Rock and Roll with Ember.js

Balint Erdi

Set List

CHAPTER 3

Templates and data bindings

Tuning

We have an app skeleton that Ember CLI has created for us, so we can start adding functionality.

We can start the development server (written in express.js) by typing `ember serve` at the command line:

```
$ cd rarwe
$ ember serve
```

This command first builds the whole client-side application, then launches the development server on port 4200. Thanks to the `ember-cli-inject-live-reload` package it will also pick up any changes (file updates, additions, or deletions) in the project files, run a syntax check (eslint) on them, and rebuild the app.

When we navigate to `http://localhost:4200` we are presented with an empty page with a "Welcome to Ember" header.

New styles

To give our application a decent theme, let's open up the `app/index.html` which is the skeleton of our Ember app, and add a few classes to the body element:

```
// app/index.html
<!DOCTYPE html>
<html>
  (...)
- <body>
+ <body class="avenir near-white bg-mid-gray">
    (...)
  </body>
</html>
```

HTML

(The CSS library that we'll includes the necessary rules for styling.)

Let's do the exact same thing for `tests/index.html` which is used when Ember is running in testing mode.

On top of that, in order for our first template to look a bit better, write the following CSS rules into `app/styles/app.css`:

```
// app/styles/app.css
a {
  color: inherit;
  text-decoration: none;
}

.page-header {
  padding-bottom: 9px;
  margin: 40px 0 20px;
  border-bottom: 1px solid;
}

.page-header > .app-title {
  margin-top: 0;
  margin-bottom: 0;
}

.app-title > .subtitle {
  font-weight: normal;
}

.list > li {
  margin-bottom: 10px;
}
```

With these in place, we can start getting acquainted with templates.

The first template

To start off, let's remove the header and paste in the following content instead:

```
1  <!-- app/templates/application.hbs -->
2  <h2 id="title">Welcome to Ember</h2>
3
4  {{outlet}}
5  <ul>
6    {{#each model as |song|}}
7      <li>{{song.title}} by {{song.band}} ({{song.rating}})</li>
8    {{/each}}
9  </ul>
```

(If you don't want to type in or copy-paste the following code snippets, you don't have to, but I'd recommend you do it. Not only does it build character, it also accelerates the learning process.)

Templates are chunks of HTML with dynamic elements that get compiled by the application and inserted into the DOM. They are what build up the page and give it its structure.

Ember's templating engine is built on top of Handlebars. Anything enclosed in mustaches (`{{ ... }}`) is a dynamic expression. The first such expression is a helper, `{{#each}}`. It loops over the items after the `each` keyword making it accessible inside the block with the name given after the `as` keyword.

In the above case, for each song in `model`, the block that the helper wraps is going to be rendered with `song` referring to the current iteration item.

Okay, but where is the `model` that contains the list of songs specified? Good question! It is set by the model hook of the corresponding route (we'll see how in more detail in the Routing chapter), so let's first generate the application route by running the following command (let's answer by "no" when asked if the template should be overwritten):

```
$ ember generate route application
```

and then paste the following code into it, overwriting its generated content:

```js
// app/routes/application.js
import Route from '@ember/routing/route';
import EmberObject from '@ember/object';

export default Route.extend({
  model() {
    let blackDog = EmberObject.create({
      title: 'Black Dog',
      band: 'Led Zeppelin',
      rating: 3
    });

    let yellowLedbetter = EmberObject.create({
      title: 'Yellow Ledbetter',
      band: 'Pearl Jam',
      rating: 4
    });

    let pretender = EmberObject.create({
      title: 'The Pretender',
      band: 'Foo Fighters',
      rating: 2
    });

    return [blackDog, yellowLedbetter, pretender];
  }
});
```

JS

A WORD ABOUT GENERATED TEST FILES

BACKSTAGE

Generating routes (and, in later chapters, other components) also creates the test file stubs needed to test the generated entity. Please delete them as we'll have a separate chapter on Testing where acceptance, integration and unit tests are all covered and we'll create the test files we need.

