Peep Code

screencasts

TRANSCRIPT FOR

Full Stack Node.JS Part 2

SOFTWARE USED

Node.js

Express.js

Mocha

AUTHOR

Geoffrey Grosenbach

GRAPHICS

Paula Lavalle

MUSIC

Giles Bowkett

APRIL 2012

TABLE OF CONTENTS

INTRODUCTION

It's PeepCode! This is Full Stack Node.js Part 2. We'll build on the concepts from Part 1 to build out the rest of the Hot Pie application.

In this screencast we'll build a data model that stores Pie data in the Redis database. We'll build an administrative dashboard for managing pies, protected by the simple authentication system we started building in Part 1.

We've been writing CoffeeScript and JavaScript on the server, but we'll extend that by dynamically compiling and serving JavaScript assets to the client.

Some people noticed that so far, we haven't implemented a home page. We'll do that here; it's readable by the public without authentication.

We'll finish the app by connecting the server and client in real-time. The public list of hot pies will be updated immediately when a pie comes out of the oven.

Let's get started!

CHAPTER NINE: PIE MODEL

In this chapter we will build a data model for storing and retrieving pies.

Because Express is such a minimal framework, it doesn't come with any kind of data persistence built in. You have your choice of databases to use, of libraries to use to access those databases, or you could build your own. Because our needs for this screencast are so simple, we're just going to

build our own persistence class in CoffeeScript which will save the data for a single pie to a Redis database.

As before, we have to decide where the files for the data model are going to be stored. I've decided to put my models in the root of the application's directory. Our `views` directory is already there, and data models will probably be used by several mini-apps. Any mini-app can `require` the model and use it from here.

But models will still be independent and can be tested apart from any other code in the application. This first model will be in a file named `pie.coffee`.

We're not practicing test-first development throughout this entire video, but I think it's relevant here. It will show how this model will be used from a URL handler in a mini-app's routes. And, it shows how quick and easy it is to test plain JavaScript classes with Node.js.

Make a directory in the `test` directory that mirrors that: a new folder named `models`, and a new file named `pie-test.coffee`. Let's setup the test to run. Load some libraries that we'll use. We'll need the built-in `assert` library.

We want to run the model tests along with the others, so add it to the test script. Go to `bin/test` and add all model tests. Use `test/models/*.coffee` in the test script.

Now for the actual test. We'll be describing a `Pie`. Specifically, we'll create a new `Pie` with some attributes. What's important about those attributes? We'll want to ensure that our implementation sets the name of the pie. The Pie should also start with some state which will be either *inactive*, *making* (which means it's in the oven), or *ready* (which means it's hot out of the

oven). We'll need some kind of unique identifier for each record.

But to do any of those assertions we first need a pie. Let's make one.

Use the `before` callback to setup the pie for use by other examples in this test. The `before` function will be run once, then the examples will each be run against the same object. Make a new pie with a name.

In the controller/route tests, we always had to call `done` when we were finished with all callbacks. In this case we don't need to call `done` because we aren't calling any asynchronous callbacks. However, we do need to setup the local `pie` variable so it will be available to our other examples in the `it` sections.

Before writing any assertions, try to run the test. There's already an error. Can you spot it? `"id is not defined."` We're not working with the `id` yet, but I misspelled `it`. If you're lucky enough to make only obvious errors, they will be easy to find and fix! My goal for this year is to only write bugs that are easy to fix. Not the ones that you have to stare at the screen for 30 minutes to figure out.

Run the boilerplate test code again to see if we get a better error. `"Pie is not defined."` Now this makes more sense.

We've used the `Pie` class but we haven't required it into this file. And we haven't implemented it either. Relative to this file, we'll use `models/pie.coffee`. Run the test again to see that it found the file, but we haven't implemented the `Pie` class itself. That's what `"object is not a function"` means. JavaScript objects are functions and no function has been defined that creates a Pie. Let's do that.

Make a CoffeeScript class named `Pie`.

Make this single `Pie` object available to code outside of this file. Use the same idiom `module.exports` to expose only the `Pie` class.

Run the test suite again. We're all green. One of the addicting things about test-driven development is the frequent confirmation that things are working!

Let's add a meaningful assertion so these tests verify the behavior of the `Pie` class. We would expect the new Pie's `name` attribute should match the name provided above: `Key Lime`. But that hasn't been implemented so we see an error like this: `Assertion Error: "undefined" == "Key Lime"`.

Part of the benefit of writing tests and learning in this way is learning what errors look like. I find that when learning any topic. Let's say you're learning to fix a bike. Knowing what a flat tire looks like helps in understanding what an operational tire should be like. It's the same here: understanding errors will help you fix them and also know when things are working correctly.

It expects to see `Key Lime` but got `undefined` instead. We need to implement a constructor that will set those attributes on the `Pie` object.

In CoffeeScript it's named `constructor`. It will take some attributes. For every key and value of the attributes, we'll assign that key and value to the new `Pie` object. I got into the habit of returning the current object, but it's not only unneccessary, it's also useless. Unlike other functions, CoffeeScript won't return the last value from a `constructor` method. It *will* return the new Pie object after running the `constructor` method on it.

At one point, I was super excited about mastering callbacks in Node and wrote every single function to operate on a callback, even `new`. After consulting with seasoned Node developers, I realized this was overkill. It's OK for objects to operate synchronously when being created. They shouldn't take a callback as an argument, only attributes or other values needed to set up the new object.

Run the test suite. All green.

The next test example describes the next feature to implement: assign a default state. The whole purpose of this application is to keep track of pies to show if they are gone, baking, or hot.

