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

Start Your App

Ember CLI takes a very app-centric approach. The first thing you do is to create the app, and every module you create is part of that app. As a result, we're going to follow suit, and explore all of Ember's key concepts in the context of one app.

We'll be writing an app called EmberNote, a slimmed-down clone of Evernote,[4] the web-based note-taking app. Our first step is to create the app's repository (so called because Ember CLI assumes you'll be using Git as a version control system).

Creating the App Repository

Open a command prompt, navigate to a location where you keep your code, and run the following command:

```
$ ember new ember-note
```

This will create a new directory for your app. Take a look at the tip that follows if any issues arise.

New Projects with Ember CLI

As of this writing, there's a probable race condition that can occur with Ember CLI when creating a new repository. If you see an error that looks something like this: `ENOENT, open 'ember/package.json'`, you may have run into this error. Usually, the `ember new` command succeeds on the second try.

On certain non-Windows systems, you may also see a console warning about not having `watchman` installed. Ember CLI uses `watchman` to monitor your filesystem for changes and live deploys them to your local server as you're working on your Ember app. It's good to have `watchman`, if for no other reason than to avoid seeing this error, but it's not necessary. Without `watchman`, Ember CLI uses a tool called `NodeWatcher` for the same purpose. If you want to install `watchman`, visit facebook.github.io/watchman/docs/install.html for platform-specific installation instructions.

Before we write our UI code, let's also install a Bootstrap stylesheet so we can enjoy a UI that doesn't look like it's from the early 1990s. From a command prompt in the newly created `ember-note` directory, run the following commands:

4. evernote.com

```
$ cd ember-note
$ npm install
$ bower install
$ bower install bootstrap --save
```

Then add these lines to ember-note/ember-cli-build.js, right before the last line of code:

```
app.import('bower_components/bootstrap/dist/css/bootstrap.css');
app.import('bower_components/bootstrap/dist/css/bootstrap.css.map',{
  destDir: 'assets'
});
```

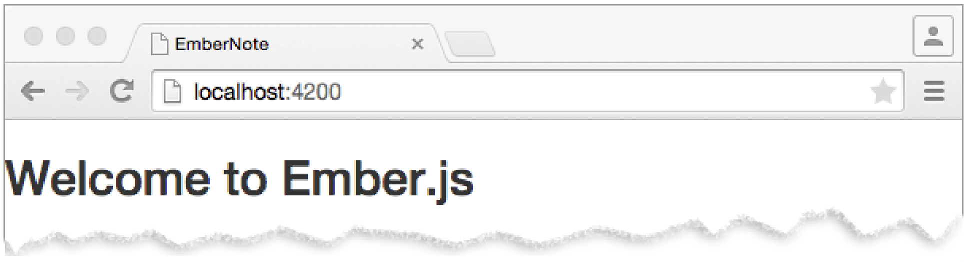
When you run these commands, Ember CLI creates your app's folder structure for you. Our task throughout the rest of this book is to take that structure and fill in all of the details to make an app out of it. Make no mistake, Ember and Ember CLI are opinionated tools. They have built-in expectations of how to develop web apps, known as *convention-over-configuration*. The Ember team hopes to take a set of best practices, codify them into the framework, and allow the developer to focus on more important decisions. To get good use from these tools, you'll want to follow their lead.

Running Your App

Let's run the app we made and see what that looks like. From a command prompt in your new ember-note directory, run the following command:

```
$ ember serve
```

You should see some command output that tells you that your build was successful and that your app is running on port 4200 of your localhost. Let's open the browser and check. You should see something similar to the output shown in the following figure if you point your browser to localhost:4200.



Welcome to Ember.js

The ember serve command did a few things for you. First, it built a deployable version of your app and placed it in the ember-note/dist directory. Then, it started a node instance at port 4200 and deployed your app to that instance for testing. Once you've run this command, it will pick up on changes you make to the source code.

Let's try one now. Open the ember-note/app/templates/application.hbs file. Its contents should look like this:

```
<h2 id="title">Welcome to Ember</h2>

{{outlet}}
```

Make a change to the file so the text says "Welcome to EmberNote." The file should then look like this:

```
<h2 id="title">Welcome to EmberNote</h2>

{{outlet}}
```

As soon as you save the change, look at your browser. Ember CLI detected the change, rebuilt your app, and deployed it when you saved. This is a big time-saver, because you don't have to manually take steps to redeploy the app yourself. It all happens out of the box with Ember CLI!

