browser.clickLink(link, callback)
  },

  fillIn: function(selector, text, callback) {
    this.browser.dom(selector).val(text)
    callback()
  },

  clickButton: function(selector, callback) {
    var form = this.browser.dom(selector).parents("form")
    this.browser.submitForm(form, callback)
  },

  checkPath: function(pathname, callback) {
    this.assertEqual(pathname, this.browser.location.pathname)
    callback()
  },

  checkText: function(selector, text, callback) {
    var node = this.browser.dom(selector)
    this.assertEqual(text, node.text())
    callback()
  },

  cleanDatabase: function(callback) {
    store.getConnection().flushdb(callback)
  }
})
```

These steps let us write a replica of Figure 5.80, “`node/chatroom/testium/signup_spec.js`”, but instead of a Selenium-driven desktop browser, we use our `Browser` class that runs within the Node process. The only differences are that the function names are slightly different, we’re starting the app up from within the tests rather than in a start-up script, and instead of deleting the cookies at the end of each test we make a fresh browser instance with a new cookie jar at the start of each test.

Figure 5.85. *`node/chatroom/spec/signup_spec.js`*

```
var JS    = require("jstest"),
    app   = require("../app"),
    steps = require("./browser_steps")

JS.Test.describe("Signing up", function() { with(this) {
  include(steps)

  before(function() { with(this) {
    startServer(app)
    createBrowser()
  }})

  after(function() { with(this) {
    stopServer()
    cleanDatabase()
  }})

  it("redirects to the sign-up page when not logged in", function() { with(this) {
    visit("/")
    checkPath("/signup")
  }})

  it("rejects invalid usernames", function() { with(this) {
    visit("/signup")
    fillIn("[name=username]", "$%^&")
    clickButton("[type=submit]")
    checkText(".error li:first-child",
        "Usernames may only contain letters, numbers and underscores")
  }})

  it("rejects blank passwords", function() { with(this) {
    visit("/signup")
    clickButton("[type=submit]")
    checkPath("/signup")
    checkText(".error li:last-child", "Password must not be blank")
  }})

  it("accepts valid sign-ups", function() { with(this) {
    visit("/signup")
    fillIn("[name=username]", "alice")
    fillIn("[name=password]", "something")
    clickButton("[type=submit]")
    checkPath("/chat")
    checkText(".navigation li:first-child", "Logged in as alice")
  }})
}})
```

Reproducing our Testium tests is one thing, but the main advantage of having the app and the tests running in the same process is that we can stub out the app’s internals. In Figure 5.81, “`node/chatroom/testium/login_spec.js`”, we tested that login worked by first putting a user record in the database, and then entering correct and incorrect credentials in the login form to check it worked. The central idea under test here is: if the user’s password is correct, then they should be logged in and taken to `/chat`, and if not they should remain on the login page. The details of what ‘valid password’ actually means can be ignored; from the point of view of the Express app, if `user.checkPassword()` returns `true` then the user is logged in. So, rather than introduce the complexity of having a database and a slow authentication procedure, we can stub `user.checkPassword()` to return different things and check that this makes the application do the right thing.

To achieve this, we create a `User` instance but don't save it to the database. Instead, we stub `User.findById()` and `User.findByUsername()` to yield this user, so that the `sessions.postLogin()` and `loadUser()` functions are handed this user. We need to give the user an ID in this case, so that the ID ends up in the session and `loadUser()` is made to load the user rather than skipping this logic.

Figure 5.86. *`node/chatroom/spec/login_spec.js`*

```
var JS    = require("jstest"),
    app   = require("../app"),
    User  = require("../../orm/persisted_user"),
    steps = require("./browser_steps")

JS.Test.describe("Logging in", function() { with(this) {
  include(steps)

  before(function() { with(this) {
    this.user = new User({username: "alice"}, 61)
    stub(User, "findById").given(61).yields([null, user])
    stub(User, "findByUsername").given("alice").yields([null, user])

    startServer(app)
    createBrowser()
  }})

  after(function() { with(this) {
    stopServer()
    cleanDatabase()
  }})

  describe("when the password is valid", function() { with(this) {
    before(function() { with(this) {
      expect(user, "checkPassword").given("california").returns(true)
    }})

    it("redirects to the chat page", function() { with(this) {
      visit("/login")
      fillIn("[name=username]", "alice")
      fillIn("[name=password]", "california")
      clickButton("[type=submit]")
      checkPath("/chat")
      checkText(".navigation li:first-child", "Logged in as alice")
    }})
  }})

  describe("when the password is not valid", function() { with(this) {
    before(function() { with(this) {
      expect(user, "checkPassword").given("new york").returns(false)
    }})

    it("displays the login page with an error", function() { with(this) {
      visit("/login")
      fillIn("[name=username]", "alice")
      fillIn("[name=password]", "new york")
      clickButton("[type=submit]")
      checkPath("/login")
      checkText(".error", "Invalid username or password")
    }})
  }})
}})
```

The use of `expect(user, "checkPassword")` here serves a dual purpose. The `given()` modifier lets us check that when a password is submitted via the form, that the app forwards that password to the user model for verification. The `returns()` modifier creates a canned response; we make `user.checkPassword()` return `true` and make sure this results in a successful login, then we make it return `false` and check that this results in an error displayed to the user.

5.5. Client-side JavaScript

With our authentication system out of the way, we now come to the main page of the app: the chat page. We've already built validation and persistence logic for posting and retrieving messages, and now we need a UI that makes use of this API. The `GET /chat`, `GET /chat/:roomName` and `POST /chat/:roomName` endpoints route to the `Chats` module shown below.

The `Chats` class has three methods: `get()`, `post()` and `poll()`. `get()` checks that the user is logged in, and if so it renders the `chat` template, which contains the HTML for the user interface. You'll notice that the `get()` method sets a template variable called `response.locals.port`; this is because later on, when we use WebSocket to poll for messages, the client-side code will need to know which port the application is running on.

The `post()` and `poll()` methods expose the validation and persistence logic we've already built via JSON as before, but with a couple of tweaks. First, the `post()` method makes sure there is a logged-in user making the request; only people who are logged in are allowed to access the chat service. Second, whereas our previous version of the chat API, Figure 5.49, "`node/services/app.js`" got the user ID from `request.body.userId`, this application gets the user ID from the session instead. This is a small change at the HTTP layer, and doesn't affect the underling `ChatService` at all: `ChatService.postMessage()` takes the user ID as an argument, and doesn't care where that ID came from. Since `ChatService` is not coupled to HTTP, we can easily make changes at the HTTP layer; this will become important when we add WebSocket support. The `poll()` method remains unauthenticated: anyone is allowed to poll the messages for a room, and the login mechanism is only there to verify the identity of usernames.

It's worth noting that whereas the `signup` and `sessions` routes loaded the `User` class themselves using `require()` — see Figure 5.76, "`node/chatroom/routes/sessions.js`" — the `Chats` class uses dependency injection, taking a reference to a `ChatService` in its constructor rather than loading it with `require()`. Like the `port` variable, this construction will also be useful when adding WebSocket support, because we will need multiple app components to reference the same `ChatService` instance.

Figure 5.87. `node/chatroom/routes/chats.js`

```
var validation = require("../../services/lib/validation")

var Chats = function(service) {
  this._service = service
}

Chats.prototype.get = function(request, response) {
  if (!request.user) return response.redirect("/signup")
  response.locals.port = Chats.PORT || 0
  response.render("chat")
}

Chats.prototype.post = function(request, response) {
  if (!request.user) {
    return response.json(401, {errors: ["You must be logged in"]})
  }

  var roomName = request.params.roomName,
      userId   = request.session.userId,
      message  = request.body.message,
      now      = Date.now()

  var errors = validation.checkMessage({roomName: roomName, message: message})
  if (errors) return response.json(409, {errors: errors})

  this._service.postMessage(roomName, userId, message, now, function(error, messageData) {
    render(response, error, 201, messageData)
  })
}

Chats.prototype.poll = function(request, response) {
  var roomName = request.params.roomName,
      since    = parseInt(request.query.since, 10),
      now      = Date.now()

  var errors = validation.checkPoll({roomName: roomName, since: since})
  if (errors) return response.json(409, {errors: errors})

  this._service.pollMessages(roomName, since, now, function(error, messages) {
    render(response, error, 200, {messages: messages})
  })
}

var render = function(response, error, status, data) {
  if (error) {
    response.json(500, {errors: [error.message]})
  } else {
    response.json(status, data)
  }
}

module.exports = Chats
```

Now we come to the front-end code itself. The `GET /chat` endpoint uses the following template to render the page. It contains four main elements: a form for setting the name of the room you want to join, a paragraph displaying which room you're currently in, a form for posting new messages, and a list that will display all the messages posted to the room while you're present.

Although this page won't do much without JavaScript running on the client, starting from a foundation of meaningful HTML rather than rendering everything with JavaScript will make it easier for us to use our browser simulator to test this page, which will come in handy later.

Figure 5.88. `node/chatroom/views/chat.jade`

```
extends layout

block content
  h2 Chat

  form(method="post", action="/chat", class="room")
    input(type="hidden", name="_csrf", value=csrf)
    label(for="room") Room
    input(type="text", name="room", id="room")
    input(type="submit", value="Join")

  p Current room:
    span.current-room

  form(method="post", action="chat", class="message")
    input(type="hidden", name="_csrf", value=csrf)
    label(for="message") Message
    input(type="text", name="message", id="message")
    input(type="submit", value="Post")

  ul.messages

  script.
    var SERVER_PORT = #{ port }

  script(src="/jquery.js")
  script(src="/chat.js")
```

The final piece of the puzzle is the client-side JavaScript that will make the interface work. This module, shown below, needs to let the user set their current room, show them which room they are in, let them post new messages and request and display new messages from everyone in the room.

It does this by intercepting `"submit"` events on both the forms. For the room form, it just stores the name of the room the user entered, and setting the ‘last-poll’ time to a few minutes ago, so that the next poll request will pick up the room’s recent history. When the user joins a new room, the message list is emptied. When the user submits a new message, we send the message to the server by making a request to `POST /chat/:roomName`, which will invoke the `Chats.post()` and `ChatService.postMessage()` methods on the server side.

Finally, we set up a polling loop, using `setInterval()` to check for new messages once a second. This uses the `GET /chat/:roomName` endpoint to fetch messages since the last-poll time, and adds any messages it gets back to the page. When we display new messages, we set the last-poll time to one millisecond after the final message’s timestamp, so that next time we poll we only ask for messages posted since the last one we saw.

Figure 5.89. *`node/chatroom/public/chat.js`*

```
var messageList = $("ul.messages"),
    currentRoom = null,
    lastPoll    = null

$("form.room").on("submit", function(event) {
  event.preventDefault()

  currentRoom = $(this).find("[name=room]").val()
  lastPoll    = Date.now() - 5 * 60 * 1000

  $(".current-room").text(currentRoom)
  messageList.empty()
})

$("form.message").on("submit", function(event) {
  event.preventDefault()
  if (!currentRoom) return

  var form = $(this)

  $.post("/chat/" + currentRoom, form.serialize(), function() {
    form.find("[name=message]").val("")
  })
})

setInterval(function() {
  if (!currentRoom) return

  $.get("/chat/" + currentRoom, {since: lastPoll}, function(response) {
    $.each(response.messages, function(i, chat) {
      var username = chat.username, text = chat.message
      messageList.prepend("<li><b>" + username + ":</b> " + text + "</li>")
      lastPoll = chat.timestamp + 1
    })
  })
}, 1000)
```

With the app complete, you can have a go at using it: run the following command from the `Code/node/chatroom` directory to start the server:

Figure 5.90. Running the chatroom app

```
$ REDIS_DB=15 PORT=3000 node main.js
```

If you visit `http://localhost:3000/` in your browser, you will be prompted to sign up and start chatting. If you open a couple of different web browsers, you can log in as two different users and start a conversation.

To test this, we need to use a browser that can run JavaScript; our simulator won't cut it, so we'll use Testium again. Remember that we should limit full-stack tests to checking one or two illustrative examples, rather than going into exhaustive detail. We can write a test that reproduces what you just did in the browser: logging in, setting your room, posting a message and seeing it show up in the list.

A Testium test for this is shown below. We begin by creating a user in the database that we can log in as, setting their `workFactor` to `1` to speed up the login process a little. At the end of the test, as before, we must clear the browser's cookie jar and empty the database. The test itself goes through the motions of logging in, choosing a room, and posting a message, by telling Testium to type values into form fields and pressing submit buttons; the first submit will take the user from `/login` to `/chat`, but the latter two are intercepted by JavaScript and the user remains on the same page.

Figure 5.91. `node/chatroom/testium/chat_spec.js`

```
var injectBrowser = require("testium/mocha"),
    User          = require("../../orm/persisted_user"),
    store         = require("../../orm/store"),
    env           = process.env

store.configure({host: env.REDIS_HOST, port: env.REDIS_PORT, database: env.REDIS_DB})

describe("Chatting", function() {
  before(injectBrowser())

  beforeEach(function(resume) {
    new User({username: "bob", password: "sideshow", workFactor: 1}).save(resume)
  })

  afterEach(function(resume) {
    this.browser.navigateTo("/")
    this.browser.clearCookies()
    store.getConnection().flushdb(resume)
  })

  it("displays messages the user posts", function() {
    this.browser.navigateTo("/login")
    this.browser.type("[name=username]", "bob")
    this.browser.type("[name=password]", "sideshow")
    this.browser.click("[type=submit]")

    this.browser.type("[name=room]", "attic")
    this.browser.click("form.room [type=submit]")

    this.browser.type("[name=message]", "Hi, everyone!")
    this.browser.click("form.message [type=submit]")

    this.browser.waitForElementVisible(".messages li")
    this.browser.assert.elementHasText(".messages li", "bob: Hi, everyone!")
  })
})
```

As before, this test runs very slowly; on my machine it takes over 2 seconds to run. If we test every code path like this we'll end up with a very slow test suite, so make sure you use this technique sparingly and cover as much behaviour as possible with fast unit tests. If we wrap the client-side JavaScript up into a class, and stub out the Ajax calls, we can use the techniques discussed in Section 4.3, "Talking to the server" to probe what the client does given different server responses, or if the server returns an error. We already have unit tests for the server's JSON API, the underlying validation and persistence service, and so adding unit tests for the client's interaction with that API will cover most of the code paths in the app.

There is one minor apparent flaw in the above test: it makes sure that a user is shown messages they posted themselves, but doesn't check they are shown messages from other people. Unfortunately this isn't possible with Selenium, because you're driving one browser with a single cookie jar, so we can't write a test where two people are logged in at the same time.

Fortunately, we needn't worry about this too much. We know from the tests in Figure 5.48, "`node/services/spec/chat_service_poll_messages_spec.js`" that the `GET /chat/:roomName` endpoint will return all the messages in a room, whoever posted them and whoever is making the polling request. Since we know that, all that remains to check is that when the client makes a polling request and receives these messages, it displays them in the correct way, and we can cover that with client-side unit tests. This is an example of a case where you can avoid fully replicating real-world usage if you identify what must be true for the service to work correctly, and comprehensively unit-test those requirements.

5.6. Real-time updates

In the previous section, we used a periodic polling request to get new chat messages. This takes advantage of the existing server-side infrastructure we'd built up, but it's hardly the most appropriate technique for handling a chat server. To make the service feel snappy, we'd rather push new messages to the client as soon as they arrive, rather than waiting a second or two for them to show up.

We can accomplish this using WebSocket; if we open a socket to the server, we can tell the server which room we want updates for by sending the name of the room over the socket, and the server can send us messages by pushing their JSON representation back up the wire. Below is an adaptation of our previous client-side code, with the polling logic replaced with a WebSocket. When the user chooses a new room, the room name is sent to the server, and when the socket receives a new message, it's added to the page. We no longer need to maintain a last-poll time, because the server will push messages to us instead of us having to ask for them.

Figure 5.92. `node/websockets/public/chat.js`