So far so good, we can see the list of songs:

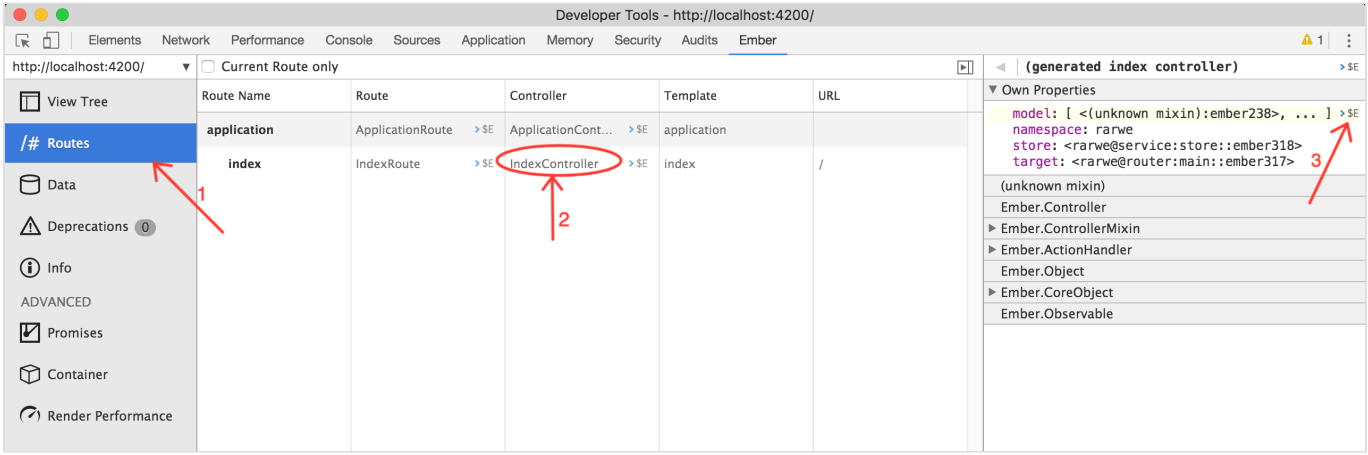
- Black Dog by Led Zeppelin (3)
- Yellow Ledbetter by Pearl Jam (4)
- The Pretender by Foo Fighters (2)

Let's now try to change the title of one of the songs and see what happens.

In order to do that, let's install the Ember Inspector extension that lets us inspect (and poke) the internals of our Ember application. It is a fantastic tool that is indispensable for developing Ember apps.

You can download it either for Chrome, or for Firefox.

Once you have installed the extension, open the Developer Tools and you'll have an Ember tab right beside the default tabs. Click it and select Routes from the left panel. Next, click `index` in the Controller column and then the $E next to the `model` property in the right sidebar (you will see the `$E` when you hover over the model property)



We have now stored this value as $E and can play around with it in the console.

Let's type in the following:

```js
blackDog = $E.get('firstObject');
blackDog.title = 'White Cat';
```

When we do that, we get the following error:

```
"Assertion Failed: You must use Ember.set() to set the `title` property (of
<Ember.Object:ember233>) to `White Cat`.
```

`Ember.set` takes three arguments: the object to modify, the property to set, and the value to set it to. We can now follow Ember's guidance and set the song title:

```js
Ember.set(blackDog, 'title', 'White Cat');
// => "White Cat"
```

When we send this command, the list updates automatically with the new title:

- White Cat by Led Zeppelin (3)
- Yellow Ledbetter by Pearl Jam (4)
- The Pretender by Foo Fighters (2)

(If you want to learn more about why the properties of Ember objects need to be accessed via `set` and `get`, as opposed to just using the JavaScript property accessors, read the `EmberObject accessors` section in the Encore.

Ok, so this works! Let's now define a new song in the Console:

```
let brother = {
  title: 'Brother',
  band: 'Alice in Chains',
  rating: 4
};
```
JS

... and add it to the existing ones:

```
$E.push(brother);
// => 4
```
JS

It seems to have succeeded (`push` returned the number of songs) but nothing happens, the list stayed the same.

