PeepCode: Full Stack Node.js Part 1

Introduction

It's PeepCode!

Node.js has surged to popularity in the past few years. It has more watchers on GitHub than Ruby on Rails. Large sites are using it for everything from business to games.

But many features are poorly documented. And the popular Express web framework doesn't give you the full stack in a single package: tests, front-end, data. That's where we come in. We've spent over a year using Node and a few months putting together a full featured application that illustrates the range of tasks involved in building a web application.

And the good news is that it's not too difficult. That's not to say that there's no frustration. We hit quite a few walls and switched software packages several times while putting this video together. But we think the final product is straightforward and shows how easy (and fun) it can be to write web apps with Node.js.

Like all PeepCode tutorials, we like to work on an application that's fun and nicely styled. We thought of this application while eating pie at a local shop, Sweet and Savory Pie in the Fremont neighborhood of Seattle.

The application you'll learn to build in this video and the next is a list of hot pies that have recently come out of the oven at a bakery. You can see the pies that are in the oven and almost ready. You can see the pies that are fresh out of the oven and the others that have been out for a while but are still tasty and delicious.

If you go to /admin, you'll be prompted to log in you will be able to add a new pie to the database or change the status of a pie. The public-facing display will be automatically updated as soon as you make a change. Even though this uses Node.js, which is JavaScript on the server, we used a very small amount of JavaScript on the client; just enough to handle some clicks, do some AJAX calls, and respond to events pushed from the server. You will learn all the things you need to know to write a pretty full-featured server application with Node.js, Express, CoffeeScript, Redis, and the Mocha test framework.

Chapter One: Install

Let's install Node.js and npm, Node's package manage for installing third-party libraries. If you've already installed Node, you can skip this chapter.

The absolute easiest way to install Node is with the graphical Mac or Windows installer linked from the home page of nodejs.org. You'll get the most recent version of Node from a precompiled graphical installer. Props to the Node team for making it so quick and easy.

Click the "download" button and find your operating system. Double click the installer and walk through the prompts. This will install both Node and NPM to /usr/local/bin. Use "--version" on either command to see that you now have the newest version of Node and also NPM.

You may need to add "/usr/local/bin" to your shell PATH. The Windows installer should do that for you automatically. Edit your ~/.bashrc if you haven't already done that.

If you're on a Mac and prefer to work from source, I like to install and upgrade Open Source software with Homebrew. Run "brew update" and "brew install nodejs" will give you a recent version of Node.js. It may take a few minutes to download and compile it. It gave me a version that was a few minor revisions out of date, but still works well.

Type "node --version" to see the version. You'll also need NPM, the Node package manager. Copy this one line and paste it to the terminal to get the latest version of the NPM package manager.

This does assume that your /usr/local directory is writable by you. If it's not you'll have to use "sudo".

Ok, that was quick. Let's start using Node!

Chapter Two: Express.js

In this chapter we'll install the Express web framework and design a directory structure for our application.

Express is popular web framework for Node. It's easy to compare to Ruby's Sinatra, but don't make the mistake of thinking it's only for toy apps. People are using it for all kinds of large and small Node applications.

I prefer to install most dependencies locally inside the application's directory, which is the default with NPM. But Express is worth installing globally so you can generate new Express applications with it. Install it with "npm install" and use the "-g" flag to install it globally to /usr/local. I now have the "express" command available and can generate an application. Let's do that: "express new". The application will be named "HotPie".

Already you may notice how quickly this happened. Granted, it didn't have to do much, but Node.js is extremely fast and it makes development fun to know that you won't be waiting for things to happen. For most of the development of this application, my test suite ran in under half a second; about 500 ms.

Let's change into the HotPie directory. Let's take a quick look at the files it created for us. Express creates a minimal directory structure which is enough for the very simplest of applications. We'll add to it in order to write a more full featured application.

App.js is the entry point for the application. It will setup configuration and require any other code files that need to run. Package.json lists third-party dependencies and version numbers that should be installed. Public holds static stylesheets and images that will be served to the client. Routes holds controller code and URL handlers. And "views" has templates to render HTML or other markup.

To run this starter application, first run the "npm install" command to retrieve and install all dependencies to the local directory. You can see it has created a "node_modules" directory that contains Express and Jade, the default templating framework that ships with Express. There's nothing super special about this directory or these libraries. They're just directories with code that can be used by your application. You can delete them at any time or rebuild it by running "npm install" again.

Now let's start the server. The command is "npm start". Or at least, it should start the application except that it doesn't (in the current version of Express). Why not? NPM expects this "app.js" file to be named "server.js". This convention is used by many hosting companies and is the default filename that the "npm start" command looks for. So I'll rename it to "server.js". Now typing NPM start will start this application.