```
var messageList = $("ul.messages"),
    currentRoom = null

$("form.room").on("submit", function(event) {
  event.preventDefault()

  currentRoom = $(this).find("[name=room]").val()

  $(".current-room").text(currentRoom)
  messageList.empty()
  ws.send(currentRoom)
})

$("form.message").on("submit", function(event) {
  event.preventDefault()
  if (!currentRoom) return

  var form = $(this)

  $.post("/chat/" + currentRoom, form.serialize(), function() {
    form.find("[name=message]").val("")
  })
})

var port = SERVER_PORT || location.port,
    host = location.hostname + (port ? ":" + port : ""),
    ws   = new WebSocket("ws://" + host + "/realtime")

ws.onmessage = function(event) {
  var chat     = JSON.parse(event.data),
      username = chat.username,
      text     = chat.message

  messageList.prepend("<li><b>" + username + ":</b> " + text + "</li>")
}
```

Note how this module uses the `SERVER_PORT` variable that we set up in Figure 5.88, "`node/chatroom/views/chat.jade`". You might think we should just use `location.host` to get the domain and port that the page is running on, and use that to construct the WebSocket URL. However, Testium makes the browser access your app via a proxy; say you've booted your server on `localhost:57974`, Testium makes the browser go to `localhost:4445`, a proxy server that forwards requests onto `localhost:57974`. During my testing, I found that this proxy breaks WebSocket connections, and so rather than connecting to `location.host` and going via the proxy, we need to connect directly to the origin server. This is why the client-side code needs to know the server port.

We need to do some work on the server to support the client's WebSocket connection. This app uses a very slightly modified version of Figure 5.71, "`node/chatroom/app.js`" for its Express frontend; it mostly reuses components from previous examples, with two exceptions. First, it uses a modified version of `ChatService`, which we'll visit shortly, and it has a new class called `Sockets`, which will be responsible for handling WebSocket connections and routing messages to them. Notice how the `ChatService` instance is passed in to both `new Chats(service)` and `new Sockets(service)`; giving both these components a reference to the same service object is going to allow posted messages to be routed to the connected sockets.

Figure 5.93. `node/websockets/app.js`

```
var connect     = require("connect"),
    http        = require("http"),
    path        = require("path"),
    store       = require("../orm/store"),
    loadUser    = require("../chatroom/middleware/load_user"),
    home        = require("../chatroom/routes/home"),
    signup      = require("../chatroom/routes/signup"),
    sessions    = require("../chatroom/routes/sessions"),
    Chats       = require("../chatroom/routes/chats"),
    ChatService = require("./lib/evented_chat_service"),
    Sockets     = require("./lib/sockets")

var service = new ChatService(store.getConnection()),
    chats   = new Chats(service),
    sockets = new Sockets(service),
    app     = require("express")()

app.set("views", path.resolve(__dirname, "..", "chatroom", "views"))
app.set("view engine", "jade")

app.use(connect.static(path.resolve(__dirname, "public")))
app.use(connect.urlencoded())
app.use(connect.cookieParser())
app.use(connect.session({secret: "4855bca1b9a0ae6d5752a2ee7f2b162f66219b23"}))
app.use(connect.csrf())
app.use(loadUser)

app.get("/",                 home)

app.get("/signup",           signup.get)
app.post("/signup",          signup.post)

app.get("/login",            sessions.getLogin)
app.post("/login",           sessions.postLogin)
app.get("/logout",           sessions.logout)

app.get("/chat",             chats.get.bind(chats))
app.post("/chat/:roomName", chats.post.bind(chats))
app.get("/chat/:roomName",  chats.poll.bind(chats))

var server = http.createServer(app)
server.on("upgrade", sockets.acceptSocket.bind(sockets))

module.exports = server
```

The only addition to the set of route bindings is that all `"upgrade"` requests are routed to `Sockets.acceptSocket()`. Node treats 'upgrade' requests (those that, like WebSocket connections, have an `Upgrade` HTTP header) separately from 'normal' requests, and upgrades don't go through the

Express stack[48]. We must add our own event listener directly to the HTTP server to handle socket connections.

We need to somehow cause newly posted messages to be routed to any sockets that are in that message's room. We know the client will send a message identifying which room it wants messages for, and expects to receive JSON representations of messages for that room. So, we need to get the `ChatService.postMesssage()` method to trigger this behaviour when it accepts a new message. However, we don't want to couple a service that deals with persistence to the HTTP layer, so how can we go about this? Well, recall that both the `Chats` request handler and the `Sockets` socket handler both have a reference to the same `ChatService` instance. `Chats` tells `ChatService` to save a message when it receives a `POST` request, and the `Sockets` class wants to be notified when this happens. We can have `ChatService` emit an event when it receives a new message; the `Sockets` class can listen to this event, and route the new message to whichever sockets need to know about it.

Below is a subclass of `ChatService`, called `EventedChatService`, that accomplishes this. The code is quite ugly, but it is mostly boilerplate. First, we create a constructor called `EventedChatService` that takes a database connection as input, just like `ChatService`, and applies both the `ChatService` and `EventEmitter` constructors to the new object. We then use `util.inherits()` to subclass from `ChatService`, and `_.extend()` to add all of the methods from `EventEmitter` to our new class. (JavaScript uses single inheritance, so we can't use `util.inherits()` for both 'parents'. Instead we genuinely inherit from one parent and copy the methods from the other.) We then store a reference to the original `postMessage()` function, before assigning a new version to `EventedChatService.prototype`. This version is just a wrapper around the original; it forwards the arguments onto the original `postMessage()` method, and if that method does not yield an error, then we emit a `"message"` event with the name of the room and the full message data for the new chat message.

Figure 5.94. `node/websockets/lib/evented_chat_service.js`

```
var ChatService = require("../../services/lib/chat_service"),
    events      = require("events"),
    util        = require("util"),
    _           = require("underscore")

var EventedChatService = function(connection) {
  ChatService.call(this, connection)
  events.EventEmitter.call(this)
}
util.inherits(EventedChatService, ChatService)
_.extend(EventedChatService.prototype, events.EventEmitter.prototype)

var postMessage = ChatService.prototype.postMessage

EventedChatService.prototype.postMessage = function(room, userId, message, now, callback) {
  var self = this

  postMessage.call(this, room, userId, message, now, function(error, messageData) {
    if (error) return callback(error)

    self.getMessage(messageData.id, function(error, message) {
      self.emit("message", {room: room, chat: message})
      callback(null, messageData)
    })
  })
}

module.exports = EventedChatService
```

[48]This means they don't go through the session and authentication layers in the Express app; authenticating WebSocket connections is a complex topic beyond of the scope of this book and we will not use WebSocket for posting messages.

We use `getMessage()` to fetch the full message data (`messageData` doesn't contain the `username`, for example) so that the payload we send over the socket will be the same as the message format exposed by the `GET /chat/:roomName` endpoint. This means the client-side code that translates message data into elements on the page does not need to change.

To test this, we just need to ensure that posting a message to this modified chat service results in it emitting an event; we can use mocking to assert that the service should call its `emit()` method with the event name `"message"` and the room name and message data we expect. Getting the full message data involves looking up the person's username; `ChatService` doesn't rely on the `User` API but directly accesses user records in the database itself, so we need to put a user in the database before the test (and, as always, clean the database afterwards).

Figure 5.95. `node/websockets/spec/evented_chat_service_spec.js`

```
var JS          = require("jstest"),
    store       = require("../../orm/store"),
    User        = require("../../orm/persisted_user"),
    ChatService = require("../lib/evented_chat_service")

JS.Test.describe("EventedChatService", function() { with(this) {
  before(function(resume) { with(this) {
    this.service = new ChatService(store.getConnection())
    this.user = new User({username: "james", password: "x", workFactor: 1})
    user.save(resume)
  }})

  after(function(resume) { with(this) {
    store.getConnection().flushdb(resume)
  }})

  it("emits an event after publishing a message", function(resume) { with(this) {
    var timestamp = Date.now()

    expect(service, "emit").given("message", {
      room: "garage",
      chat: {id: 1, message: "knock knock", timestamp: timestamp, username: "james"}
    })
    service.postMessage("garage", user.id, "knock knock", timestamp, resume)
  }})
}})
```

Once we know that `EventedChatService` will emit events for new messages, we can build the `Sockets` class that will listen to these events and forward them onto connected sockets. To handle WebSocket connections we'll use the faye-websocket[49] module, which provides the same WebSocket API you get in the browser, but on the server side of the connection.

The `Sockets` class below takes a `ChatService` instance in its constructor, and registers an event listener to the `"message"` event we implemented above. The `acceptSocket()` method takes the arguments yielded by the server's `"upgrade"` event and accepts the WebSocket connection, listening for messages from the client. When a message containing a room name is received, we place the socket in an index of sockets subscribed to that room. The socket is given a random ID, and the subscription index stores sockets by their ID so they can be easily removed again. The `routeMessage()` method looks up the index for the message's room name, and sends the message to all the sockets in that index in JSON format.

[49]https://www.npmjs.org/package/faye-websocket

Figure 5.96. `node/websockets/lib/sockets.js`

```
var WebSocket = require("faye-websocket"),
    crypto    = require("crypto")

var Sockets = function(service) {
  this._service = service
  this._rooms   = {}

  this._service.on("message", this.routeMessage.bind(this))
}

Sockets.prototype.acceptSocket = function(request, socket, body) {
  var ws   = new WebSocket(request, socket, body),
      id   = crypto.randomBytes(20).toString("hex"),
      room = null,
      self = this

  ws.onmessage = function(event) {
    if (room) delete self._rooms[room][id]
    room = event.data
    self._rooms[room] = self._rooms[room] || {}
    self._rooms[room][id] = ws
  }

  ws.onclose = function() {
    if (room) delete self._rooms[room][id]
    ws = null
  }
}

Sockets.prototype.routeMessage = function(message) {
  var room = this._rooms[message.room] || {}

  for (var id in room) {
    room[id].send(JSON.stringify(message.chat))
  }
}

module.exports = Sockets
```

It is probably not obvious at first how to test this module. It logically sits at the same level in the stack as the Express request handlers; it deals with exposing some domain logic (chat message notifications) via a networked interface, in this case WebSocket, which is an extension to HTTP. All our previous HTTP testing efforts have started with spinning up a server, possibly faking out some internal APIs, then interacting with the server by issuing genuine HTTP requests, either via a simple HTTP client or via a browser of some description. We can apply that approach again here; we can create a `Sockets` instance by passing in an `EventEmitter` rather than a real service object, since the only aspect of the service object that `Sockets` actually relies upon is its capacity to emit events. We don't need to spin up the whole Express app either; we can create a standalone HTTP server and bind a `Sockets` instance to its `"upgrade"` event.

5.6.1. Testing with real connections

With this server running, we can open a WebSocket connection to it; the faye-websocket module that we're using to handle server-side connections also provides a client that behaves like the `WebSocket` class you'll find in the browser. We can then replicate what we expect our client-side code to do: use `ws.send(roomName)` to subscribe to a room, and listen for messages from the server. When the `Sockets` class picks up a `"message"` event from the underlying service, that event should be relayed to the client that's subscribed to that room's notifications.

Here's a test that carries these steps out. It is rather verbose: there a lot of moving parts to orchestrate. We must create an `EventEmitter` to act as our service object, create a `Sockets` instance from that, call `http.createServer()` and bind an `"upgrade"` listener to it, start the server, open a WebSocket connection to it and register a message listener, and that's just the setup phase. In the test itself, we use `ws.send("basement")` to subscribe to a room, make the service emit a `"message"` event, then check the WebSocket received a message containing the event data. We must also check that if the client is not subscribed to any room, then it receives no notifications from the server.

Figure 5.97. *`node/websockets/spec/sockets_server_spec.js`*

```
var JS        = require("jstest"),
    events    = require("events"),
    http      = require("http"),
    WebSocket = require("faye-websocket").Client,
    Sockets   = require("../lib/sockets")