Ember.js takes Handlebars templates and sprinkles magic fairy dust on them to make them "auto-updating", as we have just seen with the song title. It does this by observing the arrays (the array that the `model` function returns) and properties (each song title, band and rating) referenced in the template and updates the appropriate template segments when they change.

However, it can only work its magic if it knows about the change, and the default JavaScript methods (setting a property via a simple assignment via = or using `Array.push`) do not ensure this.

That's why we needed to use `Ember.set` and that's why we need to use the "change-aware" counterpart of `push`, `pushObject`:

```
$E.pushObject(brother);
```
JS

We now see the list update correctly:

- White Cat by Led Zeppelin (3)
- Yellow Ledbetter by Pearl Jam (4)
- The Pretender by Foo Fighters (2)
- Brother by Alice in Chains (4)
- Brother by Alice in Chains (4)

Note that the new song appears twice which goes to show that it was indeed added successfully the first time, with `push`, but the template did not learn about the change. When `pushObject` made the template re-render, there were already 5 elements in the array, the last two of them with the new song title, Brother.

BACKSTAGE

HANDLEBARS' FAIL-SOFTNESS

An important feature of Handlebars is "fail-softness". If a property in the template is missing (`undefined` or `null`), Handlebars silently ignores it. This proves useful in many situations, especially when using 'property paths'.

Imagine we have the following expression in a template:

HBS

```
{{band.manager.address.street}}
```

A band might not have a manager. If it does, the manager might not have an address. And if he does, the street address might not be given. So imagine that Handlebars threw an error when looking up a missing property, or iterating on such a property. We would have to have several levels of conditional expressions in our template or use some other programming technique to guard against this possibility.

The flip side of fail-softness is that it can lead to bugs that are hard to find. A misspelled property name (e.g `sonsg` instead of `songs`) can induce a relatively long debugging session.

Be aware of this Handlebars feature, and if something should really work in the template, but does not, check the spelling of the properties.

Before we move on, let's take a closer look at the first hand-written JavaScript file in our application, the application route:

```
// app/routes/application.js
import Route from '@ember/routing/route';

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

JS

`import` and `export` sounds suspiciously as if there was a module system beneath, and in fact there is. You can read more about it in the Encore.

Let's now create our first model class to represent the songs:

```
1  // app/routes/application.js
2  import Route from '@ember/routing/route';
3  import EmberObject from '@ember/object';
4
5  let Song = EmberObject.extend({
6     title: '',
7     band: '',
8     rating: 0
9  });
10
11 export default Route.extend({
12   model() {
13     let blackDog = EmberObject.create({
14     let blackDog = Song.create({
15       title: 'Black Dog',
16       band: 'Led Zeppelin',
17       rating: 3
18     });
19
20     let yellowLedbetter = EmberObject.create({
21     let yellowLedbetter = Song.create({
22       title: 'Yellow Ledbetter',
23       band: 'Pearl Jam',
24       rating: 4
25     });
26
27     let pretender = EmberObject.create({
28     let pretender = Song.create({
29       title: 'The Pretender',
30       band: 'Foo Fighters',
31       rating: 2
32     });
33
34     return [blackDog, yellowLedbetter, pretender];
35   }
36 });
```

JS

The first line,

```
import Route from '@ember/routing/route';
```

is importing the Route class from the `@ember/routing/route` package.

Ember uses ES6 modules and the framework exposes its pieces via this mechanism, so almost all of our application files will begin with a series of import statements. You can see the full list of Ember modules here.

The module for the application route defines a default export which is an extension (a descendant) of the Ember `Route` class.

Let's now focus to the next line:

