Rock and Roll with Ember.js

Balint Erdi

Set List

Preface

The dawn of client-side frameworks

(You are reading version 20180104 of this book.)

Classic web development is server-side development. Most of the code for web applications has historically lived on the server. However, providing a decent user experience demands that not all user interactions require a roundtrip to the server, since whole page reloads break the flow of communication between the user and the application.

Traditionally, this has been relieved by adding sprinkles of client-side code that interpret a specific user interaction, such as clicking a checkbox of a to-do list item. The client-side code transforms that intent into action (marks the item as complete) and it issues an XHR request to the server to carry out that action.

With this change, user experience has definitely improved, since the action happens right away. The server response only needs to repaint a small part of the page and the user is given immediate feedback.

This model works well as long as the application is very simple. However, as more elements are added to the page and additional information derived from the same data source is displayed in several places, it starts to break down. For example, imagine a to-do list app. It shows four incomplete tasks, and somewhere else it displays the number 4 as the corresponding number of pending to-dos. If the user marks a to-do item as completed, the client-side code needs to make sure that the counter is decremented -- in this case the counter should go to 3 when the user marks off one task as completed. This means that a change in a data source has a ripple effect. Now picture that this app has several to-do lists on the page, which are ordered by the number of incomplete tasks they have. Ticking off an item decreases the number of pending to-dos on that list, which in turn might change the ordering of the lists.

Not only is it tiresome for developers to keep track of the implications that each change in data may have, it also makes things more and more error prone to move further away from the original source. Fortunately, a library or framework eliminates such a source of bugs.

The functionality needed by client-side applications overlaps heavily. Data binding, for example, can eliminate the to-do list problem described above. This spurred developers to extract this functionality and include it in libraries. Quite recently, full-fledged frameworks that leverage some of these libraries have appeared, and serve as single-stop, "all-parts included" tools to build a complete application.

During the same period, the devices executing client-side JavaScript evolved to have much better performance. While 10 to 15 years ago both client hardware and the software running JavaScript were "slow and dumb", this is certainly no longer the case. Nowadays even the slowest smartphone wields 1 GHz CPUs and has integrated GPUs.

On the software side, the release of Google's browser Chrome, boasting the ultra-fast JavaScript engine V8, sparked a new era of browser wars. Fierce rivalry made JavaScript engines in competing browsers ever faster, to the benefit of all Internet users, and the authors of JavaScript libraries in particular.

There has never been a better time to get acquainted with front-end web development, especially wholesale client-side frameworks.

As web applications become ever more complex and powerful and the whole-page reload model of yesteryear falls by the wayside, the demand for fast desktop-like user interaction on the internet rises constantly. The time has come for client-side web application development to rise to the status of their server-side counterparts. Client-side frameworks are paving the way.

My love affair with Ember.js

My enthusiasm for Ember.js began in February 2013. At that time, Ember was going through the 0.9 => 1.0.beta.1 change and a sizable portion of the API was being rewritten. Guides to the differences didn't exist, and the documentation was lacking. Some blog posts described the 0.9 API, some the 1.0.beta. A fellow developer who I was working on a pet project with found this annoying enough to quit.

I managed to get past the frustration, supported in part by the helpfulness and care that members of the core team showed in reply to my despairing tweets, and in part because I started to see the decisions shaping the framework's architecture and they really appealed to me.

Having heard from several sources that Ember is really hard to understand, I started to feel that I could help people climb over the fence and see the true Ember, which for me felt like a wonderland of a green meadow filled with flowers and bunnies.

I held a workshop at a Ruby conference in Berlin and started a mailing list. With absolutely no prior knowledge about screencasting, I made a video series about building a simpler version of the example application we'll develop in this book. I wrote a blog post every week for about 6 months.

In various blog posts and at various conferences I claimed both that the future for Ember.js is bright, and that it has never been a better time to become a new Ember developer (or a better one). While this latter claim might sound silly, I still uphold it today, as the framework evolves and matures and gets better every day.

Building applications with Ember, whether small and large, is a fascinating process. I hope to infect you with my affection for Ember and equip you with the necessary knowledge or add to your existing skills, so you can start the journey with courage.

Acknowledgements

You wouldn't be reading this book had I not received encouraging words from a lot of people from all around the world. From almost the first moment my screencast went live and I sent it out to the few dozen email subscribers I had at the time, I started receiving kind emails which inspired me to keep going and doing more. This happened again with my blog posts, and also when I sent out sample chapters of the book. I want to thank Thomas Martineau, Curtis Wallen, Chase Pursley, Antonio Antillon, Stefan Penner, Bence Golda, Stephen Kane, Tony Brown, Evgenij Pirogov, Fayimora Femi-Balogun, Claude Precourt, Gregory Gerard and David Toth for keeping me going.

Furthermore, I would like to say thanks to the many people who generously spent some of their precious free time reading review copies of this book and sending me edits. In some cases, the edits were embarrassing - I used the wrong verb ending, or said things like "on February 2013". In other cases, the suggested corrections were more subtle but still very valuable. I learned that you really need a second pair of eyes to make a book publishable, and each additional pair makes it better.

Taras Mankowski, Matthew Beale, Cory Forsyth, Laszlo Bacsi, Igor Terzic, Tom De Smet, Sergio Arbeo, David Lormor, Ricardo Mendes, David Chen and Raul Murciano suggested a slate of technical improvements while Ugis Ozols, Gaspar Vajo, Charlie Jones, Philip Poots, Ankeet Maini, Christoffer Persson, Gustavo Guimaraes, Jaime Iniesta, Roel van der Hoorn and Bence Golda helped filter out typos and sloppy sentence construction (some of you contributed to both, but this sub-categorization could go on forever). You made the book much better and I am hugely indebted to all of you.

I would also like to thank those who contributed to the marketing of the book, whether by including my blog posts in newsletters, talking about my book at Ember meetups, or retweets. A great product without good marketing is certainly a failure, so I would like to express my gratitude to Owain Williams, Taras Mankowski, Zoltan Debre, Philip Poots, Tom Dale, Gaspar Vajo, Cory Forsyth, Luca Mearelli, Jaime Iniesta, Dan Stocker, Andras Tarsoly, Reuven Lerner, Gabor Babicz, Joachim Haagen Skeie, Barry van Someren, Attila Gyorffy, Spyros Kalatzis, Adrian Kovacs and Thomas Martineau.

When it comes to the creative endeavors of building a product and an audience, I learned everything I know from Brennan Dunn, Nathan Berry, Paul Jarvis and Sacha Greif. Sacha also shared very practical tips with respect to writing and self-publishing an ebook. Thank you to all of you.

I mustn't fail to mention the members of my weekly MasterMind group. They listened to my challenges and helped me with practical advice. These fine gentlemen are Reuven Lerner, Barry van Someren, Spyros Kalatzis and Luca Mearelli.

Meredith Weinhold did a fantastic job proofreading the book. She even spotted an error in a code snippet and pointed out that Black Dog should have a higher rating than 3/5. Should you need help with editing, proofreading, or both, you can find her on Upwork.

Almudena Garcia went far beyond just adding style to the pdf version and making a great-looking sales page. She extracted whatever I needed from a psd, coded Ruby, and prepared the images for stickers. As well, she had great ideas on how to make the book and the sales page better. If you have a project that needs something similar, you can hire her through her personal site.

I owe a debt of gratitude to my wife, Petra, who put up with my spending the first hours of the night sitting in front of the computer (especially in the last two months), and my constantly talking about this book of mine.

If I have failed to mention you here, please accept my sincere apologies and deepest gratitude. Please, write me an email so that I can include you.

Last, but not least, the Ember Core Team deserves huge kudos. Without you this book would obviously not exist. Thank you for your relentless (and sometimes what seems like superhuman) work.

CHAPTER 1

Introduction to Ember.js

Ember grew out of the SproutCore 2 project and is one of a handful of full-fledged frameworks used to build elaborate client-side applications. The first commit happened in April 2011, and it reached its first stable production version, 1.0, in August 2013.

Ember.js is a very opinionated framework, which means there is usually a single best way to implement a certain task. It adopts the "convention over configuration" philosophy, found in Ruby on Rails, that there should be a default way that the building blocks of the framework fit together without the application developer explicitly specifying so.

The most obvious useful application of that concept is naming. Instead of explicitly defining how components are tied together (for example, which controller a certain route uses), Ember.js adopts naming conventions that establish these connections. This cuts down hugely on boilerplate code while still providing a way to override the convention in the rare cases when this is needed.

The flip side of naming conventions is that developers new to Ember might be baffled at first about how things work and feel frustrated, especially if documentation does not make these conventions crystal-clear. What's worse, newcomers might abandon learning Ember.js altogether and give up on all the joys that developing with such a wonderful framework gives, which would be a shame.

Why Ember.js?

Ember.js is designed for building ambitious web applications, as its motto states. It is without doubt the most comprehensive browser application framework out there, which makes it a wholesale solution for building client-side applications. There is no need to add libraries that plug into it to access different functionality, like managing views or handling routes.

Although it is a comprehensive solution for building client-side applications, it is not a monolith block of code. It leverages several small JavaScript libraries, like router.js for routing, rsvp.js for promise handling, backburner for managing the run loop, and Handlebars for rendering templates.

BACKSTAGE

DIAL M FOR MODEL

All frameworks inspire lively discussions about what should form part of the core framework, and which features should be provided by external libraries. The justification for not having something in the core usually boils down to two basic arguments: either the feature is not integral to most of the applications the framework is considered ideal for, or the requirements for that feature would be so vastly different in each case that it is nearly impossible – or does not make sense – to provide a general solution in the framework.

For quite a while, the notable missing piece from Ember was the "M" from MVC, the model layer. There were several model libraries that could fill that hole, while certain applications (the biggest of which is Discourse, the flagship OS Ember application) chose to implement their own.

The reason for not having a "persistence" layer implementation built into the core is probably the fact that it is the most easily separable piece of architecture.

That said, Ember Data can now be considered the official Ember model layer, and starting from version 1.13, they are in lockstep, so their latest stable (major and minor) versions are the same. Ember 2.3, for example, will be most perfectly matched by Ember Data 2.3.