JS.Test.describe("Socket server", function() { with(this) {
  before(function(resume) { with(this) {
    this.service = new events.EventEmitter()
    this.sockets = new Sockets(service)
    this.ws      = null
    this.message = null
    this.chat    = {username: "alice", message: "What up"}

    this.server  = http.createServer()
    server.on("upgrade", sockets.acceptSocket.bind(sockets))

    server.listen(0, function() {
      ws           = new WebSocket("ws://localhost:" + server.address().port + "/realtime")
      ws.onmessage = function(m) { message = m }
      ws.onopen    = function()  { resume() }
    })
  }})

  after(function(resume) { with(this) {
    ws.onclose = function() {
      server.close(resume)
    }
    ws.close()
  }})

  it("routes chats to a subscribed socket", function(resume) { with(this) {
    ws.send("basement")

    setTimeout(function() {
      service.emit("message", {room: "basement", chat: chat})

      setTimeout(function() {
        resume(function() {
          assertEqual( {username: "alice", message: "What up"}, JSON.parse(message.data) )
        })
      }, 10)
    }, 10)
  }})

  it("does not route chats to an unsubscribed socket", function(resume) { with(this) {
    service.emit("message", {room: "basement", chat: chat})

    setTimeout(function() {
      resume(function() { assertNull( message ) })
    }, 10)
  }})
}})
```

Notice how each of the steps in the first test is separated by a `setTimeout()` call; we have to allow some time for the subscription message to go over the wire to the server *before* we make the service emit an event, and since there's no callback on the client side to tell us when the server has received the message, we have to guess at a suitable amount of time to allow this to happen. Similarly, once the service has emitted the event, we need to allow time for the message to reach the client. Using `setTimeout()` like this tends to be very brittle; we want to make the time interval as short as possible to keep the tests as fast as possible, but if the functionality under test gets slower, the tests can break and it's usually not obvious what the cause is. The only feedback you'll get is that the client didn't receive a message, and it won't be obvious why, especially if you've not changed any of the relevant code recently.

But we can do better than this: we don't have to make real WebSocket connections, we can fake them and still reliably test the message routing functionality, while greatly simplifying the test.

5.6.2. Unit testing with fake sockets

Look at the `acceptSocket()` method in Figure 5.96, "`node/websockets/lib/sockets.js`" again, considering only what's going on at a language level and ignoring what the code actually means. An object is created using the `WebSocket()` constructor, and a random ID is generated. The object has two methods added to it, `onmessage()` and `onclose()`. Calling `onmessage()` with an object whose `data` property is a string subscribes the socket to a room, discarding any existing subscription. Calling `onclose()` unsubscribes the socket. In the `routeMessage()` method, we find all the sockets subscribed to a room and call `send()` on them.

So, imagine that we inject a fake socket object into the module somehow. We could then call its `onmessage()` or `onclose()` methods, have the service emit `"message"` events, and check the socket's `send()` method is called in the right way. To produce fake socket objects, we'd need to stub the `WebSocket()` constructor call in `acceptSocket()`, but there's no obvious way to do that. Stubbing `faye-websocket.Client()` won't work since `sockets.js` stores a reference to the original constructor when it is first loaded; and we can't stub the `WebSocket` reference because it's a local variable inside the `sockets.js` module rather than a public method that we can reference from outside.

However, there is a tool that lets us load modules while stubbing out their `require()` dependencies. The proxyquire[50] library would let us write

```
var Sockets = proxyquire("./sockets", {"faye-websocket": function() { ... }})
```

to load `sockets.js` with the value of `require("faye-websocket")` set to some fake constructor we've set up in the test. This lets us set up a constructor that returns a fake WebSocket object that we can control from our tests suite. The test below calls

```
stub("new", stubs, "faye-websocket").returns(socket)
```

which turns `stubs["faye-websocket"]` into a constructor that will return `socket`. We create an `EventEmitter` to stand in for `ChatService` as before, and create a `Sockets` instance with that, then call `sockets.acceptSocket()` with fake request, stream and body arguments. This call will invoke the fake constructor, injecting our fake WebSocket, and `acceptSocket()` will add the `onmessage()` and `onclose()` methods to it. We can then call those methods from the test, emit `"message"` events from the service, and check that the fake socket receives `send()` at the right times and with the right arguments.

The tests are now much more terse, since there is less infrastructure to set up and everything happens synchronously. This improvement makes is easier to write a few more tests for various state scenarios, while keeping the tests readable.

[50]https://www.npmjs.org/package/proxyquire

Figure 5.98. `node/websockets/spec/sockets_spec.js`

```
var JS         = require("jstest"),
    events     = require("events"),
    proxyquire = require("proxyquire"),
    json       = require("../../basic_server/json_matcher")

var stubs = {
  "faye-websocket": function() {}
}

JS.Test.describe("Sockets", function() { with(this) {
  before(function() { with(this) {
    this.socket = {}
    stub("new", stubs, "faye-websocket").returns(socket)

    var Sockets  = proxyquire("../lib/sockets", stubs)
    this.service = new events.EventEmitter()
    this.sockets = new Sockets(service)
    this.chat    = {username: "alice", message: "What up"}

    sockets.acceptSocket({headers: {}}, {}, "")
  }})

  it("does not route chats to an unsubscribed socket", function() { with(this) {
    expect(socket, "send").exactly(0)
    service.emit("message", {room: "basement", chat: chat})
  }})

  describe("with a subscribed socket", function() { with(this) {
    before(function() { with(this) {
      socket.onmessage({data: "basement"})
    }})

    it("routes chats to a subscribed socket", function() { with(this) {
      expect(socket, "send").given(json(chat))
      service.emit("message", {room: "basement", chat: chat})
    }})

    it("does not route the chat if the socket changes rooms", function() { with(this) {
      socket.onmessage({data: "kitchen"})
      expect(socket, "send").exactly(0)
      service.emit("message", {room: "basement", chat: chat})
    }})

    it("does not route the chat to a socket subscribed to the wrong room",
    function() { with(this) {
      expect(socket, "send").exactly(0)
      service.emit("message", {room: "kitchen", chat: chat})
    }})

    it("does not route the chat to a closed socket", function() { with(this) {
      socket.onclose()
      expect(socket, "send").exactly(0)
      service.emit("message", {room: "basement", chat: chat})
    }})
  }})
}})
```

5.6.3. Full-stack testing

Although we’ve covered most of the required infrastructure with unit tests, it’s still advisable to have one or two integration tests to wire everything together. However, although we’ve changed the notification mechanism from polling to WebSocket, from the end-user's point of view, the application behaves exactly the same; only the implementation details have changed. So, the Testium test for this app remains exactly the same as in Figure 5.91, "`node/chatroom/testium/chat_spec.js`":

Figure 5.99. *`node/websockets/testium/chat_spec.js`*

```
var injectBrowser = require("testium/mocha"),
    User          = require("../../orm/persisted_user"),
    store         = require("../../orm/store"),
    env           = process.env

store.configure({host: env.REDIS_HOST, port: env.REDIS_PORT, database: env.REDIS_DB})

describe("Chatting", function() {
  before(injectBrowser())

  beforeEach(function(resume) {
    new User({username: "bob", password: "sideshow", workFactor: 1}).save(resume)
  })

  afterEach(function(resume) {
    this.browser.navigateTo("/")
    this.browser.clearCookies()
    store.getConnection().flushdb(resume)
  })

  it("displays messages the user posts", function() {
    this.browser.navigateTo("/login")
    this.browser.type("[name=username]", "bob")
    this.browser.type("[name=password]", "sideshow")
    this.browser.click("[type=submit]")

    this.browser.type("[name=room]", "attic")
    this.browser.click("form.room [type=submit]")

    this.browser.type("[name=message]", "Hi, everyone!")
    this.browser.click("form.message [type=submit]")

    this.browser.waitForElementVisible(".messages li")
    this.browser.assert.elementHasText(".messages li", "bob: Hi, everyone!")
  })
})
```

We do however need to make one small change to the start-up script. Recall that Figure 5.87, "`node/chatroom/routes/chats.js`" requires `Chats.PORT` to be set to the port the app is running on; we need to set this variable in the main script where we boot the server:

Figure 5.100. *`node/websockets/main.js`*

```
var store = require("../orm/store"),
    env   = process.env

store.configure({host: env.REDIS_HOST, port: env.REDIS_PORT, database: env.REDIS_DB})

var app = require("./app")
app.listen(env.PORT)

var Chats = require("../chatroom/routes/chats")
Chats.PORT = env.PORT
```

This illustrates the difference between high-level integration tests and unit tests: integration tests ought to describe the end result from the end user's point of view, and they should not need to change if you only change the implementation details. These are the tests that tell you you’ve not broken anything from the point of view of the outside world. Your unit tests *will* need to change as you change the application’s internals, but you should still approach the design of internal components with a view to keeping their APIs relatively stable. Unit tests help you make sure those internal interfaces continue to work as expected, so that you can confidently build functionality on top of them.

5.6.4. Simulating the client

There is one final approach we can explore, a middle-ground between full-stack out-of-process integration testing and isolated unit tests. Recall that we built a simple `Browser` class that can follow links and submit forms in Figure 5.83, “`node/chatroom/spec/browser.js`”, and that the chat UI uses semantic HTML generated in Figure 5.88, “`node/chatroom/views/chat.jade`”. The client-side code posts messages by serialising the message form and posting it to `/chat/:roomName`; if we change the `action` attribute of the form to point to a particular room endpoint, we can do the same thing using our simple browser.

So, given that we have a browser and a WebSocket client that we can use within our Node tests, we can write a test that does the following. The code to perform these steps is shown below.

- Create a user in the database for us to log in as
- Start a server on an available port
- Start a browser we can use to navigate the site
- Connect a WebSocket to the server and send a subscription message
- Use the browser to log in and visit the chat page
- Set the `action` form attribute to simulate the client’s `POST` request
- Enter a message and submit the form
- Check the WebSocket received a notification of the new message
- Close the WebSocket, stop the server and clean the database

Figure 5.101. `node/websockets/spec/websocket_spec.js`

```
var JS        = require("jstest"),
    async     = require("async"),
    WebSocket = require("faye-websocket").Client,
    Browser   = require("../../chatroom/spec/browser"),
    store     = require("../../orm/store"),
    User      = require("../../orm/persisted_user"),
    app       = require("../app")

JS.Test.describe("WebSockets", function() { with(this) {
  before(function(resume) { with(this) {
    var user = new User({username: "james", password: "x", workFactor: 1}),
        self = this

    user.save(function() {
      self.server = app.listen(0, function() {
        var port     = server.address().port
        self.browser = new Browser("Mozilla/5.0", "http://localhost:" + port)

        self.ws      = new WebSocket("ws://localhost:" + port + "/realtime")
        self.message = null
        ws.onmessage = function(message) { self.message = message }
        ws.onopen    = function() { resume() }
      })
    })
  }})

  after(function(resume) { with(this) {
    async.series([
      function(cb) { ws.onclose = function() { cb() } ; ws.close() },
      function(cb) { server.close(cb) },
      function(cb) { store.getConnection().flushdb(cb) }
    ], resume)
  }})

  it("echoes published messages to a subscribed WebSocket", function(resume) { with(this) {
    ws.send("library")

    async.series([
      function(cb) {
        browser.visit("/login", cb)
      }, function(cb) {
        browser.dom("[name=username]").val("james")
        browser.dom("[name=password]").val("x")
        browser.submitForm(browser.dom("form"), cb)
      }, function(cb) {
        var form = browser.dom("form.message")
        form.attr("action", "/chat/library")
        form.find("[name=message]").val("Shh, people are reading!")
        browser.submitForm(form, cb)
      }
    ], function() {
      resume(function() {
        assertEqual( {
          id:        1,
          timestamp: instanceOf("number"),
          username:  "james",
          message:   "Shh, people are reading!"
        }, JSON.parse(message.data) )
      })
    })
  }})
}})
```

This test is very verbose, and in most circumstances I would try to avoid writing a test with this many moving parts. However for our current problem it has one significant advantage. `Browser` and `WebSocket` are not connected to one another — `WebSocket` is not running in the `Browser` JavaScript context, indeed `Browser` doesn't even run JavaScript on pages it visits — and do not share a cookie jar. This means the WebSocket connection is not authenticated as the user we log in as in the browser, and yet the WebSocket receives a notification about the new message. So this test proves, with an end-to-end integration test, something that the Testium test with its single session context can't: the messages that one person posts are distributed to other people that don't need to be logged in as the same user.

The test also runs significantly faster than the Testium one; while Figure 5.99, "`node/websockets/testium/chat_spec.js`" takes about 2.2 seconds to run on my machine, Figure 5.101, "`node/websockets/spec/websocket_spec.js`" takes about 0.3 seconds. So for running integration tests where we don't need the full power of a real web browser, this approach can be much more productive. It's not as fast as an isolated unit test, though: Figure 5.98, "`node/websockets/spec/sockets_spec.js`" runs in under 0.01 seconds, over thirty times faster than the above test.

These different approaches serve to illustrate both the power and the problems of server-side programming. Unlike code running in the browser, server-side code can do absolutely anything your computer is capable of doing, and it can connect to other computers in all sorts of ways. This often presents a challenge when considering how to construct and test things, but it also means you have unlimited freedom in deciding how to structure your project, and no matter what you're doing, there *will* be a way to test it. The important thing, as always, is to keep the tests as simple as possible, and focussed on verifying that the APIs of your modules work as advertised.

6. Command-line interfaces

Node has become a popular platform for building command-line interfaces, or CLIs. For example, the CoffeeScript[1] compiler and the Grunt[2] task framework are both Node programs. In this chapter, we'll examine how we can build Node CLI apps that are easy to test. We'll start by treating a CLI as a black box, and move onto manipulating its internal workings using unit tests.

6.1. A black-box program

To get us started, let's write a simple little program that prints the sum of all the arguments passed to it. Here it is:

Figure 6.1. `node/blackbox_cli/bin/add`

```
#!/usr/bin/env node

var args = process.argv.slice(2)

var sum = args.reduce(function(s, n) {
  return s + Number(n)
}, 0)

console.log(sum)
```

Executable files, or just *executables*, are usually placed in the `bin` directory of a project. 'bin' is short for *binary*; executables are also referred to as binaries even if they contain source code in a scripting language.

The first line, `#!/usr/bin/env node`, is colloquially called a shebang[3]. It tells the operating system how to run the file when it's executed directly, i.e. when we run `./bin/add` in the terminal, the operating system turns this into "`/usr/bin/env node ./bin/add`". We use `/usr/bin/env` because the shebang must use an absolute path, and we don't know where the user will have Node installed; `env` is a standard Unix program[4] with a predictable location, and all it does run the arguments passed to it as a command in the current environment. That is, running "`/usr/bin/env node`" runs `node` by finding it in your `PATH`, exactly as the shell does when you type `node` at the prompt.

In Node, the program's arguments are exposed via the `process.argv` array. The first two elements are usually the string `"node"`, followed by the path of the program being run, followed by the arguments given to the program via the command-line. So, if we were to run

