Extracted from:

Deliver Audacious Web Apps with Ember 2

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Deliver Audacious Web Apps with Ember 2

Matthew White

edited by Katharine Dvorak

Deliver Audacious Web Apps with Ember 2

Matthew White

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Katharine Dvorak (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-078-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2015

CHAPTER 2

Directing Traffic with Routes

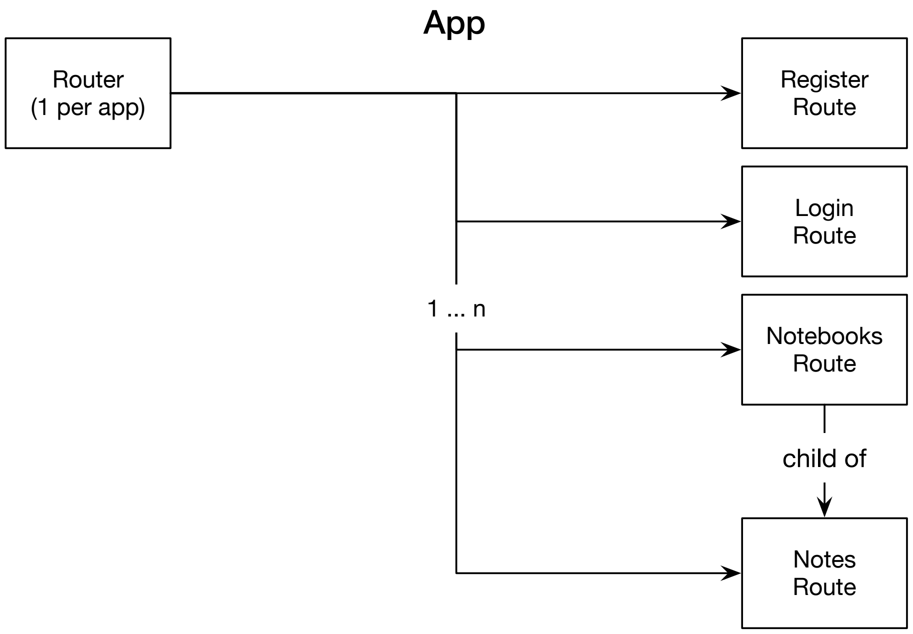
Now that we've installed Ember and Ember CLI and used them to create an app that lets users register, we're ready to dig into the major features of Ember. The first feature we'll look at is Ember routes. As I mentioned in Chapter 1, the idea of the route is one of the core principles behind Ember. A *route* is the starting point for a single unit of your app's functionality. The hierarchy of routes makes up the organizational structure of your app.

Routes are really, really important in Ember. Fortunately, they are also a really good place to start. In this chapter we'll look at how to use Ember CLI to structure your application using routes. You'll learn how to use the `Router` class to set up your route hierarchy, and how to use the `Route` class to define a route. We'll also look at how to move between routes using Ember's navigation helpers. And you'll learn how Ember uses your app URL to indicate not just your location in the app, but the records loaded into the UI. To help you keep `Router` and `Route` straight, remember that each app has only one `Router`, and within that `Router` you define the structure of your app by defining its routes. A `Route` class implementation is how you define the behavior of a single route.

To learn more about routes, we'll begin to piece together more of EmberNote's feature set by building the app's navigational structure. By the end of this chapter you'll have completed the majority of the route structure of EmberNote, and you'll know how to write the code to navigate between routes.

Use the Router Class to Organize Your App

In the previous chapter we created the `register` route for EmberNote. As you can imagine, Ember applications are made up of many routes, each one unique within the application. Taken together, the routes form a hierarchy of distinct locations in the application, exposing each of the features unique to that place in the hierarchy. Ember keeps a record of this hierarchy in a single class, the Router class, as shown in the following figure.



Because we're using Ember CLI to manage our app, we won't directly edit EmberNote's Router all that often. Most of the time we'll allow Ember CLI to do the editing for us.