For that we'll need to start each new Pie with a state. We'll need to store the time they last changed state (useful for sorting hot pies over warm pies). While there are third-party libraries that can manage state machines in Node.js, our needs are pretty minimal. The JavaScript language is powerful enough that you can even implement callbacks with an EventEmitter without needing to use a third-party library.

So we can implement simple state transitions and timestamps directly in our CoffeeScript source code. Test that by writing an assertion that expects the initial Pie state to be `inactive`.

Run the test suite. There is a similar error as before: `AssertionError: "undefined" == "inactive"`. We have not implemented a default state yet.

Let's implement it. Instead of writing all the code directly in the `constructor`, make a method to set these and other defaults. If no state has already been provided, set the state to `inactive`. This leaves open the option to explicitly set a state in the initial attributes passed to a new Pie. If no state

is passed, `inactive` will be used.

Run the test suite. It passes!

Let's write one more test before we surge on to straight implementation. The Pie model should generate a unique `id` for each Pie. This `id` will be used in URLs, CSS classes, and elsewhere in the application. A full-featured database would generate this for us, but Redis expects us to provide our own key for records. In fact, that's a big part of designing data to be stored in Redis: desiging the keys by which data will be referenced.

The name of the pie is a good candidate for a unique key, but if it has whitespace in it, it won't be usable in URLs. So as a simple conversion, we'll expect that the `id` will be the name of the Pie, but with a dash in place of whitespace: `Key-Lime`.

Run the test suite to confirm failure. As expected right now, the Pie doesn't calculate the `id`, so it fails.

Go to `pie.coffee` and implement this in the `setDefaults` method.

We'll put the code in a method named `generateId`.

Here's how it works: If we don't already have an `id`, and if we do have a `name`, then the `id` will be the name with each whitespace character replaced with a dash. The `g` flag on the regular expression instructs it to operate on all the whitespace in the string, not just the first one. Set that generated value as this model's `id`.

It should pass the test suite.

Writing tests for each feature in this application really helped me under-

stand many parts of this code with minimal frustration. If something went wrong, at least I had a consistent error to refer to and a quick way to try it again.

However, for a video like this it can sometimes get in the way of learning about the actual application. So for the rest of this chapter, we'll be working directly with the model code to implement a `save` method.

We'll save the data in Redis using the `node_redis` module. The GitHub repository is at `mranney/node_redis`, but the `npm` package is just called `redis`. The documentation is brief. I'll show you how to use it for a few commands, which will make it easier to understand how the rest of the commands work.

But most of these commands aren't specific to this Node client. They map straight through to commands in Redis. So it's important to understand a bit about how Redis stores data.

Here are the three core concepts to take away from the `redis` module:

- You will create a client that connects to a Redis server.
- You will run a command against Redis with one or more arguments (usually a key name to fetch or a key and value to store).
- The final argument to any Redis command will always be a callback. The callback's first parameter is an error, and the other argument is the data.

Given that we're covering several topics in this screencast we won't go into depth about Redis. But you do need to understand one Redis data type: the `hash`. The documentation at http://redis.io is a useful quick reference.

A Redis hash is complex data structure stored under a single key. We'll use

`Pie` as the key for storing all the individual pie objects. Each key has sub-keys (the id of each pie such as `Key-Lime`) and values (the attributes of each pie).

Redis commands all follow the same format. A single letter that matches the data type (`h` for hash), then an action such as `set` (to save), `get` (to retrieve), or `del` (for delete).

Put that all together and you can build the command to set a value in a Redis hash: `hset`.

To retrieve all the pies there is `hgetall`. Because we're working with a limited number of pies – probably no more than 50 different kinds of pies for a single restaurant – this will work well.

So let's use these concepts to see how they work. We have already installed `redis` but we need to use it in this model. Make a Redis client with `createClient` and let's write the `save` method. The `save` method's last parameter is a callback so other code can run after the `save` has completed. Let's first call our `generateId` method to make sure we have an `id` to work with. Then use the Redis `hset` command to set a value in a Redis hash. The `hset` command takes three arguments: `key`, `field` and `value`. The key is a string that identifies all the pies. The word "Pies" works well.

I don't think there's a standard for separating development, test, and production data in Redis, so let's separate them with the key name: `Pie:development`.

The `field` is the `id` we've generated. For the `value` I want to store all the model's attributes. But the `value` in a Redis hash is a string. That's easy to work with: just serialize this Pie model to JSON and store that as the

`value`. That takes care of the data that we're sending to Redis.

The callback gives us an error object and response code. I'll take the simplest solution here by just calling the callback with `null` for the error and the current Pie object. To do this in CoffeeScript, use the fat arrow so we can refer to the current object with the `@` symbol.

For a robust application, you should check for a non-null error object or even examine the specific `responseCode` returned by Redis.

That's all we need to save our model to Redis.

Let's try this out briefly in the console. Type `node` to launch the interactive JavaScript prompt. `require('coffee-script')` so we can use CoffeeScript source files. Get a reference to the `Pie`. Make a `new Pie` which will be `"pecan"`. You can see it sets some default values. Then `save` the pie. You can see that the pie was saved. Hit `Ctrl+c` to quit.

Let's clean up one thing before finishing this chapter. When saving a single Pie object to the hash of all pies, we're using the hard coded string `"Pie:development"` as the key. It would be much better if it used the a value from the environment: `development`, `test`, or `production`.

This value will be used by all Pie objects and may even be useful in tests. So let's make what's basically a class method. It will be called directly on the Pie object, not on single instances. In CoffeeScript, prefix a method name with the `@` sign to make it a class method. The value is the same `Pie` string, but with the value of the `NODE_ENV` environment variable. One of the few objects available to every Node script without an explicit `require` is the `process` object. `process.env.NODE_ENV` gives the value of the `NODE_ENV` variable.