If it started without this change, you probably have a newer version of Express with a snippet in package.json that redefines the "start" command. I'll refer to it as "server.js" for the rest of this video, so delete this "start" configuration if you see it in your generated application.

This is part of the minor frustration of working with Node. Some community conventions are still being established and aren't known by all developers or aren't followed by all libraries. I've been filing bugs and have seen quick positive responses by developers to adopt these conventions once they learn about them.

So go to localhost:3000 and you'll see a very simple Express application. To quit, hit ctrl-c.

Now that we've proved a simple application works, let's prepare it for further development of this tutorial application. In the "public" directory, delete its contents.

Copy the contents of the "public" directory from the final project and paste them to your working project. In the "views" directory of the final project is a directory named "stylesheets". Paste them here to get the dynamic version of the "stylus" stylesheets used by the application.

Delete the "routes" directory. We're going to write this application in a slightly different way. Make an "apps" directory instead. We'll write several mini applications that will run together. In fact, that's what we'll do in the next chapter.

Chapter Three: Authentication

In this chapter we'll create a route that renders a simple view with a login form. We'll install dependencies with npm. We'll also learn about how to use CoffeeScript with Node and how to organize files.

Although we used Express to generate a new application, it's still very open minded as to how you organize your files. I've taken this as a license to organize my applications in the way that makes the most sense to me.

I want to show you one way I've done things that I think works pretty well. I like treating a web application as a series of smaller web applications. The Django web framework works this way and I think it's a great strategy for organizing your Express applications.

So in this "apps" directory I'm going to make subdirectories for self-contained parts of the application. Since our first feature is a login form, I'll make an "authentication" directory. Inside that I'll write all the code that relates to displaying the login form, checking the user's name and password, and setting up their session to recognize them on future visits to the pages of our application.

Let's write some code. Here in the "apps/authentication" directory, make a file named "routes.coffee". I use the term "routes" to reuse terminology already used in Express. A route is a combination of a URL path and controller code that is run when that URL is visited. Route handlers pull data from a database, read submitted form values, and render views or emit JSON data.

A basic Express route works like this. We'll have access to an "app" variable that we'll use to list the HTTP method used to access this URL, in this case a "get". Then list the path this handler will be used for: "/login". When a person visits "/login" at our domain, I want to render the login form.

The route handler calls a function that we provide. It receives two arguments: the request object and the response object. I usually avoid super short variable names, but

these arguments appear so frequently throughout Express apps that it's a convention to use "req" and "res". Both you and other developers working on your code will know what they mean.

The request object contains information about form and query parameter data. The response object can redirect to a URL or render a template.

Some of these seem so fundamental that you might wonder why they aren't just built in. Why do I have to use a variable to access form data? Here's a core concept that's important to understand: Node libraries have a tendency to be explicit. Functionality doesn't just appear out of thin air. You'll see a specific variable or import a specific library in order to run code.

So let's render the login template using the response object. The "render" method also takes a hash of keys and values that will be passed to the login template. Use the title "login" and pass the name of the stylesheet to use for styling. And there you have the basics of an Express action.

It starts with an HTTP method such as "get", a URL path such as "/login", and a function with your code. It has access to the request and the response. We render a template with some keys and values.

But we're not quite done. We need to put that template somewhere.

I've chosen to make a new directory in this mini app named "views". Create the template inside this directory. Name it "login" and use "jade" as the extension. Jade is the default template engine that ships with Express. Jade is heavily inspired by the Haml template format from Ruby, but isn't exactly the same. It uses indentation and extremely minimal syntax to describe the markup for an HTML or XML document. You won't ever see a closing tag, just indents and outdents. I really like it, and it's used widely by developers who use Express. I also think it works nicely with CoffeeScript; I don't have to do much of a mental shift to switch from CoffeeScript code to Jade markup. But take a look and make your own conclusions.

We're going to build a HTML form, so start with the word "form" to create the form tag. We want it to post to the "/sessions" path and use an HTTP POST. Then we need some input elements nested inside this tag. Indent by two spaces. Make a label. Nested in side it is an input of type "text". It will hold the name of the user.

Make another label. It holds the password and has a type of "password". Finally we'll add a submit button. Make an input of type "submit" with value "submit".

Now we have a form which has several nested labels and inputs inside it. The inputs are wrapped by the label tags. The labels aren't very useful without text. Use the vertical pipe to add static text. This is plain text that will not be interpreted as a tag of any kind. The labels are username and password.

We need to take a trip to "server.js" before this code will run as part of the application.