```
$ ./node/blackbox_cli/bin/add foo 42
```

then `process.argv` would be

```
["node", "/absolute/path/to/node/blackbox_cli/bin/add", "foo", "42"]
```

So, the program grabs the arguments using `process.argv.slice(2)`, and uses `Array.reduce()`[5] to sum the given values. It then uses `console.log()` to print the result to stdout.

Writing a test for this is simple: we can call the program with some arguments and see what comes out. Node has a built-in module called `child_process`[6] that makes running external programs quite easy.

[1]http://coffeescript.org/
[2]http://gruntjs.com/
[3]http://en.wikipedia.org/wiki/Shebang_(Unix)
[4]http://en.wikipedia.org/wiki/Env
[5]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce
[6]http://nodejs.org/api/child_process.html

In these tests, we use `child_process.spawn()` to run the program directly. This function takes a path to an executable (or just the name of the executable if it's on the current process's `PATH`), and an array containing the arguments to pass to the executable. It returns an object representing the new process, with methods to control its execution and read and write data to and from it. Just like the Node `process` object, this subprocess has three streams attached to it called `stdin`, `stdout` and `stderr` that provide access to the standard I/O streams for the process. As we discussed in Section 3.4.2, "Checking a readable stream's output", we can use the concat-stream[7] module to collect what the child process writes to stdout and check it.

This test just runs the program with a couple of example inputs and checks the output.

Figure 6.2. `node/blackbox_cli/spec/add_spec.js`

```
var JS     = require("jstest"),
    child  = require("child_process"),
    path   = require("path"),
    concat = require("concat-stream")

JS.Test.describe("add", function() { with(this) {
  before(function() { with(this) {
    this.executable = path.join(__dirname, "..", "bin", "add")
  }})

  it("prints the sum of some integers", function(resume) { with(this) {
    var proc = child.spawn(executable, ["1", "2", "3"])

    proc.stdout.pipe(concat(function(output) {
      var sum = output.toString("utf8")
      resume(function() { assertEqual( "6\n", sum ) })
    }))
  }})

  it("prints the sum of some floats", function(resume) { with(this) {
    var proc = child.spawn(executable, ["3.14", "2.72"])

    proc.stdout.pipe(concat(function(output) {
      var sum = output.toString("utf8")
      resume(function() { assertEqual( "5.86\n", sum ) })
    }))
  }})
}})
```

There's one important thing about these tests: they treat the program under test as a black box, that is, they can't see what the program's internals are like. It could be a compiled C binary and these tests would work just fine. Handy to know you can write tests in your favourite scripting language if you're building such a program.

Command-line arguments are just one source of input a program can use. It can also use its standard input stream, environment variables, config files — both system-wide and user-specific — and the contents of files it's asked to operate on. It can even get information from the internet or from a database. When we write tests, we want to manipulate these inputs and see how the program responds; what it prints to stdout, what its exit status is, what file changes it makes, and so on. These all pose different challenges, especially when used in combination, and you need to pick the right approach to make sure your tests are robust. Let's consider some input sources for our black-box program.

[7]https://npmjs.org/package/concat-stream

6.1.1. Command-line arguments

Our original example already uses command-line arguments; those are the values passed in the command following the name of the program. But the program doesn’t do anything fancy with them aside from turning them from strings into numbers. Most command-line programs have some system of flags and switches — option names that are prefixed with two dashes and control how the program behaves.

Let’s change our program so that it takes a flag called `--round`, which if set makes it round off any fractional numbers it receives. To do this, we need to differentiate this `--round` flag from normal input values, and we can do this with an argument parser like nopt[8].

Figure 6.3. `node/blackbox_cli/bin/add_args`

```
#!/usr/bin/env node

var nopt = require("nopt")

var params = nopt({round: Boolean}, {}, process.argv),
    round  = params.round,
    args   = params.argv.remain

var sum = args.reduce(function(s, n) {
  n = Number(n)
  if (round) n = Math.round(n)
  return s + n
}, 0)

console.log(sum)
```

Just like the original program, this can be easily tested using a combination of `child_process.spawn()` and `concat()`. To test the flag, we have one example that uses it and one that does not, to check the change in behaviour that the flag causes.

[8]https://npmjs.org/package/nopt

Figure 6.4. `node/blackbox_cli/spec/add_args_spec.js`

```
var JS     = require("jstest"),
    child  = require("child_process"),
    path   = require("path"),
    concat = require("concat-stream")

JS.Test.describe("add_args", function() { with(this) {
  before(function() { with(this) {
    this.executable = path.join(__dirname, "..", "bin", "add_args")
  }})

  it("prints the sum of some floats", function(resume) { with(this) {
    var proc = child.spawn(executable, ["3.14", "2.72"])

    proc.stdout.pipe(concat(function(output) {
      var sum = output.toString("utf8")
      resume(function() { assertEqual( "5.86\n", sum ) })
    }))
  }})

  it("prints the sum of some rounded floats", function(resume) { with(this) {
    var proc = child.spawn(executable, ["--round", "3.14", "2.72"])

    proc.stdout.pipe(concat(function(output) {
      var sum = output.toString("utf8")
      resume(function() { assertEqual( "6\n", sum ) })
    }))
  }})
}})
```

You can get much fancier than this with argument parsing, but this pattern remains useful: use nopt or posix-argv-parser[9] to parse the program arguments, and use `child_process.spawn()` and `concat()` to launch a process and check its output. Since you're probably going to come up with quite a lot of examples of what the program can do, it might help to extract all the boilerplate into a function to slim the tests down a bit:

[9]https://npmjs.org/package/posix-argv-parser

Figure 6.5. `node/blackbox_cli/spec/add_args_optimised_spec.js`

```
var JS     = require("jstest"),
    child  = require("child_process"),
    path   = require("path"),
    concat = require("concat-stream")

var executable = path.join(__dirname, "..", "bin", "add_args")

var execute = function(args, callback) {
  var proc = child.spawn(executable, args)
  proc.stdout.pipe(concat(function(output) {
    callback(output.toString("utf8"))
  }))
}

JS.Test.describe("add_args 2", function() { with(this) {
  it("prints the sum of some floats", function(resume) { with(this) {
    execute(["3.14", "2.72"], function(output) {
      resume(function() { assertEqual( "5.86\n", output ) })
    })
  }})

  it("prints the sum of some rounded floats", function(resume) { with(this) {
    execute(["--round", "3.14", "2.72"], function(output) {
      resume(function() { assertEqual( "6\n", output ) })
    })
  }})
}})
```

You might notice that running these tests is much slower than it would be if we tested the addition and rounding logic directly. This is because of the overhead of spinning up a new Node process for each test, and piping output into our test suite. This suggests that testing the internal functions might be preferable to executing the program from outside, but there will be even stronger motivations for doing this later.

6.1.2. Environment variables

Sometimes, you want to make an aspect of a program's behaviour controlled via an implicit setting, rather than having to explicitly use a flag every time you run it. This is what environment variables do: every Unix program runs in an *environment*, which roughly speaking is a bag of name-value pairs that contain settings. If you run `env` in your terminal you'll see the environment your shell is running with; here's a few of my environment variables:

Figure 6.6. Common Unix environment variables

```
$ env | sort
EDITOR=vim
HOME=/home/jcoglan
LANG=en_GB.UTF-8
LOGNAME=jcoglan
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
PWD=/home/jcoglan/projects/javascript-testing-recipes
SHELL=/bin/bash
TERM=xterm-256color
```

When a Unix process spawns a child, that child process inherits its parent's environment, and can make changes to it that are only visible to the child process. So, when you run a program from the terminal, it can see all those settings that `env` shows you. Some of these settings are standardised, for example `HOME` is the current user's home directory, `PATH` tells the shell which directories to find programs in, and `EDITOR` tells programs what to launch if they want the user to edit some text; for example when I run `git-commit` in this environment it will launch `vim` so I can enter a commit message.

Environment variables can be set by the user, either for the duration of a shell session by typing

Figure 6.7. Exporting an environment variable

```
$ export FOO=something
```

or by setting the variable just for one command, for example the `FORMAT` variable we use for running tests:

Figure 6.8. Setting a variable for one command

```
$ FORMAT=tap node node/hello_world/test.js
1..1
ok 1 - Hello, world! runs a test
```

So you can invent and use environment variables to control your programs. Let's replace the command-line argument `--round` to our `add` program with a `ROUND` environment variable, which Node exposes through `process.env`:

Figure 6.9. `node/blackbox_cli/bin/add_env`

```
#!/usr/bin/env node

var round = "ROUND" in process.env,
    args  = process.argv.slice(2)

var sum = args.reduce(function(s, n) {
  n = Number(n)
  if (round) n = Math.round(n)
  return s + n
}, 0)

console.log(sum)
```

Just as you can pass command-line arguments when using `child_process.spawn()`, you can also set the environment that you want the child process to run in. This lets you try running the program with different environment variables to check what happens. To do this, we make a new object that inherits from the current process's environment, and change some of its settings before handing it to `child_process.spawn()`.

Figure 6.10. `node/blackbox_cli/spec/add_env_spec.js`

```
var JS     = require("jstest"),
    child  = require("child_process"),
    path   = require("path"),
    concat = require("concat-stream")

JS.Test.describe("add_env", function() { with(this) {
  before(function() { with(this) {
    this.executable = path.join(__dirname, "..", "bin", "add_env")
  }})

  it("prints the sum of some rounded floats", function(resume) { with(this) {
    var env = Object.create(process.env)
    env.ROUND = "1"

    var proc = child.spawn(executable, ["3.14", "2.72"], {env: env})

    proc.stdout.pipe(concat(function(output) {
      var sum = output.toString("utf8")
      resume(function() { assertEqual( "6\n", sum ) })
    }))
  }})
}})
```

Note that when you use the `env` parameter to `child_process.spawn()`, that sets the entire environment for the child process. i.e. if you only pass `{env: {ROUND: "1"}}` then the child process will *only* see that one environment variable. By saying `env = Object.create(process.env)`, we make an environment that inherits from the current process[10], as environments usually work. Whether you want to inherit the current environment or start with a blank slate depends on the kind of program you're testing, but in this case the child process needs to *at least* see the same `PATH` as the test process, so that the shebang `#!/usr/bin/env node` resolves to the Node you're currently running.

6.1.3. Configuration files

Programs with more complex configuration, or programs that want to be able to save changes to their configuration, often use configuration files rather than environment variables. On most Unix systems, system-wide configuration lives in the `/etc/program-name` directory and user-specific configuration lives in `$HOME/.config/program-name` or `$HOME/.program-name`.

But before you hard-code those locations into your program, you should see a big red flag. In order to test the program, we'd need to poke configuration data into a shared system directory or into the user's home directory. Manipulating the state of the host system is dangerous, since it may clobber the user's existing config and might require root privileges. Ideally, we should isolate the effect of the tests so they don't manipulate any files outside the current project.

We can do this by allowing an environment variable to override the default config file location, and this has the added benefit that it allows the end user to change the location if they want their config stored somewhere else. Here's a version of our `add` program that looks in a config file for the rounding setting. It uses the environment variable `ADD_CONFIG_PATH` to let the user override the config path.

Figure 6.11. `node/blackbox_cli/bin/add_config`

```
#!/usr/bin/env node

var DEFAULT_CONFIG_PATH = "/etc/add/config.json"

var fs     = require("fs"),
    args   = process.argv.slice(2),
    path   = process.env.ADD_CONFIG_PATH || DEFAULT_CONFIG_PATH,
    config = fs.readFileSync(path)

config = JSON.parse(config.toString("utf8"))

var sum = args.reduce(function(s, n) {
  n = Number(n)
  if (config.round) n = Math.round(n)
  return s + n
}, 0)

console.log(sum)
```

We can test this using the technique we discussed above for passing an environment variable into the child process. Before each test, we make a new env using `Object.create(process.env)`, with `ADD_CONFIG_PATH` set to a path inside the project's test directory. Each test writes a different config file to that location using `fs.writeFile()` before running the program, in the modified environment, and checking its output. After each test, we need to make sure to delete our fake config file using `fs.unlink()`.

Notice how the inner `before()` blocks express the name of the containing `describe()` block in code; the `describe()` expresses something abstract like "with rounding enabled", and the `before()` block

[10]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create

contains the code to make that the case. Using the test names to express intention helps keep your test suite easy to understand.

Figure 6.12. *`node/blackbox_cli/spec/add_config_spec.js`*

```
var JS     = require("jstest"),
    child  = require("child_process"),
    fs     = require("fs"),
    path   = require("path"),
    concat = require("concat-stream")

JS.Test.describe("add_config", function() { with(this) {
  before(function() { with(this) {
    this.executable = path.join(__dirname, "..", "bin", "add_config")

    this.env = Object.create(process.env)
    env.ADD_CONFIG_PATH = path.join(__dirname, ".config.json")
  }})

  after(function(resume) { with(this) {
    fs.unlink(env.ADD_CONFIG_PATH, resume)
  }})

  describe("with rounding enabled", function() { with(this) {
    before(function(resume) { with(this) {
      fs.writeFile(env.ADD_CONFIG_PATH, '{"round": true}', resume)
    }})

    it("prints the sum of some rounded floats", function(resume) { with(this) {
      var proc = child.spawn(executable, ["3.14", "2.72"], {env: env})

      proc.stdout.pipe(concat(function(output) {
        var sum = output.toString("utf8")
        resume(function() { assertEqual( "6\n", sum ) })
      }))
    }})
  }})

  describe("with rounding disabled", function() { with(this) {
    before(function(resume) { with(this) {
      fs.writeFile(env.ADD_CONFIG_PATH, '{"round": false}', resume)
    }})

    it("prints the sum of some floats", function(resume) { with(this) {
      var proc = child.spawn(executable, ["3.14", "2.72"], {env: env})

      proc.stdout.pipe(concat(function(output) {
        var sum = output.toString("utf8")
        resume(function() { assertEqual( "5.86\n", sum ) })
      }))
    }})
  }})
}})
```

6.1.4. Standard input

Many CLIs read from stdin and process the input before emitting it to stdout. Such programs can be quite easily tested by writing to the child process's `stdin` stream and reading from its `stdout` stream, both of which are exposed by the object returned by `child_process.spawn()`. Also, as covered in Section 3.4.5, "Decoupled streams", it's not usually necessary to test such programs as a black box since you can implement the logic as a standalone stream that's not coupled to standard I/O.

But some CLIs are interactive; they ask the user to confirm things, or enter their password or other personal details. Some programs ask for their configuration interactively, `ssh-keygen` is a good example of this. This is typically done using the readline[11] library; here's a small example that asks the user a silly question and prints a response based on the user's input. If the input is not what the program expects, the response is written to stderr and it also changes the exit status to indicate success or failure.

Figure 6.13. `node/blackbox_cli/bin/doge`