Express supports `development`, `test`, or `production` environments but it's sometimes up to you to use them.

As an example, look at the final test for the `Pie` model. Scroll to the bottom of the file where you can see I've implemented a database reset by implementing an `afterEach` callback (supported by Mocha). After each `describe` group, I want to delete all Pies so the next test can run against a fresh database. I use `redis.del` to delete the `Pie.key()`. This deletes all Pies. Then we can build, create, or otherwise save pie objects and make assertions against them without having to worry about existing data that could confuse the tests.

Check out the final project for a full test suite for the `Pie` model.

Here's a slight optimization to consider in your own code. The `key` method doesn't really need to be a function. It could just be a value. If you choose to do this, you'll need to remove the parentheses from any place that `Pie.key` is referenced.

To make things easier in future chapters, here's a task for you. You've seen the command to delete all pies from the database. Write a `deleteAll` class method on the `Pie` class that deletes all Pies from the database. It will be useful to call from the console when we need to reset the development database, or even to use in tests as you see here.

In the next chapter, we'll use this `Pie` model in the administrative interface.

CHAPTER TEN: ADMIN PIE FORM

In this chapter we'll start building out the administrative dashboard. We'll use some features of Express to build URL handlers. We'll build an HTML form to create a new pie.

A big part of designing any web application is figuring out what the URLs should look like. I like Express because it's super flexible and, honestly, indifferent to how your URLs will look. You can use REST-style URLs with a resource and an `id` or you could use something completely different such as GitHub-style URLs that only show the unique values used to retrieve a record.

I've designed our administrative dashboard such that its URLs are nested under the `/admin` path. In fact, most of the URLs in this applications will be prefixed with `/admin`. The public portion of this site is a single page. All the rest of the functionality is used only by administrators (or more specifically, pie chefs). So even things like creating pies will be nested under `/admin`.

To reduce duplication, we'll use an add-on to Express to make this happen. It was written by one of the authors of Express: `express-namespace`.

`express-namespace` augments Express with a `namespace` method that wraps other route definitions. You can use any kind of path descriptor that Express supports. So instead of prefixing all our URLs with `/admin`, we can do it once around all of them.

Get started by installing the module: `npm install express-namespace --save`. It's now listed in our dependencies in `package.json`. In `server.js`, require it on a single line. There aren't any arguments and the result doesn't need to be saved to a variable. It modifies Express in place to

add the `namespace` method wherever Express is used in the rest of our application.

Let's use it in a mini app. Let's create a new mini app for our administrative routes. Name it `admin`. Add `routes.coffee` inside for the URL handlers.

And while you're here, make a new `views` directory for administrative views: forms, pie detail pages, and state changing screens.

In `routes.coffee`, start writing the controller code. We'll use the `Pie` object we implemented in the last chapter, so require it here. Use a relative path to the `models` directory.

For the module boilerplate, use the technique shown in the login screen to take the `app` as an argument and export it as a module. The final setup step is to go to `server.js` and require it so the server runs it when we start the server. Now we're ready to write some code.

As mentioned at the beginning of the chapter, `express-namespace` will help us create an `/admin` path that prefixes all other administrative URL paths. This is one of the reasons I love CoffeeScript; it works naturally with these kinds of APIs.

Call `app.namespace` with `/admin` as the first argument. The second argument is an anonymous function. Now we can use Express normally: `app.get`, `app.post`, etc.

The first admin route I want to implement is a listing of all pies. It will be located at `/admin/pies`. Individual pies will be located a subpath with an `id` such as `/admin/pies/Key-Lime`. Do you see any repetition here?

We need another namespace for `/pies`. Fortunately, `express-namespace`

can handle that, too.

Add another namespace for `/pies`.

Now let's implement a path with a handler that will show all pies. This uses the Express concepts that you learned in the first video. The path has been whittled down to just `/`. It takes arguments for the `request` and `response`. I want to render a template with all the pies in the database, and also show a form for adding a new pie.

Let's start with the form since it's the easiest. Make a new empty `pie` that will be used to populate values in the form. Then render the `views/pies/all` template. Let's give it a few attributes: the page title, the stylesheet to use, and the empty `pie` object. All standard stuff.

Now make the template: `all.jade` will eventually render all the pies but to begin with it just needs to render a form. The pie form might be useful to other views, so this is a great opportunity to learn a special syntax to render another template inside this one: it's called a *partial* template.

When printing plain variables to the template, we used the equals sign. But this partial template will include HTML markup. By default, Jade escapes output for safety against JavaScript-based hacks. So use the alternate `!=` to render the form as HTML content instead of turning it into `<` and `>` escape characters.

The form template resides in the same directory. It uses a naming convention where partial templates start with an underscore: `_form.jade`.

Express doesn't ship with any special syntax for building forms, but Jade provides most of what you need to create an HTML form. It starts with a

form tag which will be posted to some URL. Use the `post` method for the form. It has two fields, each with a `label`. Plain text inside a label uses the pipe syntax: `| Name`. The first `input` is a `text` field. It stores the `name` of the pie.

Following that is similar syntax for a `label` that holds the `type` of this pie (sweet or savory). We only have two options, so I'll hard code that into the form as a select box.

The final element in the form is a `submit` button.

Let's try to run it. Start the dev server and go to `localhost:3000/admin/pies`. The trailing slash is optional. We see an error: `"failed to locate view"` and it gives us the entire path: `"views/pies/all"`. Can you spot the error?

Among the configuration of paths and namespaces, I told Express to render templates from `views/pies` but neglected to put these views in a subdirectory. Let's do that. Make a `pies` subdirectory and put these two files in it. Now it matches the path we specified when we asked it to render the template.