Open "server.js". First, tell Express that we're using CoffeeScript. The CoffeeScript module will register itself with Node so .coffee files will be dynamically compiled to JavaScript when the server runs. Many projects I've looked at will run that compilation step themselves with some kind of file watching script. Save yourself the trouble! Just "require('coffee-script')" somewhere in your application and don't give it another thought.

Another approach is that people will write the entire application in CoffeeScript and will run it with the "coffee" command instead of "node". But this pretty much requires that you install CoffeeScript globally on any machine you want to run the application from. I prefer to work within the standard Node deployment system and bootstrap my application with a minimal "server.js" in JavaScript. Then install CoffeeScript locally and pull in any other .coffee files from there.

Our second task in "server.js" is to require our mini-app's routes. Scroll down through the default Express configuration. You can see that Jade is used as the default view engine. Separate configuration directives can target development or production. And below that there is a default route handler. Delete that and let's import our own. Require the file we just wrote: "apps/authentication/routes". You don't need to list the extension, just the path and filename.

Remember what I said about Express being explicit? Unlike other web frameworks, it won't automatically find all your controllers and models. You'll have to tell it about each one. As we add other mini apps, we'll list them here.

But they need more than just a simple "require". Remember that we used the "app" variable to define our URL handlers? We have to provide that to the route handler.

In Express the "app" variable is used extensively to hold configuration settings and perform other operations. Although we could set that up as a global variable, it's better practice to pass it in as an argument to code modules that need to use it. So let's use the slightly odd syntax of calling the result of "require" as a function and passing the "app" variable. Although it looks weird, this technique is recommended in the installation instructions for other Node modules and will become more familiar as you see it used elsewhere.

Now let's go back to the routes file and use the "app" variable.

All the code in a Node.js file is local. We have to explicitly tell it what parts of this file we want to make available to the outside world. I've landed on a convention that I'll use here. I'll define a "routes" variable that will eventually be exported as the only value from this file. What is that routes variable? It's a function which will contain all of the routes and route handlers for this mini app. And as an argument it will take one parameter: the app. This is the same "app" we passed as an argument in "server.js".

Now our mini-app code is basically complete. It's self-contained and can work with any Express code as long as it's given an initialized "app" object. We could copy this directory to any other application and use it to provide a login form.

Or, it nearly is. There are several minor problems that need to be fixed before it will work. But let's give this a shot and learn how to interpret several kinds of errors in Express.

Back in the terminal, type "npm start". This will start the server. We see that npm barfs up an error message in brilliant black and red. There's no confusion that something has gone wrong here!

Most of this is not very helpful, yes it fills up our screen. Scroll back toward the top to see what happened. You can see it did run the command "node server.js". We see that an error was thrown, but it's very generic. Finally, there's something useful! It cannot find CoffeeScript.

Where is it? The "npm list" command shows the software installed in this application's immediate directory. "npm list -g" shows only the software that's installed globally. If you did not see the CoffeeScript error, you may have already installed it globally.

But I want this application's list of dependencies in package.json to be comprehensive. If someone has a fresh install of Node and types "npm install", they should get 100% of the libraries needed by this application. So let's install CoffeeScript locally and add it to package.json.

When I started using Node, I would do this as a two step process: install the library, note the version number, then manually type it into package.json.

Unfortunately, many Node developers still do that even though there's an easier way. It's not in the standard npm documentation, but that's what we're here for!

Install the "coffee-script" module with "npm install", but add the "--save" flag. It will install "coffee-script" to the "node_modules" directory and also add it to the list of dependencies in "package.json". So easy!

Open package.json and you can see that CoffeeScript as listed as a dependency, including the version number.

One caveat: if package.json has any errors, it won't be updated, and no error will be printed. For example, I accidentally used single quotes instead of double quotes in package.json, which isn't valid JSON. I was able to install libraries, but none were added to the list of dependencies. Now you know.

But in the best case, the "--save" flag is extremely convenient. Now you don't have to hunt around for version numbers. Let NPM do it for you automatically.

Let's give it another try. Run "npm start" and again we see an error. This time it cannot find the module "routes". That error is on server.js line 9. Let's look at that. In server.js at line 9 it's looking for a routes file that we have deleted and are no longer using. Delete that line. Start it again and finally the server will run.

Go to the "/login" URL in the browser. Now we see another error but this time it's not a compilation error from Node; it's an error from Express. It failed to locate the view template: "views/login". The first line that mentions part of our application (not the framework) mentions that it's an error in authentication/routes.coffee. And it shows line 6. Let's see if that's actually true. "routes.coffee" line 6 happens to be *nothing* related to view rendering. This file was dynamically compiled to JavaScript, so the line numbers in the error are from that compiled version. This is something that annoys many developers, but I haven't found it to be too much of a problem. The error message and file name are generally enough to help you get started. And not too distant versions of Node will be able to show you the exact line in the CoffeeScript source file.