You can see that Ember CLI is a handy tool to have around. Now it's time to get into some code. If we're going to build an app, we'll need to be able to let users sign up, right? Let's start there.

Registering Users

Let's look at a straightforward example that will demonstrate many of the core features of an Ember app: routes, templates, and models. We're going to build a simplified registration page that will allow users to sign up for the EmberNote service. This page has a single text field where the user can enter his or her username, a button to trigger the add action, and some text that will be displayed when a new user is added.

In an Ember app, all of your functionality is organized under routes. A *route* is in essence a URL segment, as described in *Nest Routes*, on page ?. So, our next step is to use Ember CLI to create a route. Open a new command shell, and from the root of your ember-note project, run the following command:

```
$ ember generate route register --pod
```

The --pod flag, when added to the generate command, creates a directory called register, which includes the files that are needed to add your route: route.js and template.hbs. They are empty at the moment, and you'll fill them out as needed to create the route's features. Generally speaking, you use a pod when you want to organize your project by feature (that is, create a register directory with a route.js and a template.hbs file), rather than by object type.

Earlier, when you created the app with the ember new command, the following four files were created: index.html, app.js, router.js, and application.hbs. The ember generate command then edits the router.js file under your app directory. We'll dig into this more in *Use the Router Class to Organize Your App*, on page ?, but the router.js class is used to implement the hierarchy of routes within your application.

If you run the ember generate command but leave off the --pod flag, Ember CLI still creates the same objects, but instead of creating a file called route.js in the register directory, it creates a file called register.js in the route directory. For our purposes, it's preferable to see each of the files we need for a route in one location, so we'll use the --pod flag throughout the book.

Let's take a look at each of the files we've created so far.

index.html

This file is the starting point for your application. It was created when you ran the ember new ember-note command. Let's have a look.

ch1/ember-note/app/index.html

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>EmberNote</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    {{content-for 'head'}}

    <link rel="stylesheet" href="assets/vendor.css">
    <link rel="stylesheet" href="assets/ember-note.css">

    {{content-for 'head-footer'}}
  </head>
  <body>
    {{content-for 'body'}}

    <script src="assets/vendor.js"></script>
    <script src="assets/ember-note.js"></script>

    {{content-for 'body-footer'}}
  </body>
</html>
```

The majority of index.html is boilerplate HTML to create a starting page. Ember CLI adds in two stylesheets named vendor.css and ember-note.css to style the application. We won't be doing much with stylesheets in this book, but you should know that vendor.css will capture styles used in any third-party code you bring into the app, and ember-note.css is for styles that you define.

The same thing applies to vendor.js and ember-note.js. The Ember CLI build process creates one file for all of the third-party code you use in your app, called vendor.js, and one for your application code, called ember-note.js. These are included in index.html.

You may also notice the {{content-for}} expressions. These allow you to include additional content in your page using Ember CLI addons.

app.js

The app.js file is used to create the Ember environment. Take a look at the following code:

ch1/ember-note/app/app.js
```
import Ember from 'ember';
import Resolver from 'ember/resolver';
import loadInitializers from 'ember/load-initializers';
import config from './config/environment';

var App;

Ember.MODEL_FACTORY_INJECTIONS = true;

App = Ember.Application.extend({
  modulePrefix: config.modulePrefix,
  podModulePrefix: config.podModulePrefix,
  Resolver: Resolver
});

loadInitializers(App, config.modulePrefix);