Because we only changed templates, not application code, we don't need to restart the server. Refresh after rearranging the templates and now you'll see a nice form with which we can create new pies.

We left something unspecified in the form. What URL will it submit the form data to? In the absence of a better idea, I like to use REST style URLs: submit new pie data to `/admin/pies`. Existing pies should update themselves at a similar URL with their `id`: `/admin/pies/Key-Lime`. But I don't want to have to hard code this into the form. It would be nice to use

a helper method to generate the URL. Ruby on Rails makes it convenient by building these kinds of methods into the system, but it turns out to be pretty easy to build your own URLs, given a bit of information about the model. It's also a great place to learn how to write methods for use in view templates.

Let's use it in this form as if that method has been written already. I've gotten into the habit of working this way after doing test-driven development; I like to see how an API will be used before I implement it. I'll call the helper method `urlFor()`. It will take the new `pie` object as an argument.

We've already seen one kind of template helper. Let's write a different kind that's actually simpler. A dynamic helper had `request` and `response` parameters. But there are also have standard, unqualified helpers which can take any parameters. They don't have access to the `request` or `response`, but we won't need either here. The `urlFor` helper method takes a single object. I'll use that object to build a URL.

In general REST style, I'd like to say that if there's an `id` property on the object, it will submit to `/admin/pies` with the object's `id`. If there's no `id`, it will just use the collection URL for this object: `/admin/pies`.

That's easy to do in CoffeeScript. If there's an `id`, use it. Otherwise, return `/admin/pies`. The `return` keyword isn't needed; CoffeeScript will return the value of the conditional since it's the last part of the method.

I've only partly solved the problem of hard-coding URLs. As your application gets more complicated, you could build out this method to know about the class of the object being passed and use that in the URL. But for this application, it works great.

You can see that the development server automatically restarted itself when we edited those files. Refresh the page and view the source. The form does show the `/admin/pies` URL

Let's finish up this chapter by saving the pie's data.

Back in the admin routes, add a handler for a `post` to the `/admin/pies` route we just wrote. If someone posts to the `/`, we'll get the standard `request` and `response` objects. Let's build some attributes to be saved to the model. Get the name from request.body.name. As in the login form, values are posted in the request body. Do the same for the type.

Make a new pie with those attributes and save it. If it was successful, send an informative flash message to note that the pie was saved. Then redirect back to `/admin/pies`.

Try it out! The development server restarted itself. We'll make a pie for tarragon chicken which happens to be a savory pie. Submit the form and we can see the pie "tarragon chicken" was saved.

In the next chapter we'll add some authentication protection around these routes.

CHAPTER ELEVEN: ADMIN AUTHENTICATION

Now we can create pies but unfortunately so can everyone else! In this chapter we'll protect administrative routes with our existing authentication system. Only authenticated administrators will be able to access the dashboard.

Let's make a fresh start by resetting our session. All we need to do is delete the session id cookie in the browser.

Load a page and pull up the developer console. Go to Resources and find the Cookies section. Under localhost you'll see a session ID cookie named `connect.sid`. We're not currently rendering anything to the template to show that we are logged in, so this is the only indication, currently.

Select it and hit the `delete` key to delete this cookie. The server no longer recognizes me. I'm anonymous. But we haven't implemented any protection for administrative pages, so I can still view the `/admin/pies` page. Let's fix that.

What *should* happen if I try to view an administrative page right now? I should be redirected to the login screen and then back to this specific page after providing valid credentials.

I remember the first time I read through the API documentation for Mac OS X. I kept thinking, "Wow, they thought of that?" Everything was planned out and organized. I got the same feeling the first time I read through the documentation for Express. Almost anything you need to do with routing is possible. And sometimes, there are several ways to do the same thing.

Let's use a technique that I think is a clear way to protect access to any resource that's a subpath of `/admin`. So far, we've looked at the `request` and `response` arguments to every URL handler. But there's also a third parameter which is traditionally referred to as `next`. If a URL handler decides it doesn't want to be the final supplier of content for a URL, it can call the `next` function. Execution will flow down to the rest of the handlers

in the application to see if any others match.

We're going to use that feature to guard this entire namespace. Define a URL handler that will work generically for all URLs related to the `/admin` path (). Nest it under the `/admin` namespace. Use the all method to intercept all HTTP methods: `get`, `post`, `put`, `delete`, etc. Express understands a shell-style glob: `*`. So any URL paths starting with `/admin` will go through this handler first.

Now write the logic to check for an authenticated user. If there is no `currentUser` in `request.session`, then set a flash `error` message asking the person to login. Redirect back to the `/login` path. It's very important to `return` so none of the rest of the code in this method is executed.

At this point, we're done with the protection part. But if we stop here, even authenticated users will be stuck without anywhere to go. Here's where the `next` function comes in.

If the person is currently authenticated using our simple authentication system, call the third parameter: `next()`. Be sure to use parentheses so it's called as a function. Express will move on to process the current request with any of the other of the routes on the `app` that match.

With that, we have created a checkpoint so only authenticated users will be allowed to view and create pies.

Let's try it. Go to this URL now. It takes us to the `/login` page and asks us to log in.

We wrote the login system before we had any other pages, so on success it will redirect us back to the login form. Let's fix that. Go `to authentica-`

`tion/routes.coffee` and change that to redirect to the `/admin/pies` path.

Try it. Enter incorrect credentials and you'll be shown the login form again. Or, enter the correct credentials: `piechef`, `12345`. You'll be taken to the `/admin/pies` page with a confirmation message.

Here's a challenge for you: A full-featured system would store the original URL that was requested and would send the user back to exactly that URL after successful authentication. Use the existing `sessions` functionality to store and retrieve this information.