Why Ember.js (continued)

Ember.js is no doubt an opinionated framework. These opinions are woven into its core, and consequently, more often than not, there is usually a best way for each task, and a best place for each piece of code.

That doesn't mean that the developer becomes a code monkey, left to follow the framework's bidding - far from it! Instead, once working with the framework becomes second nature, the developer's creative thought cycles – and precious development time! – are freed up to focus on business logic problems, improving user experience, and writing robust, flexible code.

Its strong, opinionated nature has another great benefit: developers inheriting an Ember project or contributing to an open source Ember application do not need a lot of ramp-up time to get intimate with the code and figure out how the application is laid out and exactly how it works. Firm best practices make developers' lives easier, and save time when getting started or handing over projects.

I'm sure I've roused your curiosity about where Ember's opinions lie, so let me give you some high-level, uncompromising points of view:

1. **URLs are the most distinctive feature of web applications, whether**

   server- or client-side. Thus, any framework worth its salt has to support URLs.

   Putting development time and thought where its metaphorical software mouth is, Ember.js has an outstanding router with a wide range of features, clearly thought-out and refined along its development process.

   It supports different location types and query parameters out-of-the-box. Each route defined in the routing table begets a full-fledged route object. Its callbacks (also called route "hooks") allow for loading data from the backend, checking authorization to view the page, redirecting to another route, handling user actions, and many more.

2. **Convention over Configuration**

   Discussed in more length at the beginning of this chapter, Convention over Configuration is a chief design principle in Ember. The major building blocks (routes, controllers, templates) are held together by a naming schema derived from route names.

3. **Only code that is worth writing should be written**

   Made possible by “Convention over Configuration”, Ember goes to great lengths to relieve developers from having to write code that can be deducted by the framework. One example of this is generating the right kind of controller for a route on the fly.

4. **Templating with a tightly-integrated library, Handlebars**

   Handlebars is a string-based templating language with its own set of helpers, not tied to the DOM. When rendering HTML from a list of objects, there is, *most of the time*, a one-to-one mapping between the object and a corresponding HTML tag, like rendering a `<li>` for each item in a `<ul>`.

However, this might not always be the case, and inserting template logic in DOM nodes can lead to tricky situations (imagine wanting to create two `<li>`s in a `<ul>` for each item in an array).

While a string-based templating language is simpler to implement (think about server-side rendering), it has its own shortcomings. Not being aware of the DOM has a performance penalty, and special helpers need to be used to set node attributes (`class`, `href` or `src`, for example).

Ember's answer to this is Glimmer, a library that uses the Handlebars parser but is aware of the DOM. It landed in Ember 2.10 and opens up many possibilities.

5. **Stability without stagnation**

   During the march to Ember 2.0 the core team could have decided to part ways with concepts and syntaxes that proved obsolete or too complex. However, coming up with totally new APIs would have broken a lot of existing 1.x Ember apps out there. On the other hand, had they hung onto old ways of doing things, better ideas might not have been adopted.

   Not wanting to do either, the Ember core adapted the philosophy of "Stability without stagnation". New features were integrated into the framework over time, in minor 1.x versions, when they were found robust enough to be merged. Old features were phased out gradually, leaving Ember devs ample time to migrate their applications. Having a strict, 6-week release strategy made the whole process easy to foresee and plan for.

   This method proved very effective and has remained in place for the 2.x development process.

About this book

This book will get you over the initial bump of comprehending Ember and give specific advice on how typical tasks should be carried out to work *with* the framework, not against it. The main concepts and building blocks of Ember will be introduced by building an actual application. The application will get shinier (and/or more robust) with each chapter, helped by the Ember concept introduced in the same chapter. Not only will you learn about the Ember concept, you'll see how it can be used in a real application.

Similar to appreciating a rock and roll concert in its entirety, chapters in this book will have musically-themed sections that keep the flow of new information constant and manageable:

"Tuning" sections fine tune you, as well as preview what you will accomplish by the end of the chapter. They also give you background information to start your application - consider these the right musical notes for you to start jamming.

"Backstage" sections (like the one above, embedded in the "Why Ember.js?" section) go into more detail about Ember concepts needed to understand the feature under development. They're set apart with their title and styling.

"The road says" sections help keep you synced with the application if you build the app as you read through the book. Typically, they describe files to be deleted, css classes to be added for tests to pass and the like. If you don't develop the app in tandem, don't bother reading these. I chose to put these in their separate section at the end of the chapter so that the flow of explanation is not broken by minutia.

The canonical name of the road is road manager but for rockstars like us, it is simply "the road".

"New Song" sections are at the end of each chapter. They sum up the building of the application up to that point, as well as set the expectation of what's to come in the repertoire.

"Encore" is usually called "Appendix" in less musical books. Some Ember concepts that are useful to know but don't quite fit in the flow of building the application are found here.

Is this book for you?

This book teaches Ember.js from the very basics and does not assume any prior knowledge about the framework. It also does not require you to be a JavaScript whiz, as the code snippets are easy to understand with only basic JavaScript proficiency.

Consequently, this book might be for you if:

- You have not done anything in Ember yet and are eager to learn.
- You have started learning about Ember and would like to accelerate the learning process.
- You have played with Ember, reading a couple of blog posts here and there, but feel a bit disorientated.
- You have developed – or are developing – Ember applications, but have come across cases where you felt you had to fight the framework. Or, you just wonder whether what you do is in line with "the Ember way".
- You "get (most of) it," but are a pragmatic person who learns best by following along the development process of an application.

On the other hand, this book is probably not for you if:

- You are proficient in Ember, understand all the concepts and have built multiple applications in it.
- You hate rock & roll so much you can't even stand seeing it being used in an application.

Ambitious goal for an ambitious framework

The goal of this book is to teach you how to develop an Ember.js application. If I do a good job, you will come away understanding how the different components come together to make everything work. The roles and responsibilities of these components will be clear to you and you will be able to respond to the typical questions arising during development, both small-scale and large.

Given that most Ember applications are similar enough in their structure and operation (see above in the "Why Ember.js" section), that knowledge will also enable you to understand other Ember applications and be able to contribute to them or take over their development.

The application we are going to build

I strongly believe in learning by doing, in this case by building an actual application to demonstrate concepts with. Chapter by chapter, we'll develop an application called Rock & Roll which serves to catalog your music collection.

Here are a few features we'll include:

- Show a “dashboard” where all bands are listed on the left (with the selected band highlighted) and the songs belonging to the selected band on the right.
- Build a star-rating widget to rate songs.
- Implement a streamlined – yet very simple – flow to add a new band and then start adding songs to it.
- Hook up the application to a remote API.
- Prevent users from losing unsaved changes by warning them when they're about to leave a form that's been edited
- Safeguard our application by writing a bunch of tests, both high- and low-level.

- Sort the songs based on multiple properties, selectable by the user.
- Search songs.
- Make sure all band names and song titles have the correct case, no matter how the user entered them.
- Add a handful of animations to give more nuanced feedback about user actions
- Extracting a component in the application into an Ember addon
- Use the state-of-the-art JavaScript version, ES2015
- Show how to deploy the app onto various platforms

I encourage you to work along the book and build the app yourself. You can type out all the code snippets or you can copy-paste (some of) them from the book. If you do so, be aware that this is easier done from some PDF viewers than from others.

There are some code snippets in the book that only serve demonstrational purposes but are not part of the application. They can be demonstrated by not having a comment header that designates the file to modify.

Here is one such code snippet:

```js
export default Controller.extend({
  queryParams: ['search'],
  search: '',
})
```

JS

For contrast, here is an example of another one, with a comment header, that indicates it does need to be included in the application we're building:

```js
// app/router.js
Router.map(function() {
  this.route('bands');
  this.route('songs');
});

export default Router;
```

JS

Where it facilitates comprehension, I use "diff signs" in code snippets to show which lines were added and which removed. Added lines start with a +, removed ones use a −, as below:

```hbs
  <!-- app/templates/bands/band/songs.hbs -->
  <ul class="list-group songs">
    (...)
    {{#each model.songs as |song|}}
      <li class="list-group-item song">
        {{song.title}}
+       {{star-rating item=song rating=song.rating setAction="updateRating"}}
-       <span class="rating pull-right">{{song.rating}}</span>
      </li>
    {{/each}}
  </ul>
```

HBS

Errata

Should you come across any typos, unclear explanations or missing pieces, please create an issue in the public errata repository for the book.

Next song

To start things off, we'll install Ember CLI, the outstanding tool to manage Ember applications, and create our project.

CHAPTER 2

Ember CLI

Tuning

Ember CLI was started in November 2013 by Ember Core team member Stefan Penner and has since emerged as *the* tool for developing Ember apps. It establishes conventions on where to store your files, makes it a breeze to add libraries to your bundle, build and test your application, and many more. Above this, it's super fast and rebuilds and reloads your app whenever any of your development files changes. Let's see how to work with it.

Setting up Ember CLI

`ember-cli` is an npm (Node Package Manager) package, which means you will need to have Node.js installed on your machine.

If you don't yet have Node.js, head over to nodejs.org to download the package by following their instructions. You could also use your favorite package manager for your operating system.

npm should be installed as part of Node.js, so once the installation has finished, you should be able to run the following commands:

```
$ node -v
$ npm -v
```

and have the respective versions printed out.

Ember CLI needs at least node version 4.5 and npm 2.1.8, so don't proceed if your install versions are older.

You'll also need git, the version control system, for Ember CLI to create a new project, so make sure it's installed on your system.

Once you have the environment Ember CLI runs in, let's install the `ember-cli` package itself:

```
$ npm install -g ember-cli@2.18.0
```

The `-g` flag instructs npm to install ember-cli globally, so that it is available from everywhere on your machine.

The installation might take a couple of minutes but once it has run to completion, you should be able to query the version you just installed:

```
$ ember -v
```

`ember` is the executable that the `ember-cli` package installs, which we will use heavily throughout the tour.

You should see something like the following as the output of `ember -v`:

```
version: 2.18.0
node: 8.9.0
os: darwin x64
```

Make sure you have the latest Ember CLI, so that the `ember` command we'll use throughout the book generates the right things and works the same way on your machine as in the book. The `npm install -g ember-cli@2.18.0` step above should have taken care of this but if you already had it installed and skipped that step, you should double check with `ember -v`.