```
let Song = EmberObject.extend({
```

`Song` extends `EmberObject` which is the base class of most Ember classes. When we write `EmberObject.extend` we create a subclass of `EmberObject` so that our new class will inherit all the functionality `EmberObject` defines, such as getting and setting properties. The reason we extend the base class is to add properties and methods to it to make the class implement the logic we want in our app. Here, we only do it to define default values for instances of `Song`.

Instances of the class can be created through its `create` method, which is what we do inside the `model` function, creating three instances of `Song`. If you'd like a deep(er) dive on class and instance properties, hit up the relevant section in the Encore.

Let's spiff the list up now by adding some style.

Adding assets to the build

The fastest (and for most of us developers out there, perhaps the only) way to add a decent look to our application is by using an CSS library. We will use the popular Tachyons.

Ember CLI bundles all JavaScript files that the application needs (including the application's code) into one big ball in the build process. It names it after the project name, in our case, `rarwe.js`. It does likewise for CSS.

A tool called Broccoli is responsible for bundling our assets and its configuration is found in `ember-cli-build.js`. There are two ways to tell Broccoli to include the Bootstrap assets in our application.

The first way is to use an Ember addon suited for that library, the second is to add the assets manually to Broccoli in the `ember-cli-build.js` file. Leveraging an addon is a lot simpler, so that's the one we're going to go with:

```
ember install ember-cli-tachyons-shim
```

, which saves the addon as a dependency in `package.json` and runs the default blueprint of the addon.

```
  1  // package.json
  2  {
  3    "name": "rarwe",
  4    (...)
  5    "devDependencies": {
  6      (...)
+ 7      "ember-cli-tachyons-shim": "^4.9.0",
  8      (...)
+ 9      "tachyons": "4.9.0"
 10    }
 11  }
```

JS

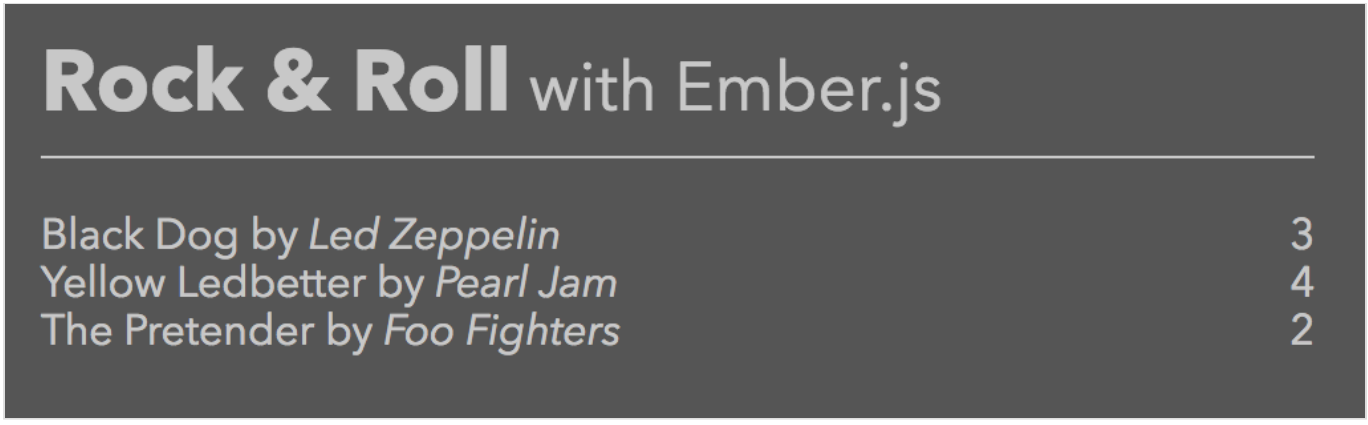
The `ember serve` process should be restarted because we have added new dependencies to the `package.json` file.

We need to extend our bare-bones markup, add a nice header and some classes to improve the general aesthetics of our app:

```hbs
<!-- app/templates/application.hbs -->
<div class="center w-50">
  <div class="page-header">
    <h1 class="app-title">Rock & Roll<small class="subtitle"> with Ember.js</small></h1>
  </div>
  <div class="main-content">
    <ul class="list pa0 ma0">
      {{#each model as |song|}}
        <li>
          {{song.title}} by <em>{{song.band}}</em>
          <span class="rating fr">{{song.rating}}</span>
        </li>
      {{/each}}
    </ul>
  </div>
</div>

```

Once the app is rebuilt, it now looks fabulous:



Next song

Routing is what we'll look at next. It is arguably the most important component in Ember applications, and is the key to understanding Ember itself, so buckle up and let's dive into it.