There's an alternate way this could have been implemented: a middleware function. Take the exact body of the URL handler and put it in its own function. You can then pass this function as an argument to any URL handler and it will be executed before the handler is run. Multiple functions can be passed this way.

I still think the URL handling version we implemented is cleaner. You define the handler once and don't have to remember to add the extra function as an argument to all the other handlers that should be protected.

Here's one last useful feature. Middleware can set properties on the `request` for use by other URL handlers. So if you needed to fetch a user object from a database, you could do that in the route protection method and set it on the `request`. Subsequent URL handlers would have access to it. Express gives you many options for designing your application's URLs.

CHAPTER TWELVE: VIEW ALL PIES

In this chapter we will display all pies on the administrative dashboard.

We'll add a method to the Pie model to retrieve all pies. We'll use that method in a URL handler. We'll enhance the existing template to display the name and type of each pie.

We've implemented a form to create new pies, but there's nothing to show us what those pies are. We already have most of the code to make this happen. When we get the root URL for `/admin/pies`, we should load a list of all the pies in the database. The syntax I will use is `Pie.all`. It will execute a callback with the standard Node error and the list of pies. The template shouldn't be rendered until the data is available, so move the call to `render` inside the callback. Now that we have some pie objects, send it to the template under the `pies` key. That completes the URL handler.

But we haven't written that `all` method yet, so let's write it. Go to `pie.coffee`.

Write a class method named `all` that takes a callback. Prefix it with `@` so it can be called directly on the `Pie` object. The relevant Redis command is `hgetall`. It will get all the objects in a Redis hash for the specified key. Those pies are stored under our Pie key. The callback's parameters are the standard error and an array of objects.

But these aren't Pie objects yet. In fact, they aren't even JavaScript objects. They're plain strings; serialized JSON.

Our aim is to make an array of Pie objects from these bits of JSON. So start with a blank array. CoffeeScript has an iterator that works for key-value objects. In fact, that's the syntax: `for key, value of objects`. The key is the pie's `id`, the value is the JSON formatted string.

Now we can make our own Pie objects. Make a new Pie where the attri-

butes are the unserialized JSON: `JSON.parse`. Add each `pie` to the array of all `pies`.

When it's all done, call the callback function with a `null` error and the array of `pies`.

We haven't implemented the template, but try it in the browser to make sure it works so far. You should see no errors.

Let's go to the template and render the array of pies. The template already exists as `all.jade`. To match the existing CSS, make an unordered list with the DOM id `listing`. Using Jade syntax, I'll iterate over `each pie in pies`.

Make a list item whose content is the pie `name`.

For styling, add a CSS class with the pie `type`. We can't do that with the syntax we've used so far. Use parentheses and specify that this is a `class`. Populate it with the pie type.

This leaves us with a mix of HTML markup, Jade looping directives, tag attributes, and dynamic object values.

Go to the browser. Refresh the page. We see the pies we added earlier! Go ahead and add another pie, such as "Apple", a "sweet" pie. Hit submit. You'll see that "Apple" was saved to the database and is styled with the proper icon.

If that *didn't* work, you might have old data in the database. Here's an easy way to reset it. Use the Redis command line tool, `redis-cli`. Use the familiar `hgetall` with the `Pie:development` key to see the existing data. Delete all the pie data with `del` and the same `Pie:development` key.

You should be back to a clean state. Go back to the dashboard, add some pies, and view them in the new template.

CHAPTER THIRTEEN: ADMIN VIEW PIE STATE

In this chapter we will put together the pie status page, use some helpers, and start integrating some client-side JavaScript.

Here's the final application. One of the most important pages of this site is the pie status page. Here we can change the pie from being in the oven, to ready, to marking that it's gone (consumed by hungry customers).

Because we've already gone through most of the concepts involved here, I want you to try to implement part of this on your own.

Here are the requirements:

- The URL is `admin/menu/stage`
- The title is "Pie Status"
- Use the `admin` stylesheet
- Fetch all the pies from the database and send them to the template
- Render a view template

I'll help you out by giving you the view template. Initially, it will look almost exactly like the template we just wrote in the last chapter.

Under `views`, make a folder named `menu`. Make a file named `stage.jade`. It will list the pies with their current status. An unordered list will display the pies. Each pie in the list of pies will be displayed as a list item with a class matching the `pie.type`. The content is the `pie.name`.

Your job, is to write a URL handler that will render this template with the appropriate pie data. Go try it.

Ok, let's look at how I implemented it. Indented about four spaces, there is another namespace for `/menu`. A `get` handler serves the stage page. It fetches the pies. It renders the `stage` view. It has a title and a stylesheet. It exposes the list of `pies` to the view template.

Try it out. Visit the URL `/admin/menu/stage` (you may need to login). You will see a list of all the pies. This looks eerily similar to the page at the end of the last chapter. Let's improve on it.

Our goal is to properly display the status of each pie, so we need to render individual, clickable elements for each of the three possible statuses: `making`, `ready`, or `inactive`. We'll use a view helper to consolidate the view logic and reduce duplication of code.

Go back to the `stage` template and add a `div` that will hold the three status buttons. Each button will be a `div` with a special class matching the pie's state. The contents are a paragraph tag with a descriptive name: `gone`, `oven`, or `ready`.

Instead of littering this template with the logic for figuring out what CSS classes to apply, let's write a helper method. This `div` will use it: `cssClassForState()`. Arguments are the state it represents, such as `inactive`, and the pie's current state. Lather, rinse, repeat with `making` and `ready`.