(If you upgrade Ember CLI from a previously installed version, please follow the instructions here.)

Creating our application

Once we have all the ingredients prepared, let's start cooking.

(If you'd like to install your project with Yarn, which I highly recommend but don't insist on, read the "Creating your application with Yarn" section in the Encore)

Go to the folder where you would like to store your 'Rock & Roll with Ember.js' application and create a new Ember CLI project:

```
$ ember new rarwe --no-welcome
```

This will create the directory structure Ember CLI works with and install the npm packages. The `--no-welcome` switch tells Ember CLI not to generate a welcome page.

Now would be a good time to fetch some coffee (unless you used yarn!).

Taking a look at a new project

Let's take a look at the created files:

```
rarwe
├── app
│   ├── app.js
│   ├── components
│   ├── controllers
│   ├── helpers
│   ├── index.html
│   ├── models
│   ├── resolver.js
│   ├── router.js
│   ├── routes
│   ├── styles
│   └── templates
├── config
│   ├── environment.js
│   └── targets.js
├── ember-cli-build.js
├── node_modules
│   ├── (...)
├── package.json
├── public
│   ├── crossdomain.xml
│   └── robots.txt
├── README.md
├── testem.js
├── tests
│   ├── helpers
│   ├── index.html
│   ├── integration
│   ├── test-helper.js
│   └── unit
└── vendor
```

The `app` directory contains the code of the application. Each type of building block has its own folder (for example, routes, templates, etc.) under this directory and this is where you will spend most of your time (and write most of the code) as a developer.

`config/environment.js` contains the configuration of your application. Several things (like the rootURL) of the application are already defined for you, but you can also add new ones specific to your application.

In `config/targets.js` you can define what browser versions are supported for your app. This allows the JavaScript transpiler, Babel, to conditionally polyfill browser features depending on whether the supported browsers you specified support it natively or not.

`ember-cli-build.js` contains the build configuration of your project. It is responsible for recompiling all assets when a project file changes, and then outputs one bundle that contains all JavaScript code and one for the CSS. It runs all preprocessors, copies files from one place to another, and concatenates 3rd-party libraries. This processing is called the asset pipeline.

`package.json` defines the npm packages you need during development, for example, the `ember-cli` package itself.

The current contents should be the following:

```js
// package.json
{
  "name": "rarwe",
  "version": "0.0.0",
  "description": "Small description for rarwe goes here",
  (...)
  "devDependencies": {
    "broccoli-asset-rev": "^2.4.5",
    "ember-ajax": "^3.0.0",
    "ember-cli": "~2.18.0",
    "ember-cli-app-version": "^3.0.0",
    "ember-cli-babel": "^6.6.0",
    "ember-cli-dependency-checker": "^2.0.0",
    "ember-cli-eslint": "^4.2.1",
    "ember-cli-htmlbars": "^2.0.1",
    "ember-cli-htmlbars-inline-precompile": "^1.0.0",
    "ember-cli-inject-live-reload": "^1.4.1",
    "ember-cli-qunit": "^4.1.1",
    "ember-cli-shims": "^1.2.0",
    "ember-cli-sri": "^2.1.0",
    "ember-cli-uglify": "^2.0.0",
    "ember-data": "~2.18.0",
    "ember-export-application-global": "^2.0.0",
    "ember-load-initializers": "^1.0.0",
    "ember-resolver": "^4.0.0",
    "ember-source": "~2.18.0",
    "eslint-plugin-ember": "^5.0.0",
    "loader.js": "^4.2.3"
  }
}
```
JS

The `node_modules` folder is where npm packages are installed.

The contents of the `public` folder are going to be copied with the same path and the same content to the final build. Fonts, images, and a sitemap are good candidates to be placed here.

`testem.js` contains testing configurations (Ember CLI uses the `testem` package to run the test suite in various ways).

Finally, `tests` contains the tests for your application. They will be covered extensively in the Testing chapter.

(If you used yarn to create the project, you'll also see a `yarn.lock` file that yarn uses to lock down the version of each package.)

We now have a high-level understanding of what Ember CLI created and for which ends, and can thus start developing our application.

Next song

To warm up, we'll display a pre-defined list of songs.

That gives us the opportunity to learn about templates, the snippets that render the HTML that compose a page. Ember.js uses Handlebars, so a very brief introduction to it is in order. We'll then see how Ember brings templates to life by establishing bindings between the properties in the templates and the DOM.

Get ready to rock!

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.

The first template

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

```
  <!-- app/templates/application.hbs -->
- <h2 id="title">Welcome to Ember</h2>

- {{outlet}}
+ <ul>
+   {{#each model as |song|}}
+     <li>{{song.title}} by {{song.band}} ({{song.rating}})</li>
+   {{/each}}
+ </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 library of choice is the small and elegant Handlebars. In 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: function() {
    var blackDog = EmberObject.create({
      title: 'Black Dog',
      band: 'Led Zeppelin',
      rating: 3
    });

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

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

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

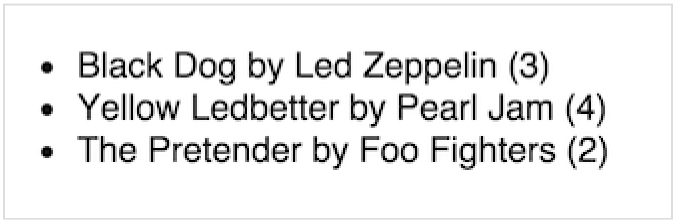
JS

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:



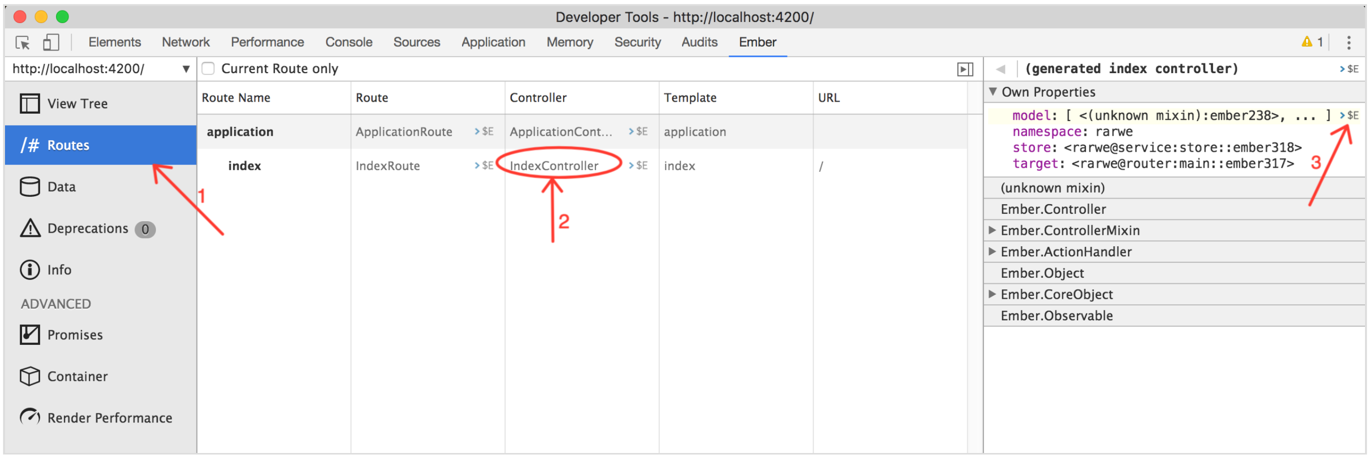
- Black Dog by Led Zeppelin (3)
- Yellow Ledbetter by Pearl Jam (4)
- The Pretender by Foo Fighters (2)

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)



Route Name	Route	Controller	Template	URL
application	ApplicationRoute	ApplicationCont...	application	
index	IndexRoute	IndexController	index	/

```
(generated index controller)
▼ Own Properties
  model: [ <(unknown mixin):ember238>, ... ]
  namespace: rarwe
  store: <rarwe@service:store::ember318>
  target: <rarwe@router:main::ember317>
(unknown mixin)
Ember.Controller
▶ Ember.ControllerMixin
▶ Ember.ActionHandler
Ember.Object
▶ Ember.CoreObject
Ember.Observable
```

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';
```

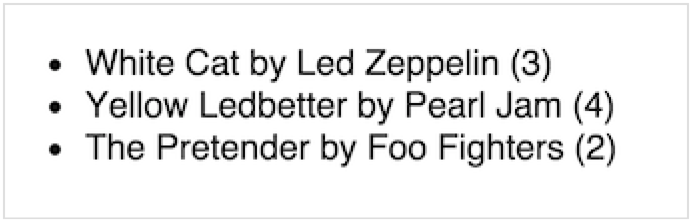
When we do that, we get the following error:

```
"Assertion Failed: You must use Ember.set() to set the `title` property (of
[object Object]) 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');
```

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:

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

... and add it to the existing ones:

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

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

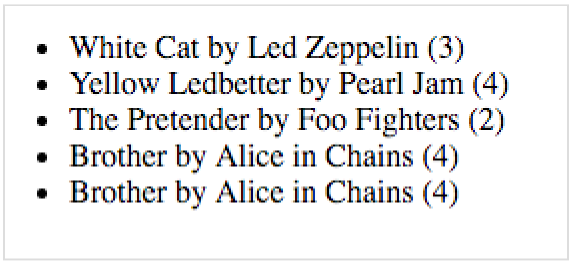
Ember.js takes Handlebars templates and sprinkles magic fairy dust on them to make them "auto-updating", as we have just seen with the song title. It does this by observing the arrays (the array that the `model` function returns) and properties (each song title, band and rating) referenced in the template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

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

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, two of those the new song, 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:

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

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

`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:

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

+ var Song = EmberObject.extend({
+   title: '',
+   band: '',
+   rating: 0
+ });

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

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

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

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

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:

```
var 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 to use an HTML framework. We will use Bootstrap (I know, boo).

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. Leveraging an addon is a lot simpler, so that's the one we're going to go with:

```
ember install ember-bootstrap
```

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