```
#!/usr/bin/env node

var readline = require("readline")

var interface = readline.createInterface({
  input:  process.stdin,
  output: process.stderr
})

interface.question("Who's a good doge? ", function(answer) {
  if (answer === "such question, wow") {
    console.log("Good doge!")
    process.exit(0)
  } else {
    console.error("Nope")
    process.exit(1)
  }
})
```

Notice how the readline interface writes its question prompts to standard *error*. This is because it's part of the user interface, not the logical output of the program, and since Unix only gives us two possible streams to send program output to, stderr is sometimes used for this.

Let's try to write a test for this. We'll spawn a child process, write the user's input to stdin, and read from stdout or stderr depending on which response we're expecting. These tests use concat-stream to bundle up the entire output of each stream before checking it.

[11]http://nodejs.org/api/readline.html

Figure 6.14. `node/blackbox_cli/spec/doge_quick_spec.js`

```
var JS     = require("jstest"),
    child  = require("child_process"),
    path   = require("path"),
    concat = require("concat-stream")

JS.Test.describe("doge", function() { with(this) {
  before(function() { with(this) {
    this.executable = path.join(__dirname, "..", "bin", "doge")
    this.proc       = child.spawn(executable)
  }})

  it("rewards a good doge", function(resume) { with(this) {
    proc.stdin.write("such question, wow\n")

    proc.stdout.pipe(concat(function(output) {
      output = output.toString("utf8")
      resume(function() { assertEqual( "Good doge!\n", output ) })
    }))
  }})

  it("punishes a bad doge", function(resume) { with(this) {
    proc.stdin.write("WOOF!\n")

    proc.stderr.pipe(concat(function(output) {
      output = output.toString("utf8")
      resume(function() {
        assertEqual( "Who's a good doge? Nope\n", output )
      })
    }))
  }})
}})
```

These seem simple enough but there are some subtleties that could cause problems later. Notice how the second test's output contains the question we asked the user, but the first test's output doesn't. This is because we write the UI to stderr, and the second test's output goes to stderr, so when we read from stderr we get the UI mixed in with the output. In more complex programs this becomes a big problem.

Also, we haven't really written the test in a way that *interacts* with the program being tested. We just throw user input to stdin immediately, without checking which question we've been asked, or even waiting for the question to be asked at all. This might affect the behaviour of the program, and makes the test quite unexpressive: it doesn't really show how the program works when a user interacts with it for real.

To fix these problems, as well as to fix the fact we're not checking the program's exit status, we need to listen out for individual prompts from the program and send responses to them at appropriate times. We need to differentiate user interface from logical output and act accordingly. The following test rewrites our previous attempt to be more explicit about the data flow of the program and how it should be interacted with.

Notice in particular how the first test does not have to check what `stdout` is emitting, but the second test does: the UI and output are interleaved in the second test but not in the first. If the program had much more conditional interface flow, however, the tests would need to reflect this.

Figure 6.15. `node/blackbox_cli/spec/doge_spec.js`

```
var JS    = require("jstest"),
    child = require("child_process"),
    path  = require("path")

JS.Test.describe("doge", function() { with(this) {
  before(function() { with(this) {
    this.executable = path.join(__dirname, "..", "bin", "doge")
    this.proc       = child.spawn(executable)
    this.output     = null
  }})

  it("rewards a good doge", function(resume) { with(this) {
    proc.stdout.on("data", function(chunk) {
      output = chunk.toString("utf8")
    })
    proc.stderr.on("data", function(chunk) {
      proc.stdin.write("such question, wow\n")
    })
    proc.on("exit", function(status) {
      resume(function() {
        assertEqual( "Good doge!\n", output )
        assertEqual( 0, status )
      })
    })
  }})

  it("punishes a bad doge", function(resume) { with(this) {
    proc.stderr.on("data", function(chunk) {
      var response = chunk.toString("utf8")
      if (response === "Who's a good doge? ") {
        proc.stdin.write("WOOF!\n")
      } else {
        output = response
      }
    })
    proc.on("exit", function(status) {
      resume(function() {
        assertEqual( "Nope\n", output )
        assertEqual( 1, status )
      })
    })
  }})
}})
```

This is more explicit but I hope you can see how quickly this approach becomes complicated and cumbersome. In addition, some libraries for consuming user input, such as the password prompting pw[12] module, only work if stdin is a TTY[13], which severely restricts our ability to test it as a child process. Even if the test process is running attached to a TTY, it's not a good idea to make the child process inherit this stream since its output will then be intermingled with the output of your tests.

We've encountered several problems that indicate that testing CLIs as black boxes can be tricky, and in cases where the program talks to an external system like a database or the internet it can be almost impossible. Let's see what we can do to improve matters.

[12]https://npmjs.org/package/pw
[13]http://en.wikipedia.org/wiki/TTY

6.2. Breaking open the black box

Testing CLIs as a black box can be awkward at best and impossible at worst. Controlling the environment, collecting stream output, dealing with config files and standard I/O all introduce complexity, but that complexity is manageable. For some tasks though, we really need control of the inner workings of the program so we can stub things out. Let’s look at a more realistic example than those we’ve considered so far: a program that talks to an API over the internet to gather and display data for us.

6.2.1. Talking to the internet

Suppose we want a program to help us monitor issues on our GitHub projects. It should request the list of issues for a project and display a summary for us. For example, here are the current open issues on one of my repositories:

Figure 6.16. Displaying open issues for a GitHub repo

```
$ ./node/github_issues/bin/blackbox faye faye-websocket-node
3 open issues:

#27: Forward networking error events to the websocket event listeners.
https://github.com/faye/faye-websocket-node/pull/27
[ramsperger] Thu, 24 Oct 2013 19:00:29 GMT

#26: Allow server to limit max frame length (avoid DOS and crash)
https://github.com/faye/faye-websocket-node/issues/26
[glasser] Sat, 12 Oct 2013 02:12:05 GMT

#16: Compression support with "deflate-frame" protocol extension
https://github.com/faye/faye-websocket-node/issues/16
[nilya] Fri, 14 Sep 2012 00:37:40 GMT
```

I’ve written the first version of this program as a simple script, hence the name `blackbox`. Its source is printed below. The program takes the repo owner and project name via `process.argv`, makes an HTTPS[14] request to `api.github.com` to get the open issues for that project, checks the response was successful, then prints the title, URL, author and creation date for each issue. If there’s an error sending the request or any non-successful response is received, the error is printed and the program exits with a non-zero status.

One thing should jump out about this program, even more so than when we were testing configuration files. This program talks to the internet, so we’d need to provide stub responses in our tests. Otherwise, the tests will fail every time anyone opens, closes or edits an issue on the repo we’re testing against. But the way the program is written, this is almost impossible to achieve without heavily manipulating the system: you’d need to spin up a local server that returns canned GitHub API responses, put an entry in `/etc/hosts` to route `api.github.com` to the local machine, and get an SSL certificate that Node will validate successfully[15].

Here’s the full program:

[14]http://nodejs.org/api/https.html
[15]This would involve changing the certificate chain in `libcurl` so that it trusts a self-signed certificate, or getting an actual SSL certificate for `api.github.com`. One of those is a lot of hassle and the other ought not to be possible.

Figure 6.17. `node/github_issues/bin/blackbox`

```
#!/usr/bin/env node

var https  = require("https"),
    concat = require("concat-stream"),
    owner  = process.argv[2],
    repo   = process.argv[3]

var request = https.request({
  method:   "GET",
  host:     "api.github.com",
  path:     "/repos/" + owner + "/" + repo + "/issues",
  headers:  {"User-Agent": "Node.js"}
})

request.on("response", function(response) {
  if (response.statusCode !== 200) {
    console.error("Repository not found")
    process.exit(1)
  }

  response.pipe(concat(function(body) {
    var issues = JSON.parse(body.toString("utf8"))
    console.log(issues.length + " open issues:\n")

    issues.forEach(function(issue) {
      var date = new Date(issue.created_at).toGMTString()
      console.log("#" + issue.number + ": " + issue.title)
      console.log(issue.html_url)
      console.log("[" + issue.user.login + "] " + date + "\n")
    })
  }))
})

request.on("error", function(error) {
  console.error("Error connecting to GitHub: " + error.message)
  process.exit(1)
})

request.end()
```

Since this program is almost impossible to test as-is, we're going to need to break it up and unit-test the internal bits. We can start with isolating the dependency on external resources, i.e. create an API wrapper for talking to GitHub.

6.2.2. Wrapping external dependencies

The first step in our refactoring will be to extract the logic for talking to GitHub into a simple API that we call from elsewhere in the program. This means that instead of having HTTP request code with `console.log()` and `process.exit()` calls baked into its response handlers, we move the HTTP logic into a module that just returns whatever the result of the request was.

It's important to keep calls to `console.log()` and `process.exit()` out of our application code since those will interfere with the tests, either by interspersing the test output with program output or by prematurely ending the test run. This has the side effect of making the code more reusable, since it doesn't make assumptions about what the user wants to do with the result of the request.

Here's a module containing just the HTTP logic and error handling; it makes a request, and if it's successful it hands back the parsed JSON body.

Figure 6.18. *node/github_issues/lib/github.js*

```
var https  = require("https"),
    concat = require("concat-stream")

var GitHub = {
  HOSTNAME: "api.github.com",

  getIssues: function(owner, repo, callback) {
    var path    = "/repos/" + owner + "/" + repo + "/issues",
        headers = {"User-Agent": "Node.js"},
        params  = {method: "GET", host: this.HOSTNAME, path: path, headers: headers},
        request = https.request(params)

    request.on("response", function(response) {
      if (response.statusCode !== 200) {
        return callback(new Error("Repository not found: " + path))
      }
      response.pipe(concat(function(body) {
        var issues = JSON.parse(body.toString("utf8"))
        callback(null, issues)
      }))
    })

    request.on("error", function(error) {
      callback(new Error("Error connecting to GitHub: " + error.message))
    })

    request.end()
  }
}

module.exports = GitHub
```

(I could have used the `request`[16] module instead of the `https` module since it's easier to use and easier to stub out. However, this example demonstrates some useful techniques that are more broadly applicable.)

Now, we need to make sure this works. That is, we need to write some tests that say what this module returns given different HTTP response scenarios. To do this, we need to stub out the `https.request()` API and make it return responses that we control. The way we're going to do this illustrates a very general approach for how to stub out dependencies in your programs, it's not just tied to HTTP.

The first thing to realise is that the code in `GitHub.getIssues()` doesn't *know* that it's making an HTTP request. Yes, it's calling an API called `https.request()`, but those are just names, they don't have any deep technical meaning to the code itself. All the code actually depends upon is the fact that `https.request()` returns some object that responds to two methods, `request.on()` and `request.end()`. Therefore, we can use a vanilla `EventEmitter` as our fake response object, adding a stubbed `end()` method to it.

The second thing is that the response object we get from `request.on("response")` is just a stream — we call `pipe()` on it to run the response body through a JSON parser. So, we can use any other stream that returns JSON as a stand-in, and add a `statusCode` property to it since that's the only other aspect of the response the code cares about. We can get some canned response data by grabbing a snapshot from the internet:

Figure 6.19. Saving stub response data

```
$ curl https://api.github.com/repos/faye/faye-websocket-node/issues \
     > node/github_issues/spec/fixtures/issues.json
```

[16]https://npmjs.org/package/request

Once we have that data, we can use `fs.createReadStream()` to create a stream that emits the contents of our snapshot, which is enough to trick our code into thinking it's processing a response body.

The tests proceed as follows. In the `before()` block we set up the fake request and response objects as we've described, and stub `https.request()` to return our fake request. In each test, we manipulate the response — either setting a status code, or emitting an error instead — and check what `github.getIssues()` returns. The order of instructions in the tests is a little unusual; since this API is asynchronous we make a call to it *before* deciding how the request should be resolved. If we emit a `"response"` event before making the `getIssues()` call, nothing would happen.

Figure 6.20. `node/github_issues/spec/github_spec.js`

```
var JS     = require("jstest"),
    github = require("../lib/github"),
    events = require("events"),
    fs     = require("fs"),
    https  = require("https"),
    path   = require("path")

JS.Test.describe("GitHub", function() { with(this) {
  before(function() { with(this) {
    this.request  = new events.EventEmitter()
    this.response = fs.createReadStream(path.join(__dirname, "fixtures", "issues.json"))

    var params = {method: "GET", host: "api.github.com", path: "/repos/foo/bar/issues"}
    stub(https, "request").given(objectIncluding(params)).returns(request)
    stub(request, "end")
  }})

  it("yields the issues on a 200 response", function(resume) { with(this) {
    github.getIssues("foo", "bar", function(error, issues) {
      resume(function() {
        assertNull( error )
        assertEqual( 3, issues.length )
        assertMatch( /^Forward networking error events/, issues[0].title )
      })
    })
    response.statusCode = 200
    request.emit("response", response)
  }})

  it("yields an error on a 404 response", function(resume) { with(this) {
    github.getIssues("foo", "bar", function(error, issues) {
      resume(function() {
        assertEqual( "Repository not found: /repos/foo/bar/issues", error.message )
        assertEqual( undefined, issues )
      })
    })
    response.statusCode = 404
    request.emit("response", response)
  }})

  it("yields an error on request error", function(resume) { with(this) {
    github.getIssues("foo", "bar", function(error, issues) {
      resume(function() {
        assertEqual( "Error connecting to GitHub: No network connection", error.message )
        assertEqual( undefined, issues )
      })
    })
    request.emit("error", new Error("No network connection"))
  }})
}})
```

This, then, describes a general approach to stubbing out system resources your programs depend on: the code that's using the `https` or `fs` modules doesn't really *know* it's making HTTP calls or talking to the filesystem; it just knows it's calling some functions, listening to events and so on. When you stub out these components, you don't need to fully recreate them, you only need to supply enough of their interface to make the code work. If you can use something that looks almost the same, like an event emitter or a stream, by all means do so, and stick whatever extra methods and properties you need onto the fake object. It's JavaScript: there's no static type system and you can change objects pretty much however you like, and this gives you a lot of flexibility when stubbing things.

Once we've shown that the `GitHub` module works as required, we are free to stub its interface rather than that of `https` when we're testing other components. Wrapping HTTP calls in a simpler API that's tailored to our use case makes it easier to stub that functionality out in an expressive way, without dealing with the complexity of HTTP requests, query strings, serialization, status codes, and so on. We've boiled it down to a single function that yields either an error or a list of issue objects, which is simpler to work with.

6.2.3. Presenting information

The other major component of Figure 6.17, "`node/github_issues/bin/blackbox`" besides fetching the issue data is how to display the issues to the user. In the black-box program, this is done by putting `console.log()` calls in the response handler of the API request, but when we modularise the program for testing this won't fly. In general, code that interacts with, and possibly mutates, globals like `process.stdout`, `process.argv` and `process.env` is hard to test, because it can't be isolated from the rest of the system or from the tests themselves. Code that uses `console.log()` will inject things into the middle of the test output, and code that requires changes to `process.env` means setting things the entire process can see and will be affected by. In short: global side-effects considered harmful.

In this situation there's a straightforward way out: rather than think of the presentation logic as a side effect, think of it as a pure function[17], one that takes an issue object and returns a textual representation of it for displaying in the terminal.

We can write a quick spec for what this function should do:

Figure 6.21. `node/github_issues/spec/presenters_spec.js`