export default App;
```

If you're a veteran JavaScript developer who hasn't looked at ECMAScript 2015 yet, the first four lines may have your eyes popping out of your head. ECMAScript is the formal name of the JavaScript language as specified by ECMA International, and ECMAScript 2015 (also known as ECMAScript 6 or ES6) is the latest release of the standard. Yes, those four lines are importing dependencies from other JavaScript files to use their modules within the current module. This is a hugely important aspect of using Ember CLI. When you start writing your code in a modular fashion, your dependencies become easier to reason about, and you eliminate the need to stuff objects into the global scope. It also helps you to structure your code in a way that makes sense. Don't worry whether the browser supports ECMAScript 2015, though. Ember CLI transpiles your code into a format that the browser can execute.

The purpose of the code in app.js is to set up app-wide constructs. One of these, the Ember Resolver, is used to look up classes throughout the app. The modulePrefix variable is set to the value ember-note, as defined when you set up your app. The podModulePrefix isn't set, as it wasn't defined yet for our project configuration, which was read from config/environment.js.

The last line of the file makes the App class available as a module. Don't worry if this doesn't quite make sense yet—you'll see many more examples of how Ember modules work as we go along. For now, let's look at another file that was created when the app was set up, the router.js file.

router.js

The router.js file is where you can really start to see how Ember works. One of the most fundamental features of Ember is the notion of a route. Simply put, a route is a location within your app. If you think of your app as a collection of pages or page sections, each nested within one another, then each of those pages corresponds to a route. And the Router class, as defined in router.js, is where you define the structure of your app. Let's look at the code:

ch1/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;
```

At the top of the router.js file you see the necessary imports. In the class definition, which begins with var Router = Ember.Router.extend, we're bringing in some configuration variables from the global configuration as defined when we set up the app.

But Router.map is where the real action happens. As you can see, we've already created a route called register. This route was added to the Router when we ran the ember generate route register --pod command. A number of other files were also generated, and we'll edit those files to create our registration feature in a moment. The register route appears to be a top-level route, but is actually at the first level of child routes beneath a default route for the application. Let's look at that default route by way of its associated template.

application.hbs

Each application has a root route, the Application route. Further, each route corresponds to exactly one template. A template is a chunk of your user interface. In the case of the application route, this template is defined in application.hbs. Let's modify this template now to look like the following:

ch1/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">
      {{#link-to 'register'}}register{{/link-to}}
    </div>
  </div>
  <div class="row">
    <div>
      {{outlet}}
    </div>
  </div>
</div>
```

Each Ember template is a mixture of HTML and the Handlebars expression language, which is the portions of code set off with "{{ }}" characters. Handlebars is an expression language used by Ember to describe those segments of Ember templates that are in some way related to the Ember application code. You'll see many examples of Ember using Handlebars expressions throughout the book, always using the "{{ }}" characters. Ember ships with a compiler that turns Handlebars template expressions into DOM objects, as we'll see in *Compile Templates*, on page ?.

As we saw earlier, each app consists of a hierarchy of routes. The {{#link-to}} expression in the application.hbs file is used to generate a hyperlink that when clicked will load the register route into the {{outlet}}.

You may be wondering where the Application route is defined. In this particular case, because we don't have any specific action handling or data loading to do, we don't need to define the route class; we can simply rely on Ember to create a route implementation on the fly using default route behavior.

All of the code we've seen thus far has set up the bare minimum of application. Now, let's see how to add a feature to our app by building our register route.

Building the Register Route

As discussed earlier, the register route will allow a user to register for the EmberNote app by choosing a username. When we created this route using the CLI, the following file was created (note, this is route.js, which is different from router.js):

ch1/ember-note/app/register/route.js

```
import Ember from 'ember';

export default Ember.Route.extend({
});
```

This is how you define an empty route class. A route has two primary reasons to exist: loading data to be displayed by the route and responding to actions. In our case, we don't have any data to display, and we need to define an addNew action to add new users to the app. We'll come back to that action shortly. First, let's see what the user interface will look like. Here's our route's template file, with the necessary UI code added in:

ch1/ember-note/app/register/template.hbs