Let's fix the bug and render the template. By default, Express will look for view templates in the "views" directory at the root of this application. To specify the local "views" directory in the mini-app, use the "__dirname" variable to get the full path to the current file.

Refresh the browser. Nothing happens. Why? Even when in development mode, Express won't load new versions of your code. If you change the code, you'll need to kill the server and restart it (or use a third party tool in development).

At the moment let's ctrl-c to kill the server, and restart it.

Finally, refresh the page. You'll see the rendered login form!

Chapter Four: The Layout Template

In this chapter we'll use the global layout template, link to stylesheets, and use styling to make the application look great.

Right now this form doesn't look very attractive. Let's modify our application's layout to use better styling.

We told Express to render a login template which shows just this form. But we'd also like to show more content along with that as part of the rest of the HTML document. That happens in a special view. I've chosen to leave the global layout in the "views" directory. The file is named "layout.jade".

Open "layout.jade". In the middle is this directive that mentions the "body" variable. This is where the login form template's content was rendered to. It can go on a separate line, nested within the HTML "body" tag.

On line 4 you can see the "title" variable used. In the browser, you can see that the window title is populated based on the title we provided in the URL handler.

But we also rendered the template with a "stylesheet" variable which isn't currently used anywhere. Let's use it.

Even though this is Jade and not CoffeeScript, we can use CoffeeScript-style string interpolation. And it even works for single quoted strings. Use the hashmark with curly braces, and type the "stylesheet" variable inside. Each page will now use an appropriate stylesheet.

Standard concatenation also works. To put "Hot Pie" before any page title, just add a descriptive string such as "Hot Pie" together with the "title" variable.

Let's do some other styling to match our prepared stylesheet. Add a section nested within the "body" tag that has a CSS ID of "header". Jade uses CSS syntax, even in HTML markup.

Within that, use an H1 tag that reuses the "title" variable.

Go back to the browser and refresh. Those modifications will give you a nicely styled page. Now we can continue development against an application that looks great.

You might also notice that we did not need to restart the server in order to do this. View templates are reloaded in development and don't require a server restart.

In the next chapter, we'll look at some tools that make development easier.

Chapter Five: Development Tools

In this chapter, we'll use a tool to automatically restart the server in development. We'll also learn how to define development dependencies separately from production deployment dependencies.

Even though we're only barely into the development of this application, we've had to stop and restart the server several times. That's going to get *pretty* annoying if we're doing a lot of development and have to hit Command-Tab, Ctrl-C, Up Arrow, Enter, Command-Tab every time we make a change. When talking to other developers about this, I was amazed at the lengths to which people went in order to get around this.

But it doesn't have to be that complicated. We're programmers, and we can automate this!

One option is to use a popular project for running Node servers in production: forever. This works somewhat like Munin or Runit where it will restart a server if it crashes. Very useful for a production server that should be running all the time.

You can also use this same tool to run against a configuration file that will restart your *development* server when code files are edited. But I'd rather not make a configuration file if I don't have to.

So after trying a few different tools, I found the node-dev package which does the same thing, but requires no configuration at all. It even works with CoffeeScript! I like it a lot. I've found it to be quite useful over the last few months.

A brief clarification: other development frameworks such as Ruby on Rails try to intelligently reload code files while keeping the server running. Node tools avoid that strategy and just kill the server process altogether before restarting it. Node is so fast that it usually takes only a half second to restart the server. Occasionally you might notice this delay, but it has rarely been a problem for me.

So let's install node-dev. As before, run "npm install node-dev --save". Look in the node_modules directory and you'll see the node-dev module. In that module, there's the "node-dev" command that can be used in place of node itself.

As before, my preference is to install all application dependencies locally and avoid system-wide directories altogether. Some people make an exception to this for modules that provide a command such as this one. But even these commands can be run from the application's local node_modules directory.

To run this directly, provide the path to the node-dev command. It tells us that it needs a script. That script is “server.js”. Now the server is running.

Try the reload feature. Open a code file, make a minor edit, and save the file. You can see that it restarted the development server. So easy! Use the standard Ctrl+c to quit.

Can we make it even easier? It would be nice to not type out this long command with the path. Here are two tips.

NPM installs local commands to the hidden ./node_modules/.bin directory. Add this relative path to your shell PATH and you’ll be able to run commands from the local directory without any qualifying path string.