Let's look at one such example. When we last worked on EmberNote, this is what our Router class looked like:

ch2/ember-note/app/router.js
```
import Ember from 'ember';
import config from './config/environment';

var Router = Ember.Router.extend({
  location: config.locationType
});

Router.map(function() {
  this.route('register');
});

export default Router;
```

Let's add a couple of routes to our app to see what happens to the Router. When users open the EmberNote app, they can register, or if they are already registered, they can log in. Once they log in, they'll see a list of their notebooks.

Let’s add two routes now, to let users log in and to let them view their notebooks. From within your ember-note directory, run the following commands:

```
$ ember generate route login --pod
$ ember generate route notebooks --pod
```

Having run these commands, you should see two new lines in your router:

```
this.route('login');
this.route('notebooks');
```

The great thing is, Ember CLI will add in new routes as you create them, without wiping out existing changes to your router.

The Router now contains the two new routes we generated via Ember CLI. We’ll now be able to define both the login and notebooks routes by building their classes.

We’ll create more routes throughout the book, and we’ll often check back in on the Router class. It’s a good class to come back to periodically, to help us stay anchored and keep the overall app structure in mind. Right now, let’s take a look at how to define a route by working through the login route.

Define Your Routes

As noted previously, each route corresponds to a given URL in your app. We defined four routes in our application with the following URLs:

- http://localhost:4200
- http://localhost;4200/register
- http://localhost:4200/login
- http://localhost:4200/notebooks

We defined register in the first chapter, and we generated the classes for login and notebook just now. The names and paths of these routes correspond to the definitions in router.js.

You may be wondering when we defined the root URL. The answer is that we defined it when we created the application. This route, known as the application route, uses Ember’s default implementation of the Route class. We modified a file called application.hbs in the first chapter. This file contains the template for the application route. By simply creating our project with Ember CLI and through Ember’s reliance on default functionality, we defined the application route.

This also raises one point about the default Ember structure. Each individual route corresponds to exactly one template. For the application route, this is the application.hbs template. For each of our generated routes, it's the template.hbs template as defined in the route's folder, assuming you've used the --pod flag; otherwise the template is named for the route, and lives in the templates folder.

Let's create the login route now. This route needs to let the user sign into the EmberNote application. It should be available from a link on the main page, just like register was, and let previously registered users sign in with their usernames. Once they've signed in, they should be taken to their list of notebooks (the other route we've generated so far). Further, once they've signed in, the register and login links should no longer appear.

To do this, we're going to make changes to the following files:

- app/template/application.hbs: To add navigation to let the user load the login route, and to hide navigation to register and login once the user has logged in
- app/login/route.js: To define the login action
- app/login/template.hbs: To define the user interface for the login route

To reach the login route, we need to first make it available via the application.hbs file. Edit this file so it looks like the following:

ch2/ember-note/app/templates/application.hbs

```
<div class="container">
  <div class="row">
    <div class="col-md-2">
      Welcome to EmberNote
    </div>
    <div class="col-md-10">
      {{#if user}}
        Hello, {{user.name}}
      {{/if}}
      {{#unless user}}
        {{#link-to 'register'}}register{{/link-to}}&nbsp;
        {{#link-to 'login'}}login{{/link-to}}
      {{/unless}}
    </div>
  </div>
  <div class="row">
    <div>
      {{outlet}}
    </div>
  </div>
</div>
```

The first addition you'll make to the file is an {{#if}} expression. The text contained within this expression is only displayed if the expression evaluates to true. In this case, if the application controller has a value in the user attribute, the user's name is displayed. We'll see how this user attribute is set when we look at the login route next.

The next section, inside the {{#unless}} expression, is displayed if the user attribute is missing. It contains the {{#link-to}} navigation item for the register route, and our new login route as well.

When users click the link to the login route, they're taken to localhost:4200/login. The login template is rendered into the {{outlet}}, meaning that the content of that template is displayed in the placeholder defined by the {{outlet}} expression. The login template, defined in app/login/template.hbs, looks like this:

ch2/ember-note/app/login/template.hbs

```
<div class="col-md-12">
  Login: {{input value=name}} <button {{action 'login'}}>Login</button>
</div>
```

This template, which is the single template associated to the login route, has a mere two elements. The {{input}} expression results in an HTML text input field being displayed and takes its value from the name attribute. The {{action}} expression results in the login action being executed in the Route, which we define next.