```
var JS         = require("jstest"),
    presenters = require("../lib/presenters")

JS.Test.describe("Presenters", function() { with(this) {
  before(function() { with(this) {
    this.issue = {
      number:     99,
      title:      "The 99th problem",
      html_url:   "http://example.com/issues/99",
      user:       {login: "zee-j"},
      created_at: "2013-10-24T19:00:29Z"
    }
  }})

  it("presents an issue as text", function() { with(this) {
    assertEqual( "#99: The 99th problem\n" +
                 "http://example.com/issues/99\n" +
                 "[zee-j] Thu, 24 Oct 2013 19:00:29 GMT\n",
                 presenters.text(issue) )
  }})
}})
```

[17]A *pure function* is a function that has no side effects and does not rely on global state; its only output is its return value and it computes this using only the values of its parameters.

The issue doesn't have to contain real GitHub data, and it doesn't have to be complete. Remember: when faking things out, you only need the fake to have the same interface as the object it stands in for, from the point of view of the code that's using it. So, we don't have a whole issue object here, we only have the fields the presenter needs. We don't use real data, but the field values are of the same type and structure as the real thing: we replace objects with objects, URLs with URLs, strings with strings and numbers with numbers.

Writing a function that satisfies this is fairly easy; we could use a template library but I'm just going to use string concatenation[18]:

Figure 6.22. `node/github_issues/lib/presenters.js`

```
var Presenters = {
  text: function(issue) {
    var date = new Date(issue.created_at).toGMTString()

    return "#" + issue.number + ": " + issue.title + "\n" +
           issue.html_url + "\n" +
           "[" + issue.user.login + "] " + date + "\n"
  }
}

module.exports = Presenters
```

So now we have presentation logic that's decoupled from `console.log()`, and we can test it and reuse it with ease. The final step is to make a class that represents the CLI itself.

6.2.4. Modelling the CLI

We've extracted out most of the ingredients of the `blackbox` program, but we still need to test the program itself, not just its components. Since we need to be able to stub out some of the internal APIs, we're not going to do this by running the program as a child process. Therefore, we need to wrap the contents of the executable in a function or class, so we can run it multiple times with different inputs in our test suite.

The job of this class will be to glue the ingredients together and wire them up to the user interface available to the command-line program: the `argv`, `env`, `exit`, `stdin`, `stdout` and `stderr` APIs. But, the class should not refer to any of those directly; since they are shared globals visible to everything in the test process, manipulating them can cause other program components or the tests themselves to behave strangely. You don't want the program mixing its output into your tests' output, or calling `process.exit()` and quitting the test suite early.

So, our approach will be to inject these `process` APIs that the program needs access to; the class's constructor will take a parameter called `ui` (for 'user interface') that contains all the things the program needs to interact with. In the tests, we can use this parameter to pass in different arguments, environment variables, and fake standard I/O streams, and when we run the program for real it's just a matter of writing a tiny glue script that passes in the real things.

Here's the `CLI` class, which takes `argv` and `stdout` from the `ui` parameter. Its `run()` method executes the logic of the program and invokes a callback to tell us whether there was an error. Rather than calling `process.exit()` in this class, we pass an error back to the caller and let it deal with it properly.

[18]If you are doing this for real, make sure you escape any characters/bytes in the data you're displaying that have special meaning to the terminal; otherwise it's possible for a cleverly named issue to make itself display in green, or hide your cursor, or clear the screen completely.

Figure 6.23. `node/github_issues/lib/cli.js`

```
var github     = require("./github"),
    presenters = require("./presenters")

var CLI = function(ui) {
  this._argv   = ui.argv
  this._stdout = ui.stdout
}

CLI.prototype.run = function(callback) {
  var owner = this._argv[2],
      repo  = this._argv[3],
      self  = this

  github.getIssues(owner, repo, function(error, issues) {
    if (error) return callback(error)
    self._stdout.write(issues.length + " open issues:\n\n")
    issues.forEach(function(issue) {
      self._stdout.write(presenters.text(issue) + "\n")
    })
    callback(null)
  })
}

module.exports = CLI
```

There's not much logic in this class: the code for gathering the data we need, and most of the code for presenting it, is in other classes that we've already tested. Our main concerns when testing this class are to check that it passes the command-line arguments through to the GitHub client correctly, that it writes the issue information to stdout when the repo exists, and that when there is an error it yields the error to the callback function.

As with all integration testing, the point is not to test all the edge cases of the components that make up the program; that should be done using unit tests of the components themselves. The aim should be to test the main different types of behaviour whose logic is in the `CLI` class. In this case, there is only one real *decision* the class makes: what to do on success and what do to on failure, and we only need an example of each test to check the code is wired up correctly.

The tests for the `CLI` class are listed below. We instantiate the class with an array containing our 'fake' `process.argv` arguments, and a stream to stand in for `process.stdout`. We can use Node's `stream.PassThrough` class for this; it's a read/write stream that emits whatever is written to it, so the `CLI` class can write to it as it would to the real stdout, and we can read that output in our tests by piping the stream into `concat()`.

Although this program doesn't make use of the environment or config files, the same approach can be used for those: just as we pass an array in for `argv`, we can pass an object in for `env` and pass in paths for config files to use during tests. When running the program for real, we would pass in `process.env` and the default config file location instead. Or, we could make the executable responsible for reading the config file instead, and it would pass the resulting settings into the `CLI` class; the tests would then just pass in config data without creating fake config files for the class to read.

As in the `GitHub` tests, we use the `issues.json` fixture file to provide some fake data for the success case, and we use `expect()` rather than `stub()` since we want to assert that the input from `argv` is actually passed through to the GitHub client correctly. For the failure case, we make the client yield an error instead of some issue data, and check that the error is passed back to the caller.

Figure 6.24. `node/github_issues/spec/cli_spec.js`

```
var JS     = require("jstest"),
    fs     = require("fs"),
    path   = require("path"),
    stream = require("stream"),
    concat = require("concat-stream"),
    CLI    = require("../lib/cli"),
    github = require("../lib/github")

JS.Test.describe("CLI", function() { with(this) {
  before(function() { with(this) {
    this.stdout = new stream.PassThrough()
    this.cli = new CLI({
      argv:   ["none", "bin/github", "foo", "bar"],
      stdout: stdout
    })
  }})

  describe("when GitHub returns issues", function() { with(this) {
    before(function() { with(this) {
      var fixture = fs.readFileSync(path.join(__dirname, "fixtures", "issues.json"), "utf8")
      expect(github, "getIssues").given("foo", "bar").yields([null, JSON.parse(fixture)])
    }})

    it("displays the issues", function(resume) { with(this) {
      cli.run(function() { stdout.end() })

      stdout.pipe(concat(function(output) {
        resume(function() {
          assertEqual(
            '3 open issues:\n' +
            '\n' +
            '#27: Forward networking error events to the websocket event listeners.\n' +
            'https://github.com/faye/faye-websocket-node/pull/27\n' +
            '[ramsperger] Thu, 24 Oct 2013 19:00:29 GMT\n' +
            '\n' +
            '#26: Allow server to limit max frame length (avoid DOS and crash)\n' +
            'https://github.com/faye/faye-websocket-node/issues/26\n' +
            '[glasser] Sat, 12 Oct 2013 02:12:05 GMT\n' +
            '\n' +
            '#16: Compression support with "deflate-frame" protocol extension\n' +
            'https://github.com/faye/faye-websocket-node/issues/16\n' +
            '[nilya] Fri, 14 Sep 2012 00:37:40 GMT\n' +
            '\n',
            output.toString("utf8") )
        })
      }))
    }})
  }})

  describe("when GitHub returns an error", function() { with(this) {
    before(function() { with(this) {
      stub(github, "getIssues").given("foo", "bar").yields([new Error("ECONNREFUSED")])
    }})

    it("yields the error", function(resume) { with(this) {
      cli.run(function(error) {
        resume(function() { assertEqual( "ECONNREFUSED", error.message ) })
      })
    }})
  }})
}})
```

Now that we've put all the program's logic into classes that we can test, all the executable has to do is create an instance of `CLI`, passing in the real `process` APIs for it to talk to. It then runs the program, and if there's an error it prints the error and exits with a non-zero status. This file is now so simple it doesn't really need testing; all the logic is somewhere else.

Figure 6.25. `node/github_issues/bin/modular`

```
#!/usr/bin/env node

var CLI = require("../lib/cli")

var program = new CLI({
  argv:   process.argv,
  stdout: process.stdout
})

program.run(function(error) {
  if (!error) return
  console.error(error.message)
  process.exit(1)
})
```

If the program makes use of environment variables or configuration files, or needs to know if stdout is a TTY[19], the executable can pass those in too, and the tests can pass in crafted inputs, alternate config paths, and so on.

Figure 6.26. Passing environment variables and configuration paths to a CLI

```
var program = new CLI({
  config: "/etc/github_issues/config.json",
  argv:   process.argv,
  env:    process.env,
  stdout: process.stdout,
  tty:    require("tty").isatty(1)
})
```

6.2.5. Command-line arguments

If you're using an option parser like nopt, it makes sense to put the parsing logic in the `CLI` class rather than the executable glue file. Argument parsing can become quite complex and the parser libraries sometimes have bugs in them, so it's worth testing how the `argv` array gets processed rather than leaving the parsing in the executable and bypassing it in the tests.

These parsers can operate on any array, not just `process.argv`, so you can easily use them in any code you like. Here's another version of the `CLI` class that has been modified to take a `--limit` option to limit the number of issues it displays.

[19]http://nodejs.org/api/tty.html

Figure 6.27. `node/github_issues/lib/cli_limit.js`

```
var nopt       = require("nopt"),
    github     = require("./github"),
    presenters = require("./presenters")

var CLI = function(ui) {
  this._argv   = ui.argv
  this._stdout = ui.stdout
}

CLI.prototype.run = function(callback) {
  var params = nopt({limit: Number}, {}, this._argv),
      owner  = params.argv.remain[0],
      repo   = params.argv.remain[1],
      self   = this

  github.getIssues(owner, repo, function(error, issues) {
    if (error) return callback(error)

    if (params.limit) {
      issues = issues.slice(0, params.limit)
    }

    self._stdout.write(issues.length + " open issues:\n\n")
    issues.forEach(function(issue) {
      self._stdout.write(presenters.text(issue) + "\n")
    })
    callback(null)
  })
}

module.exports = CLI
```

Putting the parsing logic in this class means we can test it easily. We can write a test similar to the success case in `cli_spec.js`, and change the `argv` value we pass in to include the `"--limit"` option. Remember when doing this that the elements of `process.argv` are always strings; the shell does not know you mean some things to be numbers. So, even when passing in numeric options in a fake `argv` array, pass them as strings rather than as numbers so that your tests are a realistic recreation of what happens when the program is run for real.

The modified test is shown below.

Figure 6.28. `node/github_issues/spec/cli_limit_spec.js`

```
var JS     = require("jstest"),
    fs     = require("fs"),
    path   = require("path"),
    stream = require("stream"),
    concat = require("concat-stream"),
    CLI    = require("../lib/cli_limit"),
    github = require("../lib/github")

JS.Test.describe("CLI with limit", function() { with(this) {
  before(function() { with(this) {
    this.stdout = new stream.PassThrough()
    this.cli = new CLI({
      argv:   ["node", "bin/github", "foo", "bar", "--limit", "1"],
      stdout: stdout
    })
    var fixture = fs.readFileSync(path.join(__dirname, "fixtures", "issues.json"), "utf8")
    expect(github, "getIssues").given("foo", "bar").yields([null, JSON.parse(fixture)])
  }})

  it("displays a limited number of issues", function(resume) { with(this) {
    cli.run(function() { stdout.end() })

    stdout.pipe(concat(function(output) {
      resume(function() {
        assertEqual(
          '1 open issues:\n' +
          '\n' +
          '#27: Forward networking error events to the websocket event listeners.\n' +
          'https://github.com/faye/faye-websocket-node/pull/27\n' +
          '[ramsperger] Thu, 24 Oct 2013 19:00:29 GMT\n' +
          '\n',
          output.toString("utf8") )
      })
    }))
  }})
}})
```

6.2.6. Standard input

Since we've used a `PassThrough` stream as a stand-in for standard output in our previous tests, it might be tempting to do the same thing for standard input, so we can simulate the user typing in answers to questions by writing to this stream. But this actually tends to be more hassle than it's worth; having access to the program's inner workings lets us manipulate it in more powerful and structured ways than those permitted by simple text streams. And in some cases, this approach is not even possible: some interactive input modules, such as the `pw`[20] module, do not work if the input stream is not a TTY.

You might be wondering why I've been referring to the object we pass to the `CLI` constructor as the 'user interface', since so far it's mostly been a clone of the `process` object. Well, it's also a place to put higher-level functions that describe user interactions — questions the program can ask, and ways the user can input information. Asking for a password is a good example of this.

The following executable uses the `pw` module to ask the user for a password. This is untestable since `pw` requires a TTY, so replacing it with a 'fake' stream is not an option. Running code that uses `pw` in tests will result in your tests hanging while waiting for a password that is not forthcoming. So, we leave some small amount of code in the executable to integrate with `pw` — just enough logic to handle asking the user for a password — while leaving the rest of the program logic in a `CLI` class where we can test it.

[20]https://npmjs.org/package/pw

Figure 6.29. `node/password_cli/bin/authenticate`