Or, you could make this more convenient for your entire development team, no matter their shell configuration. There's nothing that stops us from making a “bin” directory in this project with a shell script that starts the development server.

Create a script named devserver. It will run this shell command. Make it executable. Now run “./bin/devserver” to start the development server in development mode without any arguments.

And what about development dependencies? When we asked npm to install and save node-dev, it added it to the generic “dependencies” list. But production deployments of this application don’t need node-dev. It’s only needed in development. So let's list that separately.

NPM has a section for that. Make a “devDependencies” section (be sure to use double quotes to be valid JSON). Move “node-dev” there and remove the comma from the previous line. This is also a great place to list test frameworks, which is exactly what we’ll do in the next chapter!

Chapter Six: Easy Testing

In this chapter, we’ll start writing a simple test suite. You’ll see how fast Node tests can be! We’ll use Mocha to write controller tests in CoffeeScript. We’ll give you a few tips to make it work smoothly so you can avoid the head scratching and hair pulling that we’ve had to do over the past few months!

Test-driven development can be a divisive issue. If you haven’t done it, it might seem confusing or worthless. If you have, you know it can give you guidance when developing and confidence when refactoring.

Once I got over the initial hurdles of tooling and libraries, writing tests in Node has accelerated my understanding of JavaScript, of CoffeeScript, and of Express. I think Node is the idea environment in which to write tests. They are ridiculously fast. The final suite of tests for this application runs in about one second. It's fast enough that I've rewritten my Git shortcuts to automatically run my Node test suite before every commit (it only lets me make a commit if the suite passes). And for the majority of the time that I was developing this application, the test suite ran in less than a second. So it's super fast.

The code to write tests against any Node web application is not complicated, but there's barely any documentation anywhere. It takes a bit to figure out how to do it, but once you do it's pretty easy. And that's what we want to show you in this chapter.

After trying several testing frameworks, I settled on Mocha. It's written by the prolific TJ Holowaychuk, the main author of Express and Jade. It's relatively new, but I like the informative error messages and overall design. And the syntax is similar to that of other test frameworks such as Jasmine. So if you decide that you need to switch to another framework, you could probably do it without too much trouble.

As before, let's install it. "npm install mocha --save". Open package.json and you can see it's been added to the dependencies list. But as in the previous chapter, it's only needed in development, so move it to the devDependency section. Be sure to clean up the trailing comma on the previous line.

Let's build a simple test to retrieve the login page that we've already implemented. Make a new directory named "test". Inside that, make a directory named "apps" to roughly mirror the directory structure in the rest of the application. Then make a new file which I will name authentication-test.coffee. Yes, we will write our tests in clean, readable CoffeeScript!

The rough structure of a Mocha test can take several forms. I like a combination of "describe" and "it" with the standard "assert" library. This allows me to easily group tests under a descriptive string such as "authentication". The "it" could target a specific URL handler such as "GET /login" and specific output such as "has a user field".

At the risk of becoming too complicated let's start by nesting the tests under a sub-description. Remove the "GET" and make another "describe" block that will wrap all the assertions dealing with the GET on the login page. This makes it easy to write several assertions against the same data.

If we want to verify that the login page has the user field on it, we'll need to retrieve a copy of the rendered HTML document. The way to do that is in a "before" function. For

efficiency, the “before” function will be run once when the “describe GET” function runs. It won’t be run separately for every “it” example that follows.

In some other web framework environments you have to go to great lengths to instantiate a controller in test mode, fake some kind of web request, and ask it to render output. In Node tests it’s much simpler. Require “server.js” which will create a webserver and start it on a specified port. We’ll use a Node HTTP client library to make requests against that server. Then we'll examine the output and see if it matches the expected output.

The technical term for this is an integration test.

The HTTP client library we’ll use is the “request” module, a popular piece of code written by Node core member and NodeConf organizer Mikeal Rogers. When calling the request function we have the opportunity to set some options. It will call a callback with an error object, a response object, and the body of the requested document (probably some HTML markup).

We will save the body in a variable for use by the “it” examples below.

Let's configure the options argument. CoffeeScript’s object data structures are useful here. It looks like the YAML data format. The URI is the document we want to retrieve: “localhost:3000/login”. We’re already passing these options to the “request” function, so there’s no more to do there.

Next, how do we make the body of the document available to the “it” examples below? I tried several convoluted and clever solutions until I learned that the solution is much simpler. We just need a “body” variable in the proper scope. In JavaScript, there is no scope but function scope. Each “describe” or “it” section starts a new function with new scope. So we need to declare a variable at the outermost level at which this data will be needed. Define a “body” variable below “GET /login” and set it to null.