```
// package.json
{
  "name": "rarwe",
  (...)
  "devDependencies": {
    "broccoli-asset-rev": "^2.4.5",
    "ember-ajax": "^3.0.0",
+   "ember-bootstrap": "^1.0.0-rc.3",
    (...)
  }
}
```

JS

Running the default blueprint adds ember-bootstrap's build configuration into `ember-cli-build.js`:

```js
// ember-cli-build.js
/* eslint-env node */
const EmberApp = require('ember-cli/lib/broccoli/ember-app');

module.exports = function(defaults) {
  var app = new EmberApp(defaults, {
+   'ember-bootstrap': {
+     'bootstrapVersion': 3,
+     'importBootstrapFont': true,
+     'importBootstrapCSS': true
+   }
  });

  (...)
  return app.toTree();
};
```

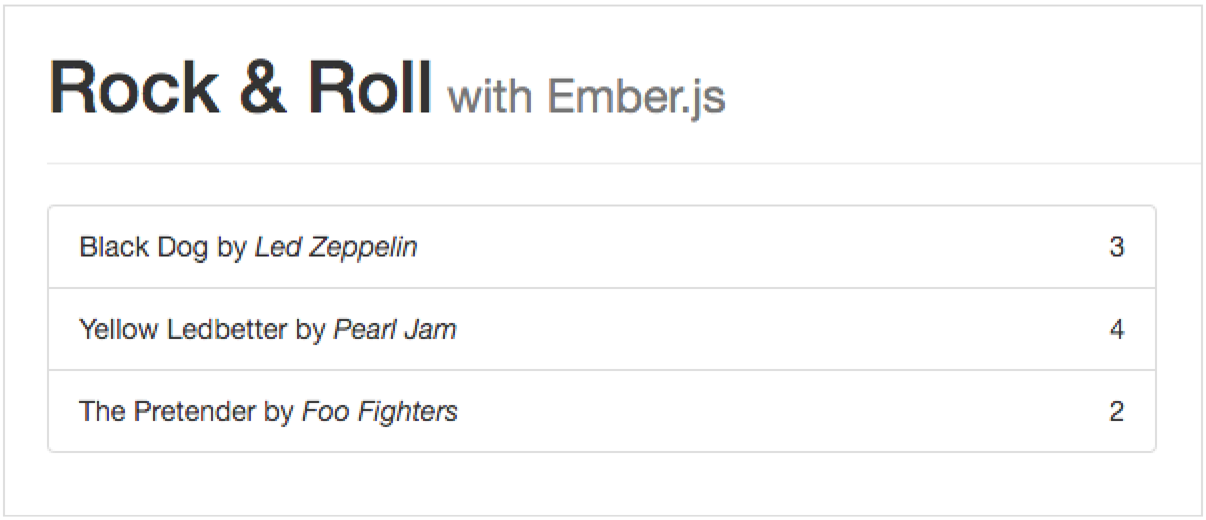
The `ember serve` process should be restarted because we have added new dependencies to the `package.json` file.

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="container">
  <div class="page-header">
    <h1>Rock & Roll<small> with Ember.js</small></h1>
  </div>
  <div class="row">
    <div class="col-md-6">
      <ul class="list-group songs">
        {{#each model as |song|}}
          <li class="list-group-item">
            {{song.title}} by <em>{{song.band}}</em>
            <span class="rating pull-right">{{song.rating}}</span>
          </li>
        {{/each}}
      </ul>
    </div>
  </div>
</div>
```

HBS

Once the app is rebuilt, it now looks fabulous:



Rock & Roll with Ember.js

Black Dog by *Led Zeppelin* 3

Yellow Ledbetter by *Pearl Jam* 4

The Pretender by *Foo Fighters* 2

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.

CHAPTER 4

Routing

Tuning

Robin Ward, a main developer of Discourse, the flagship open-source Ember application, calls Ember a "Browser Application Framework".

He doesn't like Ember being called a framework for building Single Page Applications (SPAs), because the name "SPAs" implies that the application only has one URL and communication with the app does not change it. That starkly contrasts with how Ember works.

He also doesn't like it being called an MVC framework, because client-side MVC is very different from server-side MVC. A lot of back-end developers who are well-versed in server-side MVC apply these server-side concepts to the front-end, which ends up setting back their learning.

In that spirit, I'll call the state of the application, the ensemble of the views a "page", which is described by the page's URL. URLs are the serialized form of the application's state and deserialization is handled by the routing mechanism.

As I hope will be revealed in the next couple of chapters, Ember places a lot of emphasis on routes, and it's the most important thing to get right. Routes can do wonderful things for you, if you know how to talk to them.

What do we want to achieve?

Let's start with something simple to introduce routes. When the user navigates to `/bands` in the Ember application, we want to show the list of bands, fetched from the backend (even if that backend is, for the time being, living in memory). The same goes for songs: we want them to be displayed in a list when the URL is `/songs`. Finally, we want to be able to navigate between them by clicking links.

New styles

We'll need to add a couple of css rules for the page header and the navigation links introduced in this chapter (feel free to add them now to your `app.css`):

```css
/* app/styles/app.css */
.page-header a h1 {
  color: #555;
}

.navbar-default .navbar-nav > li > a.active,
.navbar-default .navbar-nav > .active > a:hover,
.navbar-default .navbar-nav > .active > a:focus {
  color: #555;
  background-color: #e7e7e7;
}
```

Routes set up data

The first step is to define the routes which will serve as the backbone of the application. In fact, one could get a decent, high-level overview of the application by looking at the "routing table." This can serve as a starting point to further explore the application.

Routes reside in the `app/router.js` file, so let's bring it up in our editor:

```js
// app/router.js
import EmberRouter from '@ember/routing/router';
import config from './config/environment';

const Router = EmberRouter.extend({
  location: config.locationType,
  rootURL: config.rootURL
});

Router.map(function() {
});

export default Router;

```

First, we import the configuration from the `config/environment` module via a relative import. A relative import is one which begins with a `.` or a `..` and follows the rules of relative paths in UNIX operating systems. The absolute module names would be `rarwe/router` and `rarwe/config/environment`, respectively, so `./config/environment` is a functioning module import from `rarwe/router`.

Now, let's peek inside the configuration to see what value `locationType` has:

```js
// config/environment.js
module.exports = function(environment) {
  var ENV = {
    (...)
    rootURL: '/',
    locationType: 'auto',
    (...)

    APP: {
      // Here you can pass flags/options to your application instance
      // when it is created
    }
  };
  (...)
}
```

So `auto` is assigned to the router's `location` property. This specifies how the current route of the application will be manifested in the URL. Its possible values are `history`, `hash`, `none` and `auto`.

`history` uses the browser's `history.pushState` (and `history.replaceState`) to update the URL. If the current route is `band.songs` for pearl-jam, the URL is `http://rock-and-roll-with-ember.js/bands/pearl-jam/songs`.

`hash` relies on the `hashchange` event in the browser and uses the `#` in the URL. If the current route is `band.songs` for pearl-jam, the URL is `http://rock-and-roll-with-ember.js/#/bands/pearl-jam/songs`.

`none` does not affect the URL in any way. Though this may not seem to serve much purpose, it comes handy in testing or when the Ember app is embedded inside a page and updating the URL as the user interacts with the Ember app is undesirable.

`auto` uses `history` location if the browser supports it. If it does not, it tries the `hash` location. If even the `hash` location is not supported, it falls back to `none`.

Let's generate two routes in our application, one to display bands and another one to list songs.

```
$ ember generate route bands
$ ember generate route songs
```

That does three things.

First, the routing configuration in the `router.js` file is amended with the new routes:

```js
// app/router.js

Router.map(function() {
+   this.route('bands');
+   this.route('songs');
});
```

Second, a corresponding, empty route module is created for each route:

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

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

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

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

The configuration in `router.js` only *specifies* the routes of the application. The behavior for these routes need to be fleshed out in their correspoding route objects, each one an extension of the `Ember.Route` class.

Finally, the route generator also creates the corresponding templates for the pair of routes. `app/templates/bands.hbs` for `app/routes/bands.js` and `app/templates/songs.hbs` for `app/routes/songs.js`.

The generated templates contain nothing but an `{{outlet}}` (see more about outlets below), so let's paste the following content into them:

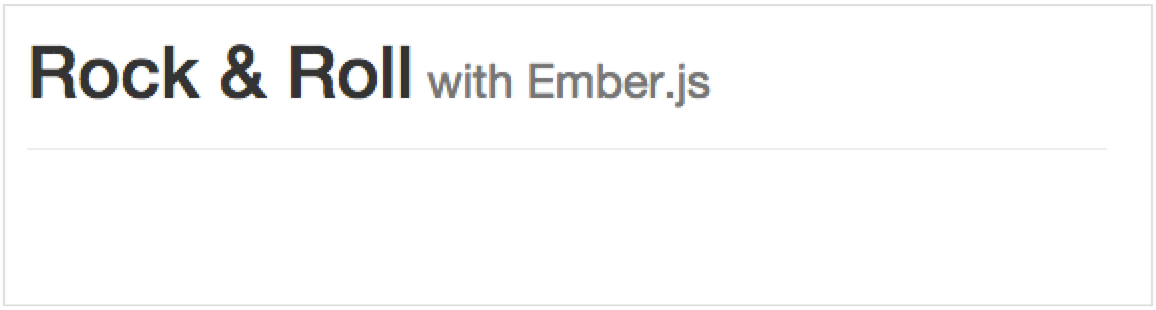
```hbs
<!-- app/templates/bands.hbs -->
<ul class="list-group">
  {{#each model as |band|}}
    <li class="list-group-item">{{band.name}}</li>
  {{/each}}
</ul>
```
HBS

```hbs
<!-- app/templates/songs.hbs -->
<ul class="list-group songs">
  {{#each model as |song|}}
    <li class="list-group-item">
      {{song.title}} by <em>{{song.band}}</em>
      <span class="rating pull-right">{{song.rating}}</span>
    </li>
  {{/each}}
</ul>
```
HBS

While we're dealing with templates, let's slim down the application template from the previous chapter so that it serves as a layout for the two other templates:

```
  <!-- app/templates/application.hbs -->
  <div class="container">
    <div class="page-header">
      <h1>Rock & Roll<small> with Ember.js</small></h1>
    </div>
    <div class="row">
      <div class="col-md-6">
-       <ul class="list-group songs">
-         {{#each model as |song|}}
-           <li class="list-group-item">
-             {{song.title}} by <em>{{song.band}}</em>
-             <span class="rating pull-right">{{song.rating}}</span>
-           </li>
-         {{/each}}
-       </ul>
+       {{outlet}}
      </div>
    </div>
  </div>
```
HBS

Our application recompiles successfully but we only see the header:


Rock & Roll with Ember.js

The router always enters the `application` template first and thus renders the `application` template. It can thus be considered the application layout. Inside that template, you can see an `{{outlet}}` definition. Outlets are slots in the template where content can be rendered from other routes.