```
<div class="col-md-12">
  Register new user: {{input value=name}}
  <button {{action 'addNew'}}>Add</button>
</div>
<div class="col-md-12">
  {{message}}
</div>
```

The {{input}} field allows us to key in a name, which will be added to the name attribute of the controller (which we'll create in a moment). The {{action}} expression, which we included in the <button>, binds the button's click action to the addNew action in our route. And the {{message}} expression simply displays the contents of the message attribute in the controller. We don't yet have a child route for register, but if we did, it would be rendered into {{outlet}}, similar to how the register route is rendered into the {{outlet}} in application.hbs.

We'll need some way to capture the UI state contained in the name and message fields. To do this, we'll rely on the default controller. In Ember, each route maps to exactly one controller, and that controller acts as a proxy for the underlying model, including any UI state. Because we don't have any data to load into our model yet, all we really use the controller for is to provide a place to put the state of the name and message fields from our template. In cases when you need only very simple functionality, you can skip creating a unique controller implementation for your route and just rely on the default implementation that Ember assigns to your route. Now, let's create the addNew action to go in our route.

Avoiding Controllers

There's another reason for not choosing to rely heavily on controllers. The Ember team plans to deprecate controllers early in the 2.x release cycle, in favor of performing similar work in the route or in components. Future versions of this book will follow suit, and for now, we're avoiding using controllers as much as possible.

Edit the register route so it looks like this:

ch1/ember-note/app/register/route.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  actions: {
    addNew: function() {
      var user = this.store.createRecord('user', {
        name : this.controller.get('name')
      });
      user.save().then(() => {
        console.log('save successful');
        this.controller.set('message','A new user with the name "'
          + this.controller.get('name') + '" was added!');
        this.controller.set('name',null);
      }, function() {
        console.log('save failed');
      });
    }
  }
});
```

The first thing we do here is to add an actions hash to the route. A *hash* is a list of name-value pairs. In this case, the names are names of functions, and the values are the function definitions. We have one function, addNew. This function will be called when we click the button in our user interface, because that's how it was defined in the template.

We're also relying on a bit of new ES 2015 syntax here. The code () =>, known as a *fat arrow function*, will let us continue to use this within the body of the then function, and this will continue to refer to the route object. The call to createRecord uses the Ember Data framework to create a new instance of a model object, and the subsequent call to save saves that record to persistent storage. Ember Data provides access to data over RESTful services. Before we test all of this, we'll need to set up a mock service for Ember Data to call, and a model object to use that data. Let's do that now.

Adding a Data Service

Ember and Ember CLI offer two ways to provide data to your app during development: fixtures and mock services. While fixtures are easier to set up, mock services offer you the ability to more fully test your code by requiring you to implement everything required to access a RESTful service. So we use mock services in this book. If you're more concerned with testing your core Ember classes, and you're not worried about data access layer testing, you might consider fixtures, which are essentially classes with hardcoded data and simple access methods.

In fact, we go an extra step beyond mock services: our services will persist data across a restart so that as we build the app we'll also see data accumulate. The default mock services created by Ember CLI don't do this, and although this behavior might be preferable for developing an app under normal circumstances, I'd prefer that our data survive a service restart so that our test data is maintained as we work through the examples. So, let's see how a mock service works.

We start by creating a model class for our user object. At the moment, our user consists of one data attribute: the name. Let's use Ember CLI to generate a stub class.

```
$ ember generate model user
```

Here's the class, with implementation details added in:

ch1/ember-note/app/models/user.js

```
import DS from 'ember-data';

export default DS.Model.extend({
  name: DS.attr('string')
});
```

This class relies on the Ember Data libraries to create an instance of a `Model`, with a single attribute called `name`. This attribute has a type of string, meaning it will hold text. That's it. This is all we need to model our user object. Now, to interact with the service we're planning to create, we need a `RESTAdapter` and a `RESTSerializer`.

To create our `RESTAdapter`, run the following command:

```
$ ember generate adapter application
```

This creates a RESTAdapter that will be used by the entire application, which is why we named it application. Now, edit the resulting file so it looks like this:

ch1/ember-note/app/adapters/application.js
```
import DS from 'ember-data';

export default DS.RESTAdapter.extend({
  namespace: 'api'
});
```

Similarly, to create the RESTSerializer, run this command:

```
$ ember generate serializer application
```

And the resulting class will look like this:

ch1/ember-note/app/serializers/application.js
```
import DS from 'ember-data';

export default DS.RESTSerializer.extend({});
```

The RESTAdapter has just one purpose: to define the path on the server where we can find our RESTful service. The RESTSerializer is an empty implementation, as we're using default functionality. Because we're going to create a service that exposes the expected API for each model object, this is all we need to do. We'll see a lot more on these in *Adapt to a Nonconventional API*, on page ? and *Use Serializers to Access Legacy APIs*, on page ?.

Generally, this is where your Ember development would stop. Creating a service to provide data is generally out of the scope of what you'd be doing with Ember, which is a front-end framework, and not for developing REST services. As I said earlier, we're creating a mock service that our app can use for testing and then taking it one step further by making the data persistent.