In the “request” callback, I’ll name the parameter “_body” and assign it to the outer “body” variable. Assertions in subsequent “it” example sections can make assertions against the “body” variable. This happens over and over when using Node. You can do a ton without resorting to libraries or hacks. Just use language features to get stuff done.

One language feature that’s unavoidable is the asynchronous callback. The “it” examples need to run *after* the request has finished. One of the facilities Mocha provides for us is a “done” callback. When we've done everything we need to in the setup, just call “done” (with parentheses) and it will run the related assertions.

Let's try one of those. Uncomment the example for "has user field". I'm restricting myself to only the built-in assertions right now, but I want to test a regular expression against the body, the HTML document. So use the generic truth test "assert.ok" and run a regular expression that looks for the string "user" in the body. Not too precise at the moment, but it works for our very first test.

This is nearly ready to run except that we need pull in the external modules used by the code in this file. One of those is the "request" module. Another is the built-in "assert". And another is the application itself. You can get that by requiring the server file. As soon as we require the file, it will start the webserver as specified in "server.js".

A nice feature of the Express boilerplate is that it exports the "app" object from "server.js". The app object exposes useful configuration settings that tests can refer to.

We need to install the third party modules that are used by the tests. Install the "request" package. Move it to the "devDependencies" section since we don't use it anywhere in our implementation code – only in tests.

Since we've chosen to write the tests in CoffeeScript, we need to figure out a way to run those tests in CoffeeScript. Here is a trick I discovered. If you create a JavaScript file that requires the "coffee-script" module, and if you pass it to the "mocha" command before any other test files, then your individual test files can be written in 100% CoffeeScript.

I've named it "_helper.js" and I put it in the "test" directory. The leading underscore sorts it at the top of directory lists. Include the one line "require('coffee-script')".

Let's run this test. In node_modules/mocha/bin/mocha is the command to run the test suite. I want to run this helper file first, then any other CoffeeScript test files. Hit return and you should see green.

Chapter Seven: Test Support

In this chapter, we'll improve our test setup by enforcing the "test" environment, writing a script to run the test suite, and integrating it all with the "npm test" command.

The first part happens in helper.js. We want to make sure we're running in the test environment before we start running tests. Node defaults to the development environment, but you can set a NODE_ENV variable in your shell to change that. We want to be able to run tests against a test database or even run the test suite while the development server is running separately. That's not going to work too well with our current setup.

One of the few objects available to all Node programs is the “process” object. With it you can read or write environment variables. We’ll do that here with NODE_ENV, setting it to “test”.

I’ve collected some open source enhancements to the built-in “assert” module and have added a few of my own. Let’s require it here so all tests can use a more full-featured set of assertions, such as “hasTag” for matching HTML tags.

For clarity, specify the exact directory and path to this file.

Although someday it may be an NPM package, right now it’s a simple file. Copy it from the final project and paste it into the “test” directory in your application. Another reason I should make it an NPM package is that it has some dependencies. It needs libxml.js to help with parsing of XML and HTML, so install it. In package.json, move it to the list of devDependencies.

Now we can use one of these assertions. In our existing test, instead of the generic “assert.ok”, use assert.hasTag. The first argument is the HTML document; in this case that’s the “body” variable. The second argument is an XPath to the element we’re looking for. In this case it’s the “title” tag inside the “head” tag. Finally, the content of that tag. We expect it to contain “Hot Pie - Login”.

Run the “mocha” command and it still passes.

Another benefit of custom assertions is better error messages. I wrote these assertions such that if they fail, you’ll see a useful error such as “Couldn’t find XPath //head/title in the document.”

Running the test suite brings up another workflow issue that can be improved. Let's do the same thing we did with the devserver command; make a script to run it with all the proper arguments. In the local “bin” directory, make a file named “test”. Make it executable. Paste the entire test command as run previously. Try it out.

Running “./bin/test” is a much easier way to run tests consistently.

NPM has a “test” command built in, but we haven't told it what to do. For completeness let's tell it to run our test file. Go to “package.json” and add a “script” section. It should have a sub-key of “test”. The value is the shell command to run, which should be “bin/test”. Now you can run “npm test” to run the test suite.

For everyday use, I find the error messages from “bin/test” are easier to read without NPM's extra failure messages, so most of the time I choose to run “./bin/test” instead of NPM test. But given the fact that the NPM team built in a “test” command, I feel like it’s

good to support it. Continuous integration servers might use it, so we're ready for that now.

To conclude this chapter, let's add test environment configuration to the rest of the application.