Having the router activate the `application` route on all requests gives us a place to define operations that are necessary for the whole application. This might include choosing the language for the site, or fetching data that is needed to render a layout that's common across the pages of the site.

Now, here's an important fact: Subroutes, by convention, render their content into the main outlet defined by their parent templates. The main outlet is the `{{outlet}}` without a name. All routes are children of the application route, and consequently the application template defines where dynamic content can be rendered and only specifies markup that is the same, at least structurally, across the whole application.

In the above case, there are two additional templates defined, `bands` and `songs`. They render their content into the outlet defined in the application template. Both just render a list, displaying relevant data about each band and song, respectively.

That's great, but we're still staring at an empty page with just a header, so let's do something about this.

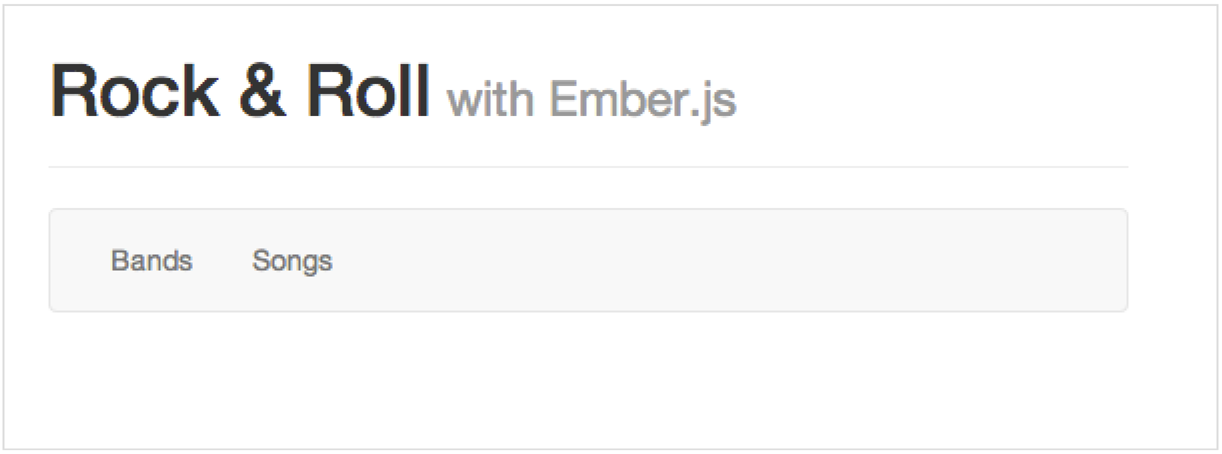
Moving around with link-to

We will quickly add two links to all pages by placing them into the application template. These will serve to navigate to the bands and songs pages. The classic way of transitioning to another route in Ember is the `link-to` helper. In its simplest form, you give the text of the link as the first parameter and the route name as the second.

Let's add a navigation header that contains two links to navigate between the routes. This should be added to the application template, so that it's present on all pages of the application.

```hbs
  <!-- app/templates/application.hbs -->
  <div class="container">
    <div class="page-header">
      <h1>Rock & Roll<small> with Ember.js</small></h1>
    </div>
    <div class="row">
      <div class="col-md-6">
+       <nav class="navbar navbar-default" role="navigation">
+         <div class="collapse navbar-collapse">
+           <ul class="nav navbar-nav">
+             <li>{{link-to "Bands" "bands"}}</li>
+             <li>{{link-to "Songs" "songs"}}</li>
+           </ul>
+         </div>
+       </nav>
        {{outlet}}
      </div>
    </div>
  </div>
```

Now when we load the application at '/', here is what we see:



The application template is rendered with the proper links in the header. Clicking on them will take us to the `bands` or `songs` routes, respectively. Let's now take a look at these templates.

Using the model property

When we pop open either the `bands` or the `songs` templates, we'll see they refer to a `model` property:

```hbs
<!-- app/templates/bands.hbs -->
<ul class="list-group">
  {{#each model as |band|}}
    <li class="list-group-item">{{band.name}}</li>
  {{/each}}
</ul>
```

HBS

```hbs
<!-- app/templates/songs.hbs -->
<ul class="list-group songs">
  {{#each model as |song|}}
    <li class="list-group-item">
      {{song.title}} by <em>{{song.band}}</em>
      <span class="rating pull-right">{{song.rating}}</span>
    </li>
  {{/each}}
</ul>
```

HBS

...but that `model` property is not defined anywhere.

`model` is a special property that Ember automatically defines on the template's context, so it is available as `{{model}}` inside the template.

The underlying concept is that each template has a "principal thing" that it works on. In the case of a template that lists bands, the model is the list of bands. In the case of a template that allows editing of a user's profile, it would be the user object.

Let's define a few bands, then, so that they can be displayed.

Showing a list of bands

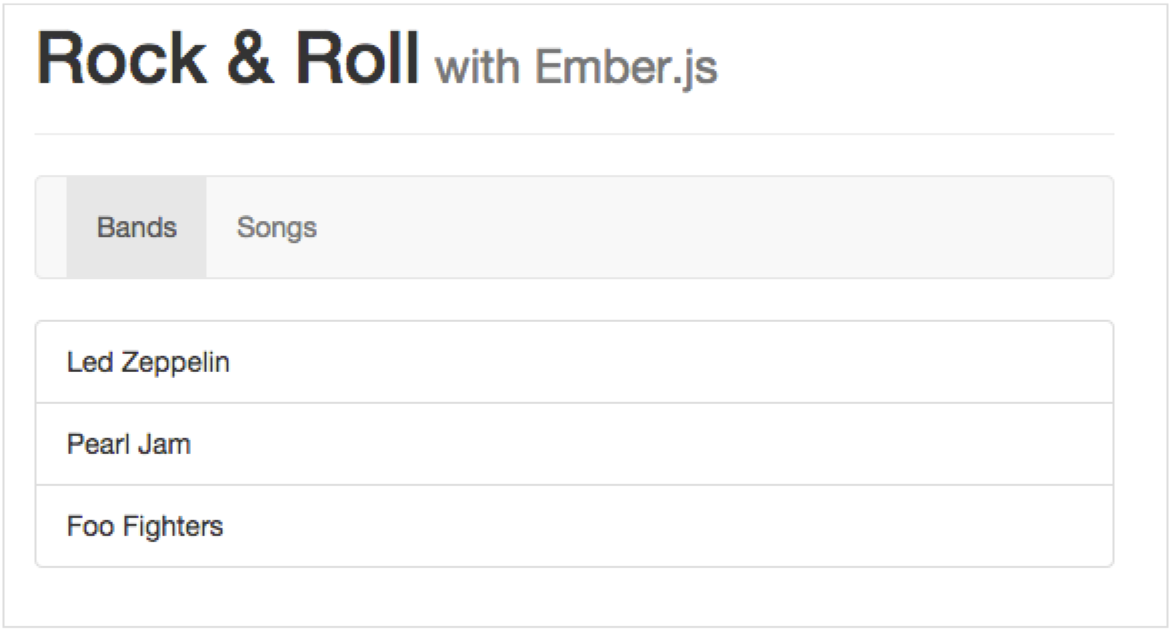
We create a `Band` class and a few instances just as we did in the previous chapter for songs:

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

+ var Band = EmberObject.extend({
+    name: '',
+ });

  export default Route.extend({
+   model: function() {
+     var ledZeppelin = Band.create({ name: 'Led Zeppelin' });
+     var pearlJam = Band.create({ name: 'Pearl Jam' });
+     var fooFighters = Band.create({ name: 'Foo Fighters' });
+
+     return [ledZeppelin, pearlJam, fooFighters];
+   }
  });
```

The route (which extends the `Route` class) has a single function called `model`. When the router enters that route, it calls this model function and its return value gets set as the `model` property *of the corresponding controller* (this happens behind the scenes).

The properties in the template need to be defined on the controller, since that is the context the template is rendered with.

The `bands` template loops through the `model` property, which is all of the bands. When our application gets rebuilt and reloaded, the band list is now rendered correctly:



Showing a list of songs

The next logical step is to do the same for songs.

We can simply cut-and-paste the content of the application route (`app/routes/application.js`) from the previous chapter.

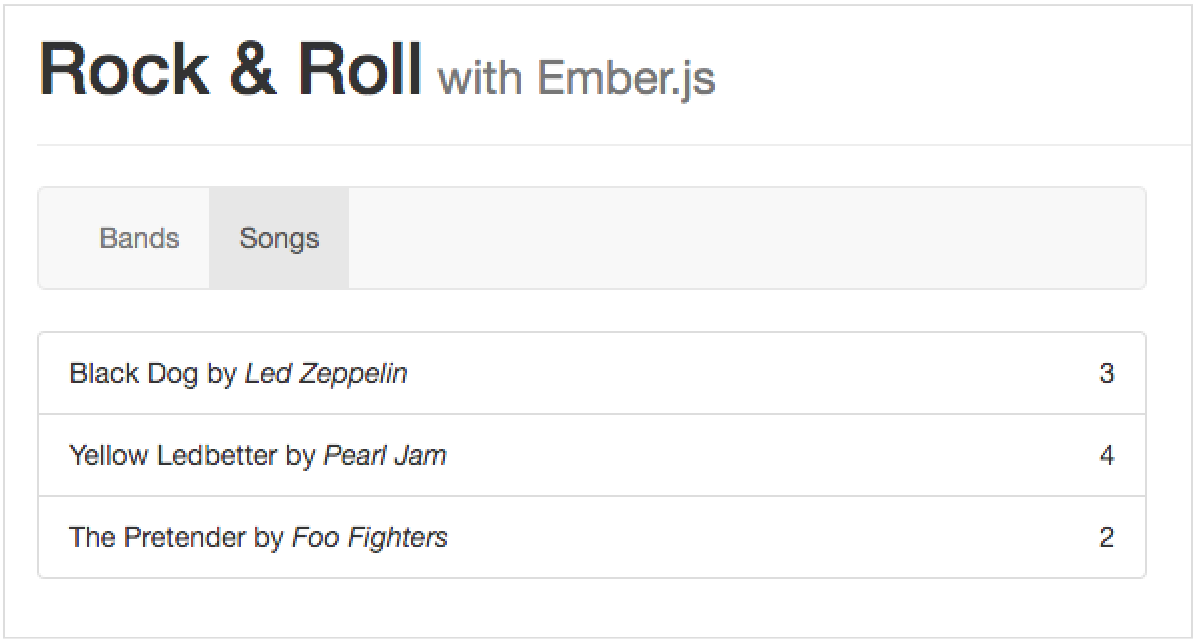
```js
  // app/routes/songs.js
  import Route from '@ember/routing/route';
  import EmberObject from '@ember/object';