```
#!/usr/bin/env node

var pw  = require("pw"),
    CLI = require("../lib/cli")

var program = new CLI({
  stdout: process.stdout,

  askForPassword: function(callback) {
    process.stderr.write("Password: ")
    pw("*", process.stdin, process.stderr, callback)
  }
})

program.run(function(error) {
  if (!error) return
  console.error(error.message)
  process.exit(1)
})
```

The `CLI` class invokes the `askForPassword()` method on the `ui` object. In production, this will actually ask the user for a password, but in tests we can stub out `askForPassword()` to inject different user input into the program.

If the password is correct, this program prints a welcome message to stdout, but if the password is wrong then it yields an error. Again, we yield an error here to avoid having the `CLI` class call `process.exit()` and aborting our tests early.

Figure 6.30. `node/password_cli/lib/cli.js`

```
var CLI = function(ui) {
  this._ui = ui
}

CLI.prototype.run = function(callback) {
  var self = this

  this._ui.askForPassword(function(password) {
    if (password === "open sesame!") {
      self._ui.stdout.write("Come on in!\n")
      callback(null)
    } else {
      callback(new Error("ACCESS DENIED"))
    }
  })
}

module.exports = CLI
```

To test this class, we inject a `ui` object as before, and use stubs of `askForPassword()` to test each scenario: one scenario has the function yield the correct password, and one does not. This accurately mimics the response of the `pw` module when used for real. We can use this approach to fake out many other kinds of terminal user interactions, including ncurses-style interfaces or interactions with system services such as using the ssh-agent[21] module to sign something using the user's SSH keys. The function name should express the high-level purpose of the interaction, rather than the technical details of its implementation, to make it easy to write tests against.

[21]https://npmjs.org/package/ssh-agent

Figure 6.31. `node/password_cli/spec/cli_spec.js`

```
var JS     = require("jstest"),
    stream = require("stream"),
    CLI    = require("../lib/cli")

JS.Test.describe("Password CLI", function() { with(this) {
  before(function() { with(this) {
    this.ui  = {stdout: {}}
    this.cli = new CLI(ui)
  }})

  describe("when the user enters the right password", function() { with(this) {
    before(function() { with(this) {
      stub(ui, "askForPassword").yields(["open sesame!"])
    }})

    it("prints a welcome message", function(resume) { with(this) {
      expect(ui.stdout, "write").given("Come on in!\n")
      cli.run(resume)
    }})
  }})

  describe("when the user enters an incorrect password", function() { with(this) {
    before(function() { with(this) {
      stub(ui, "askForPassword").yields(["let me in!"])
    }})

    it("yields an error", function(resume) { with(this) {
      cli.run(function(error) {
        resume(function() { assertEqual( "ACCESS DENIED", error.message ) })
      })
    }})
  }})
}})
```

There's one interesting twist to this test, which is that we've used a bare object `{}` instead of a `PassThrough` stream to stand in for `process.stdout`. This is because the program's interaction with stdout is so simple that it can be tested with a single `expect(stdout, "write")` assertion. But also, the fact that we're using a bare object means we'll get an error if the program calls `stdout.write()` when it's not supposed to, since the object does not have a `write()` method. This would not be true if we used a stream object; the program could write to it when it's not supposed to and we would be none the wiser.

Appendix A. JavaScript functions

To understand how to test JavaScript, and in particular how to mock and stub JavaScript APIs, it's helpful to know how JavaScript functions work. The important thing to keep in mind is that JavaScript only has one type of callable object: the function[1]. There are not multiple kinds of functions, but any function *can* be invoked in multiple ways.

A.1. Defining a function

Functions can be defined in several ways. Or, more precisely, functions are just another kind of value, and so can be defined in all the ways 'normal' values can. You can assign them to local variables:

Figure A.1. Assigning a function to a local variable

```
var add = function(x, y) {
  return x + y
}
```

You can assign them to object properties:

Figure A.2. Assigning a function to an object property

```
var object = {}

object.add = function(x, y) {
  return x + y
}
```

You can place them in object literals:

Figure A.3. A function in an object literal

```
var object = {
  add: function(x, y) {
    return x + y
  }
}
```

And finally, functions have one definition style the other values don't have: the named function declaration. This is where the function is given a name before its argument list, and it is roughly equivalent to assigning the function to a `var` local variable with the same name[2]. When this style is used, the name becomes a property of the function itself, rather than just a variable that refers to the function; its name will be used when references to the function are displayed in stack traces or in the WebKit developer tools.

Figure A.4. Declaring a named function

```
function add(x, y) {
  return x + y
}
```

But, the important thing is that despite the stylistic differences, these all create the same type of thing: a function. There are no special kinds of functions, and all the ones we declared above are equivalent.

[1]ECMAScript 6 will introduce another type of function called a *generator*, but you can safely ignore those for now.
[2]There is one subtle difference: function declarations using a named declaration instead of `var` are evaluated *before* all other code within a function scope. This effect is known as 'hoisting'.

A.2. Function calls

What tends to confuse people is that although there is only one kind of function, there are several types of function *call*. Any JavaScript function can be invoked in one of four ways. Which type of call you are making is determined by the syntax you use to make the call.

A.2.1. As a method

Invoking a function 'as a method' means that the expression before the parentheses in the call resembles a property access expression. That is, it uses dot or bracket notation to access a property of an object, and 'calls' the property by following it with parens. Here are some method calls:

Figure A.5. Functions called as a method

```
object.add(2, 3)
object["add"](2, 3)
object["a" + "dd"](2, 3)
functionThatReturnsAnObject().add(2, 3)
```

When a function is invoked with this syntax, the important thing is that inside the body of the function, the keyword `this` refers to the object 'on the left of the dot', that is, the object whose property we are invoking. This is what lets you do object-oriented programming with JavaScript.

A.2.2. As a function

Invoking a function 'as a function' means following the name of the function — or any expression that evaluates to a function — with parentheses (with the exception of method syntax described above). Most often this means the variable that refers to the function is invoked:

Figure A.6. Calling a function as a function

```
add(2, 3)
```

You can also create a function and immediately call it; this is known as an *immediately invoked function expression* or IIFE. It's usually used to do some computation with local variables without polluting the containing scope.

Figure A.7. An immediately invoked function expression

```
(function() {
  var z = x + y
  console.log(z)
})()
```

When a function is invoked in this way — either by name or using an IIFE — its `this` keyword will refer to the global object (`window` in the browser), whose properties hold the global variables, or in strict mode[3] it will be `undefined`.

The gotcha here is that if you grab a function out of an object property and store it in a local variable, then invoke it through that variable, you're making a function call, not a method call. So if the function uses `this` internally, it can assign global variables by accident.

[3]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/Strict_mode

Figure A.8. Polluting the global scope with function calls

```
var person = {
  setName: function(name) {
    this._name = name
  }
}

person.setName("Alice")         // method call
console.log(person._name)       // -> "Alice"
console.log(_name)              // -> ReferenceError

var setName = person.setName
setName("Bob")                  // function call
console.log(person._name)       // -> "Alice"
console.log(_name)              // -> "Bob"
```

Remember: there is only one kind of function, and functions have no idea where they're stored. Just because `setName()` is in the `person` object, that doesn't mean the function knows or cares that it's supposed to be used as a method on `person`. The difference in behaviour is caused by the different *syntax* you use to invoke the function.

This explains why a 'method' will behave strangely when passed as a callback. Consider this code:

Figure A.9. A method being used as a callback

```
var asyncAdd = function(x, y, callback) {
  callback(x + y)
}

var person = {
  setAge: function(age) {
    this._age = age
  }
}

asyncAdd(4 * 20, 7, person.setAge)
```

When we do this, we do exactly the same thing as assigning a 'method' to a local variable. The `asyncAdd()` function uses function-call syntax to invoke `callback()`, so `callback()` gets `this` bound to the global object. And so, the above program creates a global variable called `_age` rather than assigning it to `person`.

To work around this, we pass a callback that uses method-call syntax:

Figure A.10. Making sure a callback makes a method call

```
asyncAdd(4 * 20, 7, function(age) {
  person.setAge(age)
})
```

Or, in JavaScript versions that support it, we can use `Function.bind()`[4] to create a callback that does the same thing; `fn.bind(object)` returns a new function that invokes `fn()` with `this` bound to `object`.

Figure A.11. Making a safe callback with `Function.bind()`

```
asyncAdd(4 * 20, 7, person.setAge.bind(person))
```

[4]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

A.2.3. As an application

JavaScript lets us go even further than pulling functions out of objects and invoking them against the global object. It lets us invoke them against any object we like! All functions have a method[5] called `apply()`, which takes two arguments: a *receiver* and an *argument list*. `apply()` invokes the function with `this` bound to the receiver and its parameters populated from the argument list.

For example, let’s write a function that sets `this._name` but is not stored inside any object. We can then apply this function to whatever object we like.

Figure A.12. Applying functions to objects

```
var setName = function(name) {
  this._name = name
}

var alice = {}, bob = {}

setName.apply(alice, ["Alice"])
setName.apply(bob, ["Bob"])

console.log(alice, bob)
// -> { _name: 'Alice' } { _name: 'Bob' }
```

As well as `apply()`, all functions have a method called `call()`, which is almost the same except you pass the parameters as positional arguments instead of as an array. That is, `f.apply(o, [x, y, z])` and `f.call(o, x, y, z)` are equivalent.

Note that method-call syntax is really just sugar for this, and `object.add(x, y)` is equivalent to `object.add.apply(object, [x, y])`.

A.2.4. As a constructor

The final type of function call is the constructor call, which is where a normal function call is prefixed with the keyword `new`. This has the effect of creating a new object, making the object inherit from the function’s prototype, applying the function to that object then returning it. Or, in code:

```
var alice = new Person("Alice", 87)
```

is equivalent to[6]:

```
var alice = {}
alice.__proto__ = Person.prototype
Person.apply(alice, ["Alice", 87])
```

There is a convention that functions intended to be used as constructors have capitalised names, but remember they are just a function like any other. But we’ve introduced two strange words here, `prototype` and `__proto__`, so let’s explain those.

All functions have a property called `prototype`, which is an object. There’s nothing magical about it, it’s just an object. People often put functions in it, for example:

[5]When I say ‘method’ here, I mean ‘a function that’s intended to be called as a method’.
[6]Not all JavaScript runtimes allow `__proto__` to be modified like this, so treat this example as pseudocode.

Figure A.13. A constructor and prototype

```
var Person = function(name, age) {
  this.setName(name)
  this.setAge(age)
}

Person.prototype.setName = function(name) {
  this._name = name
}

Person.prototype.setAge = function(age) {
  this._age = age
}
```

There is nothing special about the `Person.prototype…` assignments, they're just assigning functions as object properties. Likewise, there is nothing special about `Person.prototype` itself, it's a JavaScript object like any other and you can give it any properties you like. It's only 'special' in the sense that it will have its properties inherited by objects created with `new Person()`.

What *is* special is the `__proto__` property. All objects have one, and it's what makes JavaScript's simple inheritance system work. When you look up a property like `alice.name`, JavaScript checks if `name` exists as a property inside `alice`. If not, it looks for `name` in the object `alice.__proto__`, then `alice.__proto__.__proto__` and so on, until it either finds the required property or runs to the end of the chain and returns `undefined`.

So when we say `alice.__proto__ = Person.prototype`, we're saying that `alice` should inherit all the properties in `Person.prototype`. That's why prototypes are typically used to store the methods for a class of objects. Strictly speaking, JavaScript does not have classes, but people often use the term *class* to mean a constructor and its prototype.

A.2.5. Getters and setters

Okay, I lied. There's a fifth kind of function call that was experimented with in some browsers and was eventually standardised using a feature called property descriptors. Getters and setters mean that a JavaScript property access like `alice.name` or a property assignment like `bob.name = "Bob"` can actually invoke a function rather than simply read or write a property. A good example is the `location` object in browsers; it's special because it's not an inert object: setting any of its properties like `protocol`, `host`, `pathname` or `href` makes the browser replace that part of the current URL and load a new page.

The original (and now deprecated) way to make your own getters and setters was using `__defineGetter__()` and `__defineSetter__()`, for example:

Figure A.14. Using `__defineGetter__()` and `__defineSetter__()`

```
var person = {
  dob: new Date(1984, 1, 25)
}

person.__defineGetter__("age", function() {
  var year = 365 * 24 * 60 * 60 * 1000
  return (Date.now() - this.dob.getTime()) / year
})

person.age // -> 29.7

person.__defineSetter__("name", function(name) {
  this._name = name.toUpperCase()
})

person.name = "Alice"
person._name // -> "ALICE"
```

So, `__defineGetter__()` and `__defineSetter__()` let you turn any property access or assignment into an implicit function call. They have now been superceded[7] by the property descriptor functions, which include `Object.create()`[8], `Object.defineProperty()`[9], `Object.defineProperties()`[10] and `Object.getOwnPropertyDescriptor()`[11]. Rather than letting us only set the value of a property, property descriptors let us set metadata about that property, such as whether it can have its value changed and whether it should be included in enumeration using the `for … in` syntax. Among the metadata that descriptors let us set are the `get` and `set` fields, which specify a getter and setter for the property respectively.

Using these APIs the above example could be written like this:

Figure A.15. Defining getters and setters with `Object.defineProperty()`

```
Object.defineProperty(person, "age", {
  get: function() {
    var year = 365 * 24 * 60 * 60 * 1000
    return (Date.now() - this.dob.getTime()) / year
  }
})

Object.defineProperty(person, "name", {
  set: function(name) {
    this._name = name.toUpperCase()
  }
})
```

There's also syntactic sugar for getters and setters baked into the object literal syntax on some platforms:

[7]These functions became available in Firefox 4, Internet Explorer 9, Safari 5, Opera 12, and Chrome and Node (which are both based on the V8 engine) have had them since very early versions. Source: http://kangax.github.io/es5-compat-table/
[8]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create
[9]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty
[10]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties
[11]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptor

Figure A.16. Getters and setters in object literals