It's time to implement the `cssClassForState()` helper. This needs to return the name of a Pie state and, conditionally, the word `'on'` if the button being drawn matches the curent pie's state. To do that, this standard helper needs two parameters: an `expected` state and the `actual`, current

state of the pie. If the `actual` state is the same as the `expected` state, it will return an array that includes the `expected` state and the word "on." When rendering the template, Jade will turn this array of strings into a pair of CSS classes, separated by whitespace. In an early draft of this application, I did that myself. But it turns out that it's unneccessary; Jade can do it automatically. If the `actual` and `expected` don't match, return only the `expected` class name.

Now the visuals for this interface are ready! Go to the browser. Refresh the page to see the various buttons, which are ready to show each pie's state. Unfortunately, all the pies use the default state, so you won't see any differentiation between the pies yet.

In the next chapter, we'll implement server and client-side code to mark that a pie is being made or is hot out of the oven.

CHAPTER FOURTEEN: ADMIN UPDATE PIE STATE

In this chapter, we'll implement the server-side code needed for the user to update a pie's state. We'll implement a URL handler to expose the API. We'll add a list of states to the Pie model. We'll build a model method to fetch a single pie record. We'll implement a simple state machine.

Creating data from scratch is, oddly, one of the easier parts of building a web application. Updating a small part of that data usually involves more code, especially when we're building a JavaScript-driven interface such as this one.

Let's start with the API that will be exposed by the server. Go to `admin/`

`routes.coffee` and add a URL handler under `/pies`. First, we'll need a pie `id` in order to find the correct pie to update. I'll use the HTTP `put` method here to follow a standard REST practice and distingush it from records created with `post`.

This is the first URL we've written that includes a variable parameter in the URL definition. Preface the name of the variable with a colon: `:id`. The full URL path to this action is `/admin/pies/:id`, such as `/admin/pies/Apple`.

This handler receives the standard `request` and `response` parameters. As before, let's use the eventual model method that we'll write so we can get a sense of how they're going to look. We need to retrieve a pie record by its `id`. The `id` is pulled from the `request` parameters, separate from the form body or the querystring.

The methods we'll add on the Pie model will work like this. They'll expose a method on each `pie` object that matches the state we want to switch to, such as `ready`. It will take a callback that executes after the pie has been updated in the database. I'll send a simple text response: `OK`.

The catch is we're going to have to do this dynamically using information passed as form data. So let's use JavaScript's alternate property syntax: square brackets. The actual state being switched to can be found in the form parameters: `request.body.state`.

However, this convenient syntax also exposes a security problem: people should not be able to call absolutely *any* method on this object. They should only be able to call one of three methods matching the possible model states: `inactive`, `making`, or `ready`. One way would be to control this

within the model. Another way is a verification right here in the controller.

If the proposed state is already in the list of states that the Pie model knows about, switch the pie to that state. The `in` keyword looks like it could be CoffeeScript, but it's actually a built-in feature within JavaScript and it definitely works in Node.js.

If it didn't work we could render an error template with appropriate arguments.

That's the API we want to work with from the controller. Now we're ready to implement it in the model.

Go to `pie.coffee`. Start by defining a list of possible model states. We'll use this list in several places around the application. It's a list of the three possible states a pie can be in: `inactive`, `making`, or `ready`.

Next, we need to be able to grab a pie by its `id`. Write the method called previously: `getById`. This method's first parameter is a Pie `id` to look for. The second is the familiar callback to execute when the information has been retrieved.

Redis has a command built specifically for retrieving a single object from a hash: `hget`. We'll again use the `Pie.key()` to access the hash of pie objects using the proper development, test, or production environment. The `id` will be used to retrive one specific pie and then Redis will give us either an error message or JSON data for that pie.

Here's an opportunity to use an error callback. If the JSON data is null - for example if the pie doesn't exist - then we will call a callback with an Error object. Don't ever pass a plain string as an error. Use `new Error` and a

message such as `"pie could not be found"`. This guarantees the maximum compatibility with other Node applications and is just good behavior. If there was an error, `return` before the rest of the code that will handle the successful case.

If we do have a proper JSON string, then we'll make a new Pie with the parsed version of the pie's attributes. Trigger the callback with the `pie` as the second argument. Now we have a `getById` method that will be useful anywhere in this application that you need to retrive a single Pie record.

The last bit of model code needed by this application is a state machine. A state machine can be as simple as a single field that records what an object is doing right now. Or it can be more full-featured and trigger events before and after each change of state.

I had fun thinking about and implementing this state machine. An early iteration had a full notification system using Node's EventEmitter, but this version only needs to change the `state` attribute and store the time on each change.

When the pie object is initialized, we need to dynamically define state machine methods for `inactive`, `making`, and `ready`. Group that code in a method named `defineStateMachine()`. Because JavaScript is such a dynamic language, we can do this in a pretty cool way. We're going to loop through the states we know about and build up a method for each.

We need a feature of CoffeeScript that I rarely use, but is perfect here: `do`. The `do` keyword defines an anonymous function and also executes it. It's use here has to do with scope. As mentioned in the foundational book *JavaScript: The Good Parts*, there is no scope but function scope. That is,

the only way to protect variables from outside modification is to start a new function. Loops don't create scope; conditionals don't create scope. Practically, it means that if we start defining methods without creating a new scope for variables, we'll only have a state machine method for the *final* iteration through the loop. Using `do` ensures that each iteration locks in a scope and it works correctly. This bug bit me several times while developing this application, and now I understand the concepts much better.

Inside `do`, I want to define a new method on the current pie object. So use the fat arrow to retain proper references to the current pie object when we use the `@` character.

I need to create a method for each state that does the following: take a callback as the only parameter, set the pie's `state` to the specified state, set a new property `stateUpdatedAt` with the current time, save the pie, and finally call the callback.