Right now our server is running everything on port 3000. It would be a lot more convenient (and safe) if we run tests on a separate port. I often keep a development server running continually and want to be able to run the test suite without causing a conflict. So first let's use the configuration built into Express to set a variable in test mode. This is also a great time to learn about configuration variables in Express. The application will run on port 3000 by default. We'll use that variable when starting the server. Variables can be read from "app.settings" such as "app.settings.port".

Just as we have a "development" and a "production" section, we can add a "test" environment. Here, set the port to a different number such as 3001.

But if we run the test suite with that change, we get an error saying it couldn't find the title. In reality, it couldn't load the document at all because we've hard-coded the number 3000 in the tests. We need to make sure our tests use the configured port as well.

Instead of hard coding the port, we'll use the "app" variable which is conveniently available in every route-related test. Use app.settings.port to make the request.

Now run the test suite. It passes.

Planning around your test suite and building a few tools to manage your test environment will make it much easier to write and run tests.

In the next chapter, we'll move away from tests and back to the implementation of authentication and sessions.

Chapter Eight: Sessions and Flash

In this chapter we'll implement login and logout. We'll read form data in a controller. We'll learn about storing user data in the session using Redis. We'll read and write temporary flash messages for the user, and we'll look at how to test the output.

We have a login form, but it doesn't go anywhere yet. Let's make it work.

In the login template you can see that this form posts to the "/sessions" URL path. We need to implement that first. It follows the previous format of "app." HTTP method, URL path, and an anonymous function that has request and response as parameters.

We need to read the posted user and password data from the form. A full-featured authentication system would store this information in a database, run a one-way encryption on the password, and check the submitted password against that. For simplicity, I'll just check the name and password against a hard-coded value. The rest of this code will be accurate if you want to replace line 10 with a more secure authentication system.

In this system, there's only one valid user: piechef with password 12345. I like to run equality checks with the expected value first, then the variable it's being compared to. This catches syntax errors if I happen to assign a value ("=") instead of checking it for equality ("=="). It's less of an issue in CoffeeScript with the "is" keyword, which means "==" already.

Note that in Express we don't have access to all user input from a single object. The posted body parameters are accessed with "req.body". There are separate objects for querystring parameters and URL segments.

After a successful authorization, store the user's name in their session. We'll use this to verify that we're working with an authenticated user. The session is located in the request object. I'll create the key "currentUser" in the session.

Let's also tell the person that their attempt at authentication was successful. Express has a feature that I first learned about in Rails called the flash. It's a temporary message that will be displayed to the current user on their next request to the site. I like to mark it as either information or error. The message will be "You are logged in as.." with the name of the current user.

The last step in a successful login is to redirect the user to another page. But at the moment we don't have any other pages to redirect to. So let's redirect them back to this same login page. It's very important to "return" here so the application doesn't run any of the other code in this handler. You can only call "redirect" or "render" once in Express; you'll get an error if you try to redirect or render twice. So we're done with handling the successful scenario.

Handling error cases is important for any web application, and especially for an authentication system. Using these same concepts, what happens if the user enters an incorrect name or password? Set a flash message again, but this time with an error. Inform them that "Those credentials were incorrect, try again." Redirect back to the login page.

It took me a few tries to remember which API is in the request and which is in the response. Of course you'll quickly find out if you used the wrong one, but generally,

anything associated with the session, cookies, or the flash is in the request, and the response handles redirecting and rendering.

I've been cheating a bit here. We've used the session and the flash, but neither are built in to Express. We need to configure it to use a session and to tell it where to store session data.

Go back to server.js where we will add the necessary configuration and dependencies. Express ships with only the most basic of features. You can add other features as you need it. I like this philosophy. It means that my application starts small and fast. It's only going to grow when I choose to add more code to it.

Right after the method override, use the Express cookie parser. This is needed to store a single value on the client's browser: their session ID cookie. You should never, ever store important session data in a cookie on the client, not even their username. It's too easy for a person to edit their cookies and bypass your security barriers or just send you incorrect data. But we do need a cookie for the session ID so Express can match a user with their session data that we'll store here on the server.

Next we need to store this session information. By default it will use the memory store which only lasts as long as the server process. This will be frustrating to use in development mode since we're restarting the server often. You'll lose all your session data every time you save a file.

Instead, let's use the Redis database for sessions. We'll also use Redis to store the name and status of the restaurant's Pies. It's also middleware: "app.use express.session". It needs a secret key for encryption. The store is RedisStore.

But where does the RedisStore come from? Connect, the lower-level web framework used by Express, provides that as an add-on. The documentation says to require 'connect-redis' and pass it a copy of Express.