```
var person = {
  dob: new Date(1984, 1, 25),

  get age() {
    var year = 365 * 24 * 60 * 60 * 1000
    return (Date.now() - this.dob.getTime()) / year
  },

  set name(name) {
    this._name = name.toUpperCase()
  }
}

person.age // -> 29.7
```

Why did I not count getters and setters as one of the four types of function call? Two reasons: first, depending on which platforms you're supporting, there may still be old browsers hanging around that preclude their widespread use, and second, they don't often come up in the context of stubbing and mocking. Stubbing them correctly is hard and libraries typically don't try to support it, and it's usually easier to stub a getter out with a simple property with the value that the getter would return.

A.3. Functions and mocking

In order to effectively mock and stub functions, you need to be aware of the different types of call and how they work. In particular you need to be aware that when you write something like

```
stub(object, "method")
```

all that's going on behind the scenes is that the framework is assigning some shim function like this:

```
object["method"] = function() { ... }
```

The shim will be configured to return whatever values you set the stub up with, and it will record everything it can about the calls made to it so that mock expectations can be checked. An important thing to note is that the shim function cannot tell, when it is called, exactly what type of syntax was used, for example you can simulate `fn.apply(receiver, [x, y, z])` like this:

Figure A.17. Faking `Function.apply()`

```
receiver.__tmp__ = fn
receiver.__tmp__(x, y, z)
delete receiver.__tmp__
```

This invokes `fn()` with `this` bound to `receiver`, just like `Function.apply()` does.

So, a function cannot tell exactly how it is invoked, what name it was called by, which syntax was used, and so on. The only things a function *can* tell about a given call are:

- What the value of `this` is
- What the values of the parameters are
- Whether it was invoked using the `new` keyword[12]

So, stubbing libraries typically give you tools to express stub responses and mock expectations in terms of those things, and you should think about stubbing in terms of those things, not in terms of function call

[12]You can detect whether a function was invoked using `new` by evaluating `(this instanceof functionName)`, or `(this instanceof arguments.callee)`, which will be `true` if `new` was used to invoke the current function.

syntax. The important thing when testing object interactions is, what information did object A send to object B, and what should B send back, and checking the `this` binding, the parameters and the presence of `new` is enough to make strong assertions about these interactions.

A.3.1. Mocking and bound methods

Sometimes, you will run into code that looks like you should be able to test it with mocks, for example:

Figure A.18. `browser/event_listeners/backbone_view.js`

```
var View = Backbone.View.extend({
  events: {
    "click a": "handleLinkClick"
  },

  handleLinkClick: function(event) {
    event.preventDefault()
    var linkText = $(event.target).text()
    this.$("h2").text(linkText)
  }
})
```

You might expect that you could test this by saying that `handleLinkClick()` should be called when the link is clicked, like so:

Figure A.19. Failed mocking on a Backbone view

```
JS.Test.describe("Backbone view", function() { with(this) {
  extend(HtmlFixture)
  fixture(' <a href="/">Home</a> ')

  it("calls handleLinkClick() when the link is clicked", function(resume) { with(this) {
    var testView = new View({el: fixture})
    expect(testView, "handleLinkClick")
    syn.click(fixture.find("a"), resume)
  }})
}})
```

Unfortunately, this won't work, because of the way that Backbone binds events. Essentially, it does something equivalent to this[13] when you call `new View()`:

Figure A.20. Backbone's internal event binding method

```
fixture.find("a").on("click", testView.handleLinkClick.bind(testView))
```

So, the function referenced by `testView.handleLinkClick` is looked up when the event binding is established, when the view is created. This means that any stub on that method that's set up after the view is created — as happens in our test, where `expect()` comes *after* `new View()` — will have no effect: the event listener is already bound to use the original method. If Backbone instead bound the event like this:

Figure A.21. Alternative event binding implementation

```
fixture.find("a").on("click", function(event) {
  return test.handleLinkClick(event)
})
```

then the reference to `test.handleLinkClick` would not be resolved until the event was *triggered*; this is known as late binding[14]. It means there is time between the creation of the view and the triggering of the event to insert mocks and stubs.

[13]It actually uses the `_.bind()` function from Underscore, see http://underscorejs.org/#bind.
[14]http://en.wikipedia.org/wiki/Late_binding

When confronted with bound methods, there are one or two ways to mock them. Since JavaScript prototypes are just regular objects, you could mock the method on `View.prototype` instead, before calling `new View()`. The instance would inherit the stubbed function and that would be bound to the event listener.

Figure A.22. Mocking on the prototype

```
JS.Test.describe("Backbone view", function() { with(this) {
  extend(HtmlFixture)
  fixture(' <a href="/">Home</a> ')

  it("calls handleLinkClick() when the link is clicked", function(resume) { with(this) {
    expect(View.prototype, "handleLinkClick")
    var testView = new View({el: fixture})
    syn.click(fixture.find("a"), resume)
  }})
}})
```

But, stubbing methods on the prototype means they are stubbed for *all* instances of that class, and we only really want to check *which* instance the method is invoked on. We can use the `on()` modifier to check that, but things become a little convoluted since we need to modify the mock with information that comes into existence after it's set up:

Figure A.23. Mocking on the prototype, with a specified receiver

```
JS.Test.describe("Backbone view", function() { with(this) {
  extend(HtmlFixture)
  fixture(' <a href="/">Home</a> ')

  it("calls handleLinkClick() when the link is clicked", function(resume) { with(this) {
    var mock = expect(View.prototype, "handleLinkClick")
    var testView = new View({el: fixture})
    mock.on(testView)
    syn.click(fixture.find("a"), resume)
  }})
}})
```

The line `mock.on(testView)` is what makes sure the method is invoked with `this` bound to our instance, but it makes for ugly test code.

Finally, you could always stub `View.prototype.handleLinkClick.bind()` itself to return some function that uses late binding and is therefore amenable to stubbing. But in practice, since `Function.bind()` does not exist on all platforms and people tend to use various library or home-grown helper functions for it, that gets messy very quickly. Where at all possible, try to program in a way that supports late binding, if you think you'll want to stub methods during tests.

Appendix B. The ‘with’ statement

The tests in this book mostly look something like this:

Figure B.1. A `jstest` test with `with` statements

```
JS.Test.describe("Array", function() { with(this) {
  before(function() { with(this) {
    this.array = [1,2,3]
  }})

  it("contains its elements", function() { with(this) {
    assertEqual( arrayIncluding(2), array )
  }})
}})
```

It’s worth explaining what the purpose of those `with(this)` blocks is, especially since in other frameworks like Jasmine they aren’t used:

Figure B.2. A Jasmine test without `with` statements

```
describe("Array", function() {
  var array

  beforeEach(function() {
    array = [1,2,3]
  })

  it("contains its elements", function() {
    expect(array).toContain(2)
  })
})
```

B.1. The global variable problem

The problem with this is that it requires `describe()`, `it()`, `expect()` and any other functions in the Jasmine interface to be bound to global variables. A testing framework coexists in the same JavaScript environment as your application code, and because of this it should try not to pollute the environment the application can see. We should avoid situations where our code seems to work because it relies on functionality introduced by the test environment that’s not available in production.

This means a testing framework should not add new methods to built-in or library APIs, and should try to minimise its use of global variables. `jstest` does this by restricting itself to one global variable — `JS` — and making all its other APIs into methods that are only accessible inside the tests.

Within a `describe()` block, `this` refers to the current group of tests, which has methods like `before()`, `after()` and `it()`. Within any of *those* blocks, `this` refers to the current test, which exposes all the assertion methods and which you can use to store per-test data.

So, if we were to write the `jstest` example without using `with`, it would look like this:

Figure B.3. A `jstest` test without `with` statements

```
JS.Test.describe("Array", function() {
  this.before(function() {
    this.array = [1,2,3]
  })

  this.it("contains its elements", function() {
    this.assertEqual( this.arrayIncluding(2), this.array )
  })
})
```

Now, this adds a certain amount of undesirable line noise that makes the test harder to read. So, I use the `with` statement to reduce this noise.

B.2. What does `with` do?

In JavaScript, local variables are scoped by functions. When you reference a variable, JavaScript checks if the variable is defined with `var` within the current `function() { … }` block. If it’s not defined, it looks in the next closest enclosing function, and continues up until either the variable is found or we reach the global scope. This lookup process defines the *scope chain*, and the value of a variable reference is the value of the closest matching variable in the chain.

When you assign to a variable, we go through the same lookup process to find the nearest matching variable, and assign the value to this variable. This is why, in the Jasmine example, the `it()` block can see the value of `array` assigned in the `beforeEach()` block. The `array` variable is defined with `var` in the enclosing `describe()` block, so the assignment in `beforeEach()` and the reference in `it()` can both refer to it.

So, you can think of the scope chain as a list of objects. Each scope in the chain has properties named after the variables that are defined in that scope, and the values of those properties are the values of the variables.

The `with` statement, then, can be thought of as pushing another scope onto this chain. The constructions

```
with (expression) statement
```

and

```
with (expression) {
  statement1
  statement2
  // ...
}
```

evaluate `expression`, which should produce some kind of object (anything that’s not `null` or `undefined`). They then push this object onto the scope chain and run the statement or block with this modified chain, before popping the object off the chain and letting the program continue.

This has the effect of making all the object’s properties look like local variables: all variable references will be looked up in this object first before looking at enclosing functions. This also applies to assignment; if you assign to a variable with the same name as one of the object’s properties, that property will become the target for the assignment rather than a true local variable.

So, in `jstest` you can use this to remove the `this.` prefix from all the methods and test data, by pushing `this` onto the scope chain.

B.3. Caveats

It’s usually inadvisable to use `with` in application code, since it tends to make it harder to reason about what a variable name is bound to. True local variables must appear explicitly in a `var` statement in a function that syntactically encloses the code you’re looking at, whereas variables introduced with `with` can come from any object in the system, whose properties are not immediately visible to you. Plus, use of `with` can have a negative impact on performance.

But since good tests tend to be less complex than application code, and because they don’t have the same performance constraints, we can make limited use of `with` with an object with a known, documented API in order to make our tests easier to read.

However, this is a matter of personal taste. Some people flat-out don’t like `with`, and find it confusing. It’s important that your tests remain readable, so if your team has a strong preference against `with`, avoid using it.

Additionally, due to the pitfalls of using it, it is not available in strict mode[1]. So, if you want to run your tests in strict mode, you can’t use it.

An alternative solution to the line noise problem is to write your tests in CoffeeScript, which replaces `function() { … }` with `->` and `this.` with `@`.

Figure B.4. A test suite written in CoffeeScript

```
JS.Test.describe "Array", ->
  @before ->
    @array = [1,2,3]

  @it "contains its elements", ->
    @assertEqual @arrayIncluding(2), @array
```

Used carefully, CoffeeScript can produce quite clear and readable tests that look more like spec documents than code. Again, you should see what works for you and your team.

[1]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/Strict_mode

Appendix C. Currying

> In mathematics and computer science, currying is the technique of transforming a function that takes multiple arguments in such a way that it can be called as a chain of functions, each with a single argument.
>
> — Wikipedia

In some languages, notably Haskell[1], all functions are curried by default: if you call a function with fewer arguments than it expects, you get back a function that takes the rest of the arguments. In JavaScript, we don't have this feature built into the language but we can mimic it using closures.

Say we have a function that adds three numbers:

```
var add = function(a, b, c) {
  return a + b + c
}
```

We can convert it into 'curried' form by turning it into a set of nested functions that take each argument in turn:

```
var add = function(a) {
  return function(b) {
    return function(c) {
      return a + b + c
    }
  }
}
```

When we call this function, each extra argument we pass in returns us a new function, until we pass in all the required arguments and we get a meaningful result out.

```
> add(1)
[Function]
> add(1)(2)
[Function]
> add(1)(2)(3)
6
```

C.1. Continuables

This idea is used to make 'continuables', a particular style of doing callback-based asynchronous code in JavaScript. The common way to do callbacks in Node is to pass a callback after the 'logical' arguments to a function, which in turn invokes the callback with an error and a return value (the error is `null` in the successful case). For example an async addition function might look like this:

```
var asyncAdd = function(a, b, c, callback) {
  setTimeout(function() {
    var result = a + b + c
    callback(null, result)
  }, 10)
}
```

We'd call this function like so:

```
asyncAdd(1, 2, 3, function(error, result) {
  // ...
})
```

[1]http://www.haskell.org/

The problem with this is that it mixes the logical arguments to a function with control-flow concerns, and it means the async function doesn't return a useful value to represent the computation. What continuables do is separate the logical arguments from the callback by partially currying the function, like this:

```
var asyncAdd = function(a, b, c) {
  return function(callback) {
    setTimeout(function() {
      var result = a + b + c
      callback(null, result)
    }, 10)
  }
}
```

We would call this continuable function like this:

```
asyncAdd(1, 2, 3)(function(error, result) {
  // ...
})
```

This doesn't look like a big difference, but it means that all continuable functions return something with a uniform interface: a function that takes a callback. You can think of continuable functions as generating tasks that you can execute without needing to know all the arguments that went into the task. `asyncAdd(1, 2, 3)` returns a function that, when executed, computes `1 + 2 + 3` and yields the result to your callback.

This uniformity means you can use continuables to generate inputs to various functions in the Async module to string asynchronous functions together.

C.2. Converting functions into continuables

If you already have a set of functions that use 'traditional' callback APIs, you don't need to rewrite them to take advantage of continuables. You can actually use a little metaprogramming[2] to convert a callback function into a continuable. In Section 2.7.4, "Generating steps with continuables" we used a helper module to convert our existing helper functions to continuables; the source code is listed below in Figure C.1, "`node/async_steps/curry.js`".

`Curry.continuable()` takes a function and returns a continuable-ified version of it, or rather, a function that wraps the original in a continuable interface. `Curry.object()` is a little time-saving function that takes an object and returns a copy of it with all the methods converted to continuable style.

[2]http://en.wikipedia.org/wiki/Metaprogramming

Figure C.1. `node/async_steps/curry.js`

```
var Curry = {
  continuable: function(fn) {
    return function() {
      var args = Array.prototype.slice.call(arguments),
          self = this

      return function(callback) {
        args = args.concat([callback])
        return fn.apply(self, args)
      }
    }
  },

  object: function(object) {
    var copy = {}
    for (var key in object) {
      if (typeof object[key] === "function") {
        copy[key] = this.continuable(object[key])
      } else {
        copy[key] = object[key]
      }
    }
    return copy
  }
}

if (typeof module === "object") {
  module.exports = Curry
}
```