+ var Song = EmberObject.extend({
+   title: '',
+   rating: 0,
+   band: ''
+ });

  export default Route.extend({
+   model: function() {
+     var blackDog = Song.create({
+       title: 'Black Dog',
+       band: 'Led Zeppelin',
+       rating: 3
+     });
+
+     var yellowLedbetter = Song.create({
+       title: 'Yellow Ledbetter',
+       band: 'Pearl Jam',
+       rating: 4
+     });
+
+     var pretender = Song.create({
+       title: 'The Pretender',
+       band: 'Foo Fighters',
+       rating: 2
+     });
+
+     return [blackDog, yellowLedbetter, pretender];
+   }
  });
```

JS

The application route should now be totally empty:

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

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

When the app reloads and we click on Songs, we correctly see the list of songs:


Rock & Roll with Ember.js

Black Dog by *Led Zeppelin* 3

Yellow Ledbetter by *Pearl Jam* 4

The Pretender by *Foo Fighters* 2

As you play with the app, notice how the URL changes as you click the links. It follows the state changes of your application and therefore becomes a representation of its current state. That means that you can bookmark and share the URL so that someone else opening that URL will see the same page.

That's what Robin Ward was thinking of when he said that Ember is not an SPA framework in the usual sense of the word. Other SPAs tend to think of URLs as nice-to-haves or don't even bother to have built-in URL support. On the other hand, routes are the very essence of Ember.js, the central concept the framework is built around.

The road says

"It makes no sense to have that empty application route lying around, why don't you just get rid of it?"

The road is right, as the file now only contains the default skeleton, we should remove it:

```
$ rm app/routes/application.js
```

Oops, and let's make sure we also get rid of the associated test file:

```
$ rm tests/unit/routes/application-test.js
```

Next song

This is all fine and dandy, but it would make much more sense to display songs for each artist and not in one global list. The artists would be shown on the left and the songs for the selected artist on the right (or, in narrower viewports they would be collapsed to be displayed under each other).

This UI arrangement, having a list where activating one of the items displays detailed information about that one item and leaves the list in place, is a very common UI pattern and is called "master-detail view".

Ember.js leverages its nested routes to render the above nested UI, and its elegance is one of the things that sealed my fate with Ember for good. I hope to benignly infect you with my enthusiasm.

CHAPTER 5

Nested routes

Tuning

Let's take our primitive UI one step further, and make it so that the bands and the songs are displayed at the same time. List of bands on the left; list of songs for the selected band (or a placeholder box) on the right.*

In API design, a has-many association (a band has many songs) is very often represented in the URL with the children in the association after the identified parent resource. In our case, that would be `/bands/:slug/songs`.

It is aesthetically pleasing to see such URLs, but Ember.js provides a more pragmatic benefit to those willing to heed its advice. Nested URLs automatically set up a nested UI, a so-called “master-detail” view, that is a widely used UI pattern. Let's see how.

* Or list of bands on top, list of their songs below it on narrower viewports.

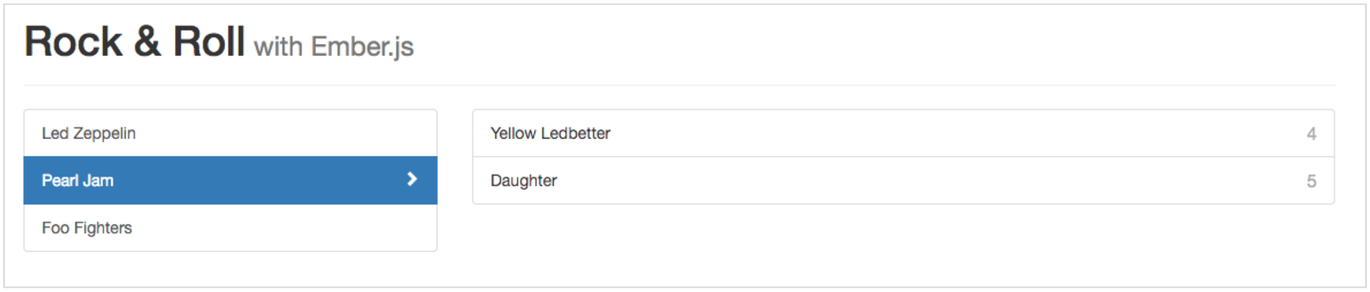
New styles

To make things look nicer, let's add the following rules to our stylesheet:

```css
/* app/styles/app.css */
(...)

.songs .rating {
  color: #aaa;
  font-size: 1.1em;
}

.empty-list {
  min-height: 100px;
  text-align: center;
}

.empty-list .empty-message {
  display: inline-block;
  color: #555;
  line-height: 100px;
}

.band-link .pointer {
  float: right;
  display: none;
}

.band-link.active .pointer {
  display: block;
}
```

Defining the nested routes

Here is a mockup of what we want to build:


Rock & Roll with Ember.js

Led Zeppelin	
Pearl Jam	
Foo Fighters	

Yellow Ledbetter	4
Daughter	5

Our routes currently look like this:

```js
// app/router.js
(...)
Router.map(function() {
  this.route('bands');
  this.route('songs');
});

export default Router;
```

These two routes have no relation to each other; they exist independently. What we want is to show songs in relation to the band they belong to, and we want the URL to reflect that.

One such URL would be `/bands/pearl-jam/songs` where the middle segment `pearl-jam` identifies the band.

In Ember, you can nest routes in the router map, and each nested level adds its path to the whole. So as you descend in the router map through nested routes, at each level, the path of that route is added to the accumulated path.

The navbar from the previous chapter needs to go, since we are going to radically change the routes of our application, and the simple `songs` route will be no more. Just delete the `<nav>` tag and the `col-md-6` wrapper div. It should now look like this:

```hbs
<!-- app/templates/application.hbs -->
<div class="container">
  <div class="page-header">
    <h1>Rock & Roll<small> with Ember.js</small></h1>
  </div>
  <div class="row">
    {{outlet}}
  </div>
</div>
```

Okay, now we're ready to write our first nested route definition.

Since I'm going to give a step-by-step definition for better comprehension, tweaking the router map through Ember CLI generators would produce a lot of additional files that would need to be deleted later. Let's modify the router map manually this time, by moving the `songs` route under the `bands` route:

```js
// app/router.js
Router.map(function() {
  this.route('bands', function() {
    this.route('songs');
  });
});
```

If you start from the top with an empty path (`/`), the first route is `bands`. If a path is not explicitly defined for a route, it is equal to its name. So in this case, the path for `bands` is `bands`, which gets added at the end to give us `/bands`. We can go down one level more, to the `songs` route. Again, no path is defined explicitly for `songs`. Adding this to the accumulated path gives us `/bands/songs`.

The router map also defines the set of URLs that are handled by the Ember application. You can think of it as walking a graph, more precisely, a tree: any URL that can be produced by starting from the top-level and stopping at points along a certain route, adding the path for each stop as seen in the previous paragraph, is served by the application.

The set of URLs handled by the application are currently:

- `/` (the top-level, implicit, `index` route)
- `/bands` (stepping down to the `bands` route)
- `/bands/songs` (descending to the `bands` and then the `songs` route)

With that understanding, let's see if we can now create the type of URLs we want. Looking at `/bands/pearl-jam/songs`, we see it is not yet covered. We're missing the middle segment, which designates the specific band.

Let's introduce a `band` route then. The path this route generates needs to be dynamic, since we want this route to stand for *any* band, not just Pearl Jam. It also has to cover `/bands/led-zeppelin/songs`, `/bands/foo-fighters/songs`, and so on. This is called a dynamic path (or dynamic segment) in Ember, and is distinguished by prefixing the path with a `:`.

So this time, we also have to pass this option to the route generator:

```
$ ember generate route bands/band --path=':slug'
```

By passing `bands/band` as the route name to be generated, we indicate that the new route should be called `band` and it should be under the `bands` route. However, Ember CLI can't tell that we want `songs` nested under `band`, and has thus generated something like this:

```js
// app/router.js
(...)
Router.map(function() {
  this.route('bands', function() {
    this.route('songs');
  });

  this.route('band', {
    path: ':slug'
  });
});
```
JS

So let's fix this up to have the following shape:

```js
// app/router.js
(...)
Router.map(function() {
  this.route('bands', function() {
    this.route('band', { path: ':slug' }, function() {
      this.route('songs');
    });
  });
});
```

By defining a dynamic path (one that starts with a `:`), our router now maps every possible band URL to a route object in our application. With this in place, the following paths are all mapped:

- `/`
- `/bands`
- `/bands/pearl-jam`
- `/bands/led-zeppelin`
- `/bands/pearl-jam/songs`
- `/bands/led-zeppelin/songs`

Furthermore, any band slug that goes in the middle segment is also mapped, by virtue of the `band` route having a dynamic segment.

Assigning a route level to a particular band also makes our router map – and thus our application – more extensible. Should we decide to also display albums for bands down the road, it is simply a matter of adding a route nested under the `band` route:

```js
// app/router.js
(...)
Router.map(function() {
  this.route('bands', function() {
    this.route('band', { path: ':slug' }, function() {
      this.route('songs');
      this.route('albums');
    });
  });
});
```

And then our app handles URLs for albums just like for songs:

- `/bands/pearl-jam/albums`
- `/bands/led-zeppelin/albums`

Full route names

So far, I referred to routes by their short names, the string that is the first parameter in route definitions, like `bands`, `band` and `songs`. All routes have a full, canonical name, too, that the Ember router uses to identify them.

The full name of a route is the concatenation of the short names on the path used to reach it. For example, the `songs` route can be reached by following the `bands`, `band` and then the `songs` route so it's full name is `bands.band.songs`.

Full names are crucial to understand as they are used both in Ember code to target routes (for example, as the parameter of the `link-to` helpers) and as the module name (and thus file path) generated for the route. The route object for the `bands.band.songs` route should be placed in `app/routes/bands/band/songs.js` and the template in `app/templates/bands/band/songs.hbs`. Most of the time, Ember CLI generators relieve you from having to know this as they generate entities correctly based on the router map but it is still important to understand this concept as misplacing code can lead to situations where the app doesn't behave as expected simply because the code is not executed.

I will continue to use short route names in explanations unless the point I want to get across is better demonstrated by using full names (which is going to be the case in most of this chapter).

Stepping through a router transition

Analyzing exactly what happens when the user visits a URL that was created by a nested route is fundamental to understanding how nested routes set up a nested UI, and probably one of the greatest "aha!" moments on your journey to Ember mastery. So let's see which routes are activated and in what order when a visitor of our site goes to `/bands/pearl-jam/songs`.

Before we do that, though, let's adapt our code to the now-existing relation between `Band` and `Song`. We bring over the definition of the Song class and the song instances from `app/routes/songs.js` and assign each song to its appropriate band:

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

  var Band = EmberObject.extend({
    name: ''
  });

+ var Song = EmberObject.extend({
+    title: '',
+    rating: 0,
+    band: ''
+ });

  export default Route.extend({
    model: function() {
+     var blackDog = Song.create({
+       title: 'Black Dog',
+       band: 'Led Zeppelin',
+       rating: 3
+     });
+
+     var yellowLedbetter = Song.create({
+       title: 'Yellow Ledbetter',
+       band: 'Pearl Jam',
+       rating: 4
+     });
+
+     var pretender = Song.create({
+       title: 'The Pretender',
+       band: 'Foo Fighters',
+       rating: 2
+     });
+
+     var daughter = Song.create({
+       title: 'Daughter',
+       band: 'Pearl Jam',
+       rating: 5
+     });

-     var ledZeppelin = Band.create({ name: 'Led Zeppelin' });
-     var pearlJam = Band.create({ name: 'Pearl Jam' });
-     var fooFighters = Band.create({ name: 'Foo Fighters' });
+     var ledZeppelin = Band.create({ name: 'Led Zeppelin', songs: [blackDog] });
+     var pearlJam = Band.create({ name: 'Pearl Jam', songs: [yellowLedbetter, daughter] });
+     var fooFighters = Band.create({ name: 'Foo Fighters', songs: [pretender] });

      return [ledZeppelin, pearlJam, fooFighters];
    }
  });
```

Now, with all our data and class definitions updated, we can start inspecting the router transition in detail.

First, the implicit `application` route is entered, as we previously saw in the Routing chapter. Next, the router will start matching segments of the URL to the specified routes. The first URL segment to be matched is `/bands`, which matches the `bands` route.

Routes are such a central piece of Ember's infrastructure that they have their own objects, descending from Ember's `Route` class, to hang your code on. When entering a route, the `model` hook of the matching route object is called.

For `bands`, it just returns all the bands:

```js
// app/routes/bands.js
export default Route.extend({
  model: function() {
    (...)
    return [ledZeppelin, pearlJam, fooFighters];
  }
});
```

In the next step, the `pearl-jam` URL segment is looked up. The dynamic `:slug` path in the `band` route is found, and must be matched. Since `:slug` in the route definition is dynamic, the matched URL segment, "pearl-jam" is passed in as a parameter to the model hook. This gives the application a chance to deserialize the URL segment and turn it into a model object that is meaningful to the application.

Here, the meaningful part is the `band` object that is identified by the slug:

```js
  // app/routes/bands/band.js
  export default Route.extend({
+   model: function(params) {
+     var bands = this.modelFor('bands');
+     return bands.findBy('slug', params.slug); // params.slug is now 'pearl-jam'
+   }
  });
```

`modelFor` is a truly great method which fetches the model of a parent route that had already been activated. Since the Ember router enters routes starting at the root and proceeds downwards, this means that any route that is “above” the current route in the tree can be used as the parameter for `modelFor`. Here, we passed `bands` to `modelFor` to access all the bands that had previously been fetched by the `bands` route. We then selected the one that has the slug that was extracted from the URL.

In order to find the band by its slug, it needs to have such a property, so let's open up `app/routes/bands.js` and extend our Band class with said property:

```
  // app/routes/bands.js
  import Route from '@ember/routing/route';
- import EmberObject from '@ember/object';
+ import EmberObject, { computed } from '@ember/object';

  var Band = EmberObject.extend({
    name: '',

+   slug: computed('name', function() {
+     return this.get('name').dasherize();
+   })
  });
  (...)
```

`computed` defines a so-called "computed property". Computed properties have a significant role to play in Ember apps, so let's go Backstage to find out more.

BACKSTAGE

COMPUTED PROPERTIES

Let's take the above `Band` model and extend it a bit:

```js
var Band = EmberObject.extend({
  name: '',
  language: '',

  slug: computed('name', function() {
    console.log('Recomputing slug');
    return this.get('name').dasherize();
  }),

  site: computed('slug', 'language', function() {
    console.log('Recomputing site');
    return 'http://bands.com/' + this.get('slug') + '.' +
this.get('language');
  })
});
```

JS

To define a computed property (CP), we need to call `computed`. The first set of parameters passed to it are the *dependent keys* of the CP. The last parameter is the function that computes the value of the CP. The value of the CP changes if and only if the value of *any* of the dependent keys changes, and is cached between such changes.

When the value of a dependent key changes, the CP is marked as stale. The next time we access its value, the computing function is run and its returned value becomes the new value of the CP.

Let's see some examples to make this clearer.

The value of the `slug` property depends only on `name`, while `site` depends on both `slug` and `language`. This also shows that computed properties can serve as dependent keys for other computed properties.

I added a couple of logging statements to the function bodies in order to see when each of the computed properties gets recomputed.

(You can also see and modify this example on JS Bin)

```
> var band = Band.create({ name: 'Mookie Blaylock', language: 'jsp' });
> band.get('site');

Recomputing site
Recomputing slug
< "http://bands.com/mookie-blaylock.jsp"
```

We create a new band and then fetch its `site` value. Since this is the first time the CP is accessed, the computing function needs to be run. In turn, that triggers the computation of the other CP, `slug`, since `site` depends on it.

Let's access the freshly computed CP again:

```
> band.get('site')
< "http://bands.com/mookie-blaylock.jsp"
```

As expected, no computing functions need to be run since none of the dependent keys changed. Let's change the property value that is at the root of both CPs, the band's name:

```
> band.set('name', 'Pearl Jam');
> band.get('site');

Recomputing site
Recomputing slug
< "http://bands.com/pearl-jam.jsp"
```

Since a change in `name` invalidates `slug` (directly) and `site` (indirectly), both CPs were recomputed. Finally, let's see what happens if I change the `language` the band's site is implemented in:

```
> band.set('language', 'php');
> band.get('site');

Recomputing site
< "http://bands.com/pearl-jam.php"
```

`slug` didn't need to be recomputed because its sole dependent key, `name` did not change. `site`, however, was, because `language` changed.

You may wonder why the properties which a CP depends on are called "dependent keys" and not "dependent properties". The reason is that not only can they be properties, but property paths too, as we will see soon.

There is another way to define CPs that leverages extending the prototype of Function:

```js
import EmberObject from '@ember/object';

var Band = EmberObject.extend({
  name: '',
  language: '',

  slug: function() {
    return this.get('name').dasherize();
  }.property('name'),

  site: function() {
    return 'http://bands.com/' + this.get('slug') + '.' +
this.get('language');
  }.property('slug', 'language')
});
```

JS

Calling `property` on a function with the dependent keys as parameters transforms a function into a CP. However, this syntax is now discouraged (and will soon be deprecated) as extending the prototype of core JavaScript classes is frowned upon. You will still find abundant examples of it, though, so it's useful to be able to quickly parse it.

So before we embarked on the "Computed properties" journey, we were talking about how the model hook in the following route finds the band whose slug appears in the URL:

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

export default Route.extend({
  model: function(params) {
    var bands = this.modelFor('bands');
    return bands.findBy('slug', params.slug);
  }
});
```

JS

The findBy method finds the first item whose property (passed as the first argument) matches the value passed as the second argument. Since slug is the hyphenated (or "dasherized") version of the band's name, this will work.

The next segment to match is /songs, which activates the bands.band.songs route (remember, route definitions create route objects whose names are prefixed with the name of their parent route) and thus calls its model hook.

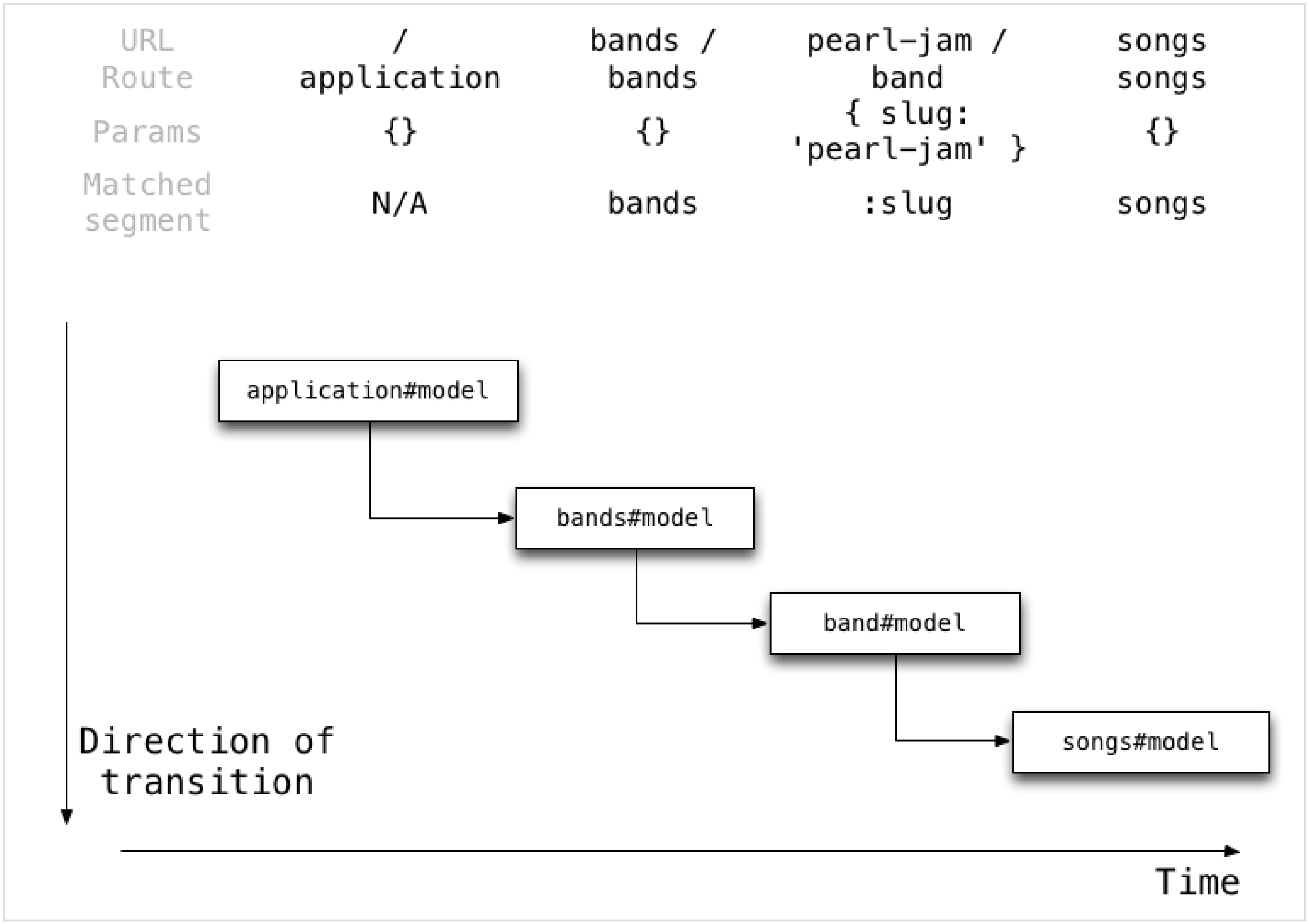
The route had already been defined in app/router.js, so let's a add a new file and manually write the following into it:

```js
// app/routes/bands/band/songs.js
import Route from '@ember/routing/route';

export default Route.extend({
  model: function() {
    return this.modelFor('bands.band');
  }
});
```

We use modelFor again, this time to retrieve the band that was returned one level higher.

The route transition is thus complete! Four routes were entered consecutively: application, bands, bands.band and bands.band.songs. Each resolved its model (if any) before the router descended to the next level. This simple yet powerful structure makes it possible for lower levels to access data that was retrieved higher up in the chain.

The next graph summarizes the analysis of the route transition:



The values returned by the model hooks will serve as data backing up the corresponding templates.

It is important to note that this full, top-down transition is only triggered when the app initially loads. If the user subsequently navigates to a route (via `link-to`), the router will make the transition directly, not descending through the parent routes on its way.

That also means that fetching data you only want to fetch once can be placed in the top-most, application route. We can be assured that this route, and thus the data fetching operation, will only happen once.

Nested templates

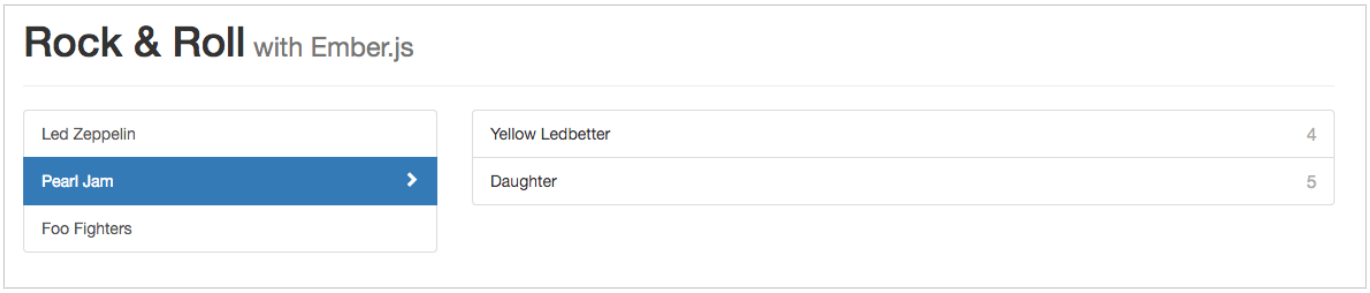
We now understand nested routes, so let's move on to the nested UI part. To prevent you having to go back, here's the route definitions again:

```
// app/router.js
(...)
Router.map(function() {
  this.route('bands', function() {
    this.route('band', { path: ':slug' }, function() {
      this.route('songs');
    });
  });
});
```

Before we delve into the templates, please note that since the application template now only contains the title bar and an empty outlet, you will not see anything of interest when you go to `http://localhost:4200/`. You need to add the `/bands` manually (so `http://localhost:4200/bands`) to see the list of bands.

We saw in the Routing chapter that templates must be named to exactly match their corresponding routes. That suggests we should have an `application`, a `bands`, a `bands.band` and a `bands.band.songs` template.

We also want to change how songs are displayed. We want to show them next to the list of bands and mark the band to which they belong, just as the comp showed at the beginning of the chapter:



Ember has a very neat way of supporting nested (or layered) UIs. If you define an `{{outlet}}` in the template of a route, it creates a slot for the children routes to render their content in. We'll leverage this in two ways.

First, when the user goes to '/bands', we want to show the list of bands, and we also want each name to link to the song list for the band. So let's open up our `bands` template and make it look like this:

```hbs
<!-- app/templates/bands.hbs -->
<div class="col-md-4">
  <div class="list-group">
    {{#each model as |band|}}
      {{#link-to "bands.band.songs" band class="list-group-item
band-link"}}
        {{band.name}}
        <span class="pointer glyphicon glyphicon-chevron-right"></span>
      {{/link-to}}
    {{/each}}
  </div>
</div>
<div class="col-md-8">
  {{outlet}}
</div>
```

The way we use the `link-to` helper is novel in two ways. First, we use it with a "closing tag", or, in Handlebars parlance, in its 'block form'. Anything inside the block becomes the content of the DOM element. In this case, the band's name and the pointer will go inside the `a` tag.

Secondly, the `link-to` has a second argument, the band object. In the block form of `link-to`, positional arguments after the first one (which is the route's name we link to) become 'context objects' for generating the URL for the route. The `bands.band.songs` route needs to generate a dynamic segment, `:slug`, so we are passing in an object from which it needs to be extracted. The serialization process fetches the property defined in the dynamic path from the passed-in object. That works, since `band.get('slug')` gives us the desired `pearl-jam`, `led-zeppelin`, and so on.

On the same page where bands are listed but none are selected (in other words, on the `/bands` URL) we also want to hint at having to choose one of them to see their songs. The index route of each route allows us to do just that.

So let's create a template for `bands.index`:

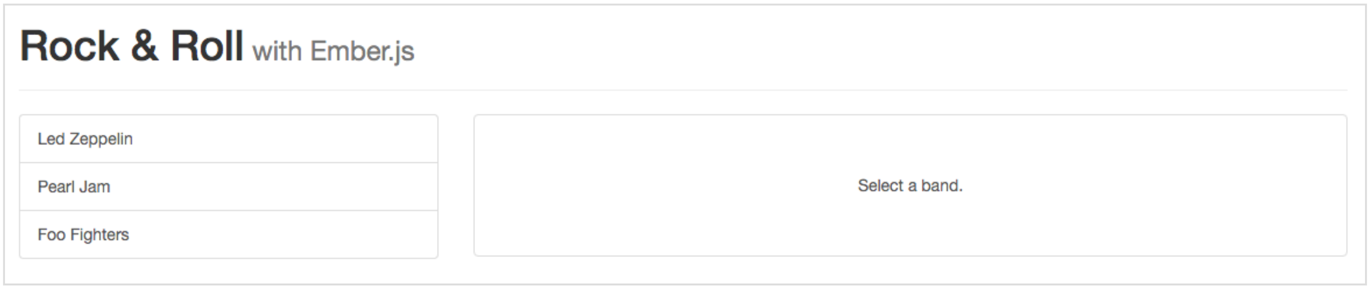
```
$ ember generate template bands/index
```

...and then place the hint in it:

```hbs
<!-- app/templates/bands/index.hbs -->
<div class="list-group">
  <div class="list-group-item empty-list">
    <div class="empty-message">
      Select a band.
    </div>
  </div>
</div>
```

The parent template `bands` creates a list of the bands in the left column (`col-md-4`) and has an `{{outlet}}` (under `col-md-8`) where content from the child routes is rendered.

The `bands.index` template, being a child of `bands`, renders a simple message that serves as the hint in that outlet:



We also want to show the songs of a band on the right when that band is clicked. The link triggers a transition to `bands.band.songs`.

We now understand the order in which the routes `bands.band` and `bands.band.songs` will be entered and their corresponding templates rendered. `bands.band` is a child of `bands`, so its template is rendered in the outlet defined by `bands`.

Finally, `bands.band.songs` is a child of `bands.band`, so its template gets rendered in the `{{outlet}}` defined by the `bands.band` template.

The template for the `bands.band` route had already been generated for us, with the right content:

```hbs
<!-- app/templates/bands/band.hbs -->
{{outlet}}
```