Now I have three methods: `pie.inactive` that does all this for the `inactive` state, `pie.making` that does it for the `making` state, and `pie.ready` that does the same for the `ready` state.

Take a deep breath. That took some of hard thinking! In the next chapter we'll use this API from client-side JavaScript.

CHAPTER FIFTEEN: CLIENT SIDE ASSETS

In this chapter, we'll use JavaScript for the first time on the client. We'll use middleware to compile CoffeeScript for the client. We'll use special

directives to bundle jQuery and other assets together. We'll reference the bundle from HTML markup. And we'll execute JavaScript on the client to update the Pie state.

One of the appeals of Node.js is the simplicity of using the same language on both the server and the client. Traditionally that meant JavaScript, but it's almost as easy to use CoffeeScript in both places.

One way to easily compile and serve CoffeeScript to the client is through a third-party module written by CoffeeScript master Trevor Burnham. The `connect-assets` module compiles CoffeeScript dynamically and also giving us options for listing all our client-side dependencies so they can be served as a single file.

In the terminal, install `connect-assets` and save it to the `package.json` manifest.

Following the directions, we will also add it to our configuration. Go to `server.js`. In the global configuration, add `app.use` to require `connect-assets` and immediately call it as a function. Now we have the ability to compile CoffeeScript (or JavaScript) out of an `assets` directory, `js` subdirectory.

Libraries such as jQuery or Underscore can be put here, or our own code such as this `application.coffee` file. I recommend that you copy this from the final project and paste it into your application. It will give you jQuery, Underscore, and the rest of the client-side code you need for this application.

What does that client code do? Go to `assets/js/application.coffee`. You'll see two directives in comments that import jQuery and Underscore, which

will be concatenated together with the rest of the contents of this file.

Skip past the beginning section and you'll see that we use jQuery to handle a click on a status item. It does some backflips to get the `id` of a pie and the `state` to change it to. From this it builds a URL that will run a `put` to the server with the new state data.

We'll the database by making an API call and the appearance of the user interface by modifying CSS classes on the list of pies.

A crucial part of this connection between client UI and server API is the `id` of the pie. It's stored in an HTML5 `data` attribute. Go to `views/menu/stage.jade`. Add a `data-id`, which stores the pie `id`.

However, all this JavaScript isn't going to do us any good unless we deliver it to the web browser. That's done in the global `layout.jade`.

At the bottom of the `body` section use a method provided by `connect-assets`: `js()`. This returns an HTML tag, so use the `!=` syntax to render it unescaped. It will build a script tag pointing to the compiled, combined version of `application.coffee` that we've just been working with.

Back in the web browser, refresh the page. There's no visible change, but you can see the effect if you view the page source. Since we're in development mode, separate copies of jQuery, Underscore, and the compiled `application.js` have been delivered to the client.

The buttons should also work. Click to mark that the "Chicken" pie is in the oven. There isn't any immediate confirmation other than the highlighted icon. However, refreshing the page shows that the change stuck. It saved the data and we've successfully updated the pie!

In this chapter, we'll build a public-facing page that displays pies that are in the oven or ready to eat. We'll make another mini-app for this functionality. We'll add to the Pie model to retrieve a list of only the active pies. We'll write a view template and make a view helper function to style the list of pies.

So far we've spent all our time on the administrative dashboard. But the purpose of this application is to show a list of pies to potential customers. So it's a good time to create a final mini-app for the public-facing home page.

What should we call it? It could be `public` or `index` or `home page`. Boring! This page will be displayed on a tablet-type screen at the front of the restaurant, facing the sidewalk. So how about calling it `sidewalk`? It's more descriptive and talks about the purpose of the mini-app rather than the technical implementation.

So create a directory named `sidewalk`. Follow our convention of storing the routes in `routes.coffee`. Type the basic boilerplate that accepts an `app` parameter and exports these `routes`.

We'll implement a URL handler for the root page of the application: `/`. Previously, we were interested in all the pies. But now, we don't want to show pies that are `inactive`; neither being baked nor ready to eat. They are technically on the menu but don't need to be shown to the public. It would just be confusing.

The as-yet unimplemented model method to retrieve this filtered list will

be `Pie.active`. Once we get an array of suitable pies, we'll render a local template with a `title`, a new `stylesheet`, and the array of `pies`.

Let's implement the model method. Go to `pie.coffee`. The `@active` method is again a class method. If there were a huge number of pies in the database, it would be more efficient to craft a custom query of some sort, but it's entirely sufficient to reuse `Pie.all` and filter the list from there. In fact, it's super easy with CoffeeScript. We can do it in a single line.

We want the pie if the pie in the array of pies has a state that isn't "inactive". We didn't even need `Underscore.js` to do that, just the CoffeeScript language.

Execute the callback with the new filtered list of active pies.

Now we have a route and we can retrieve the data. Let's implement the view template.

Make a `views` directory. Create a template named `index.jade`. As before, start with an unordered list that displays the name of each pie. Each pie is displayed as a list item. Each list item has a CSS class matching the pie's `type`. The content is the pie `name`.

We've created a new mini-app, so add it to `server.js`. Copy the existing `admin` mini app. Change it to `sidewalk`.

Will it work? I'll let you start the server in the terminal. For the first time, visit `/` in the browser. An error! `Reference Error: Pie is not defined`. Can you figure it out?

In `sidewalk/routes.coffee`, we used the Pie model but never required it in that file. You know the line of code. Require it with a relative path to the

`models` directory.

Try it again. A pie is shown!

It even works - somewhat - from the admin dashboard. Open a separate browser window to `/admin/menu/stage`. Change the status of a pie to something other than `gone`. Refresh the home page and that pie appears in the list, too.