If you don't already have it, install Redis with homebrew. After installation, it will show directions for starting it.

The sequence of tasks follows what we've done before. Install the NPM package now.

Let's get redis, and also hiredis, a high performance adapter for Redis. We've seen connect-redis, the session adapter. As before, use "--save" so they will be added to package.json.

Let's try it. Start up the development server. Visit the login page. Enter piechef, 12345. You should now be logged in.

But how would we know? We're back at the login page. We don't see any flash message.

Let's add the flash message to the layout template so we can see the info and error messages. This will involve some new Jade syntax and also a new concept: the view helper method.

In the layout.jade template, we want to show both 'info' and 'error' messages. Jade can loop through an array of items. In this case, we want to use "info" and "error". The syntax in Jade is "each item in array". Then we'll see if there is a flash message of that type. We'll print out a paragraph element with a CSS class of "flash". The leading dot is for a CSS class. Set another class with the flash type (info or error). Then print the message with the equals sign. This emits a variable to the HTML output.

There's a lot going on here and several words are repeated in different ways. Let's take a look.

We have the flashType which is a variable. It's used as both a key into the flash hash and a CSS class in the HTML markup. The "flash" function is the helper method that we're about to implement. It's also a CSS class on the paragraph tag.

The end result is that we we'll print any flash messages to the output. This is in the global layout so it works for the entire application.

Now, the "flash" view helper method. In Express, we provide templates with variables that can be used as CSS classes or as content, but we also occasionally want to manipulate data or retrieve other data from the environment. Express doesn't provide that automatically, but we can do it easily.

Let's make a "helpers" file in the "apps" directory. I'll use CoffeeScript again. Use the same technique that we did in the routes where the code is wrapped in a function that is also exported. Exporting makes it easier to write unit tests against these methods.

Express offers two kinds of helpers. Standard helpers work independently with any arguments. Dynamic Helpers can access the request and response and can return parts of them for use in the template.

Since we want to use the flash object which resides in the request, we'll use a dynamic helper. The parameters are the request and the response. We don't have to call it with those; the framework sends them when we call the "flash" helper method from the view. This one is very simple. I only need to return "request.flash". It even works as a single line of CoffeeScript.

That's all it takes to get access to the flash with a dynamic helper in Jade.

As before, Express is explicit rather than magical. If we want to use these helpers, we're going to have to manually require them into our application. Go to server.js. At the bottom of the file, make a section for helpers. Require “apps/helpers” and pass a reference to the app to match the way we’re calling the routes.

Because we added a new file, restart the development server to make sure it will monitor changes in the new helper file.

Now let's try it again. Load the “/login” page. Initially we get a bunch of flash messages that were stored up in Redis from previous requests. Let’s try again with an incorrect name. The error message appears as expected. Try piechef, 12345. We have logged in. It works!

Let’s finish this with logout in the routes. Here's the code. To logout we're going to handle the “delete” HTTP method which is just “del” -- “delete” is already a reserved word in JavaScript so we can’t use that. If we send a delete on the “/sessions” URL, it should regenerate the session, write a flash message, and redirect back to some page, in this case the only page on our site, “/login”.

At first, I tried to use the session’s “destroy” method instead of “regenerate”. But that destroys the session for the remainder of the current request, so you won’t be able to write a flash message. So use “regenerate” to start over with a new session.

How about tests? A few ideas are relevant here. Peeking briefly at the final project, we can see a few things. To POST to “/sessions”, we set up some options as before. The URI we've already seen, but now we'll set up form data including the faked user and password as if it were being submitted from the form. This is how you post form data using the request module. It works great in tests.

I've also found that the followAllRedirects option is useful when you are posting form data in a test. Otherwise you'll just get the redirect headers back and you won't see the rendered form data from the page that was redirected to. I want to examine the output to see if the flash message was set, so I use followAllRedirects to get the final rendered page after a successful login.

I also used the technique of setting up a local variable for the body and the response. I can examine both response headers and the HTML markup in the body.

I'm also using the hasTag assertion that I've added in my assert-extra module. I'm using an XPath to specify that I should be looking for a CSS class of flash and error on a paragraph tag which has certain text in it.

To test logout, running a delete action on the session's URL is similar. As with Express, “delete” is a reserved keyword so the request module uses “del” to send the HTTP DELETE action.

Now using this kind of integration test approach means we can't actually examine the data in the session unless we query Redis and decrypt the session. However, we *can* look at the output that was rendered from templates and verify it is correct.

Conclusion

That wraps Part 1! We’re finishing Part 2 which goes deeper into building an administrative dashboard, implementing a data model, and using Socket.io to push events to the client.