All of this ties together in the login route class, like this:

ch2/ember-note/app/login/route.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  actions: {
    login: function() {
      this.store.query('user', {
        name: this.controller.get('name')
      }).then((users) => {
        if(users.get('length') === 1) {
          var user = users.objectAt(0);
          this.controllerFor('application').set('user',user);
          this.transitionTo('notebooks');
        }
        else {
           console.log('unexpected query result');
        }
      });
    }
  }
});
```

As we saw in Chapter 1, this Route class uses the ES6 module syntax. The first line of the class imports the Ember framework, which is then used when we extend the base Route class in our class declaration. We're using an anonymous declaration and export of the Route. Ember CLI takes the path to the file and uses it to conclude that this class, though declared anonymously, represents the login route.

The next block of code defines the actions hash, which is a name-value pair listing of action names and implementations. Our login route requires only one action, the aptly named login action.

As described earlier in this chapter, the login action verifies that the requested user has properly registered, and then displays the list of the user's notebooks. The next block,

```
this.store.query('user', {
  name : this.controller.get('name')
})
```

executes a query against our server API using the value of the name field from template.hbs as a query parameter. We'll update the API next.

For your login route to work, you need to make a small change to the mock server we created. Open your server/mocks/users.js file and update the get endpoint as follows:

ch2/ember-note/server/mocks/users.js

```
usersRouter.get('/', function(req, res) {
  userDB.find(req.query).exec(function(error,users) {
    res.send({
      'users': users
    });
  });
});
```

Ember CLI should have deployed your changes as you made them, but if for some reason it hasn't, you may need to run the ember serve command from your ember-note directory to catch up. Once you have, visit the login route and try logging into EmberNote. You should be redirected to the notebooks route.

Because the store.find call to the API is asynchronous, it returns a Promise object to the login route. A *Promise* can be described as an object that will eventually resolve to a value. Depending on whether the function is successful or fails, different logic can be executed. Once this Promise has resolved successfully, the function in the then block is executed. First, we check that we received only one row back from the API. Assuming we have, we load the first and only row into the user object. Then, in order to make the {{#if}} and {{#unless}} blocks work properly, we set the user attribute in the application controller. Our last step is to run one of Ember's navigation functions, transitionTo. This function redirects the user to the notebooks route.

RSVP: Ember's Promise Library

Promises are a widely used idea, without a canonical implementation. ECMAScript 2015 is expected to include a Promise implementation, but in the meantime, Ember relies on RSVP. RSVP is compliant with the in-process specification of ES 2015, but it's available now. It's maintained by members of the core Ember team and available on GitHub at github.com/tildeio/rsvp.js.

We've covered more than just routes in this section to let us build a good example. The key elements for you to focus on now are the {{#link-to}} expression, the transitionTo function, and the actions hash. As used in the application.hbs template, the {{#link-to}} expression allows the user to load the login route's template.hbs into the {{outlet}} in application.hbs, when the resulting link is clicked. For this to work, we need the login route defined in router.js. The {{link-to}} expression is one way of causing the parent route to load a child route into its outlet.

Another way to implement navigation is by using functions like transitionTo to move between routes. As used in the login route, it takes the currently loaded route (as determined by the instance calling the function), and replaces it with the route named in the function call. As noted in our example, we can use this function to indicate not only which route to load, but also which record it should load. We'll see more on how the destination route knows which record to load when we look at setting a route's model shortly.

Last, the actions hash is used to define name-value pairs of actions and their related functions. This concept isn't route-specific, and we'll see more examples of this later, but it's worth noting that one option for action handling is to add your actions to a route in this way.

Now we understand more of what a route is and how it's used, let's take a look at how to use a route to load and display data via the route's model hook.