Let's add one last enhancement to the home page. The current display doesn't indicate whether a pie is ready to eat or if it's just in the oven. It would be *devastating* to arrive at the pie shop and have to wait 20 minutes more for a pecan pie. I also might like to know if it's hot out of the oven so I can leave the office immediately.

Go to `index.jade`. Previously we wrote a helper that returned an array of CSS classes. We can do the same thing in the template, but also use another helper to calculate a class that matches the age and state of each pie.

Add square brackets around the existing `pie type`. Call a helper function: `cssClassForPieAge()`. Its only argument is a `pie`.

Current drafts of this application store all helpers in a single file. But on reflection, there are *other* ways they could be arranged. Should they be part of the mini-app? If so, the application's router could activate them with its access to the `app` object. I'll leave that as an exercise for you to try.

Right now, go to `helpers.coffee` and implement `cssClassForPieAge()`. Its only parameter is a `pie`. First, calculate the pie's age using the pie's `stateUpdatedAt`. Now we want to calculate a CSS class based first on the state of the pie, then on its age. It goes in two directions. For pies that are `making`,

we're interested if `pieAge` is larger, since that means that they are almost ready to come out of the oven. It's the opposite for `ready`. Younger pies are hot, older are warm.

For readability, it's confusing to compare against huge numbers of milliseconds or a string of multiplied numbers. It's clearer to talk about `1 minute`. In JavaScript (and CoffeeScript), we can write functions inside of other functions. So the `minute` function can live right in this function. A minute is 60,000 milliseconds. Multiply that by the number of minutes we're interested in.

From here, use these predefined CSS classes that we've styled for you: `almost-ready` and `not-ready`. The counterpart is the `ready` state. If the pie is younger than 1 minute, use `pipin-hot`. If it's less than 2 minutes old, `hot`. Otherwise, `warm`.

For the purposes of this demo, I've used increments of one minute. It's easier to see the results right away. But 15 or 30 minutes might be more accurate for the real world.

Try it! Refresh the page and you might see several colors right away. Or go to the dashboard and change the state of a pie. The color on the home page should reflect it.

CHAPTER SEVENTEEN: SOCKET.IO

In this chapter we'll finish the application. We'll update the home page in real time. We'll use socket.io to communicate between server and client. We'll emit events. We'll update the client UI when data changes on the

server.

Socket.io was an early library for real-time communication between server and client. Node.js creator Ryan Dahl even mentioned it as his favorite piece of Node.js software. When most people think of real time on Node.js, they think of Socket.io.

It's not the only Node library to tackle this problem and may not even be the best. I like `faye.js` which originally had some unique features that socket.io lacked. But many of those features have been implemented in recent versions of socket.io too. So we'll use socket.io here.

Install it with `npm`. It's now listed in `package.json`.

It's easy to end up with too much configuration in `server.js`, so make a file in `apps` for our own `socket.io` configuration.

Use the standard boilerplate; we'll need the `app` object to configure socket.io. Require `socket.io` and tell it to serve its content along with the `app` server. For debugging help, let's print a message to the console when a client connects. The syntax is `socketIO.sockets.on`. We're looking for the `connection` event. Log to the console to show that a client has connected.

Another reason to use the app object is as a storage spot for application-wide variables. I like to use `app` rather than `global`, which some developers use. Just as with development or production configuration, use app.set and store a reference to the `socketIO` server.

Back in `server.js`, require our custom socket.io configuration. Pass the `app` object as the only argument.

Now it's time to send an event to the client. Where should that occur?

A change in the data model is what we're interested in, but it feels inappropriate to communicate directly to the client from a data model. Initial drafts of this application had the Pie model inheriting from EventEmitter, which would send out events on data changes. But the simplest and most effective solution I arrived at was to send socket.io events from the controller.

The controller we're interested in is the one where the Pie status is updated. That's `admin/routes.coffee`, and it's the `put` method on the Pie `id`. In the callback to the `pie`, get a reference to `socketIO`. Sending messages to the client happen with the `emit` method. Let's emit a custom event. I'll follow the naming convention used by Backbone.js: `pie:changed`. This lets the client sort this event out from other events we might send, that might change the UI in different ways. When emitting an event, we can also send extra data. Socket.io will automatically serialize it here and deserialize it on the client. Let's send the pie object that changed.

Let's look at the client code and then tie it all together. In `assets/js/application.coffee`, I connect to socket.io. Then, using similar syntax as we used on the server, listen for the `pie:changed` event. Because we sent the pie object from the server, we get it as a parameter to the callback here.

For simplicity, we'll bluntly refresh the entire page when any pie changes. This could be a problem if thousands of pastry chefs were constantly updating thousands of pies every second, but this is a small shop and it shouldn't be triggered more than once every few minutes.

The last thing to do is deliver the socket.io library to the client. Socket.io serves it as long as we ask for it.

Go to `layout.jade`. Use a plain `script` tag to ask for `/socket.io/socket.io.js`.

We're done! Restart the server. Refresh the browser. To really see it in action, open a separate browser window to the admin dashboard. Change the status of a pie and the home page will update immediately.

Am I the only one who's hungry for some pie right now?

CHAPTER EIGHTEEN: CONCLUSION

In just a few hours, you've built a full stack Node.js application with the Express web framework. We built a simple authentication system. We organized the application into mini-apps. We rendered templates and used view helper functions. We built a user interface. We sent events to the client in real-time.

The real lesson here is that now you have mastered all the basic parts of an Express.js web application: controllers, models, views, view helpers, unit and integration tests, client assets; general application design and organization. Other features could be added for security or added functionality, but now you're ready to write and test web applications with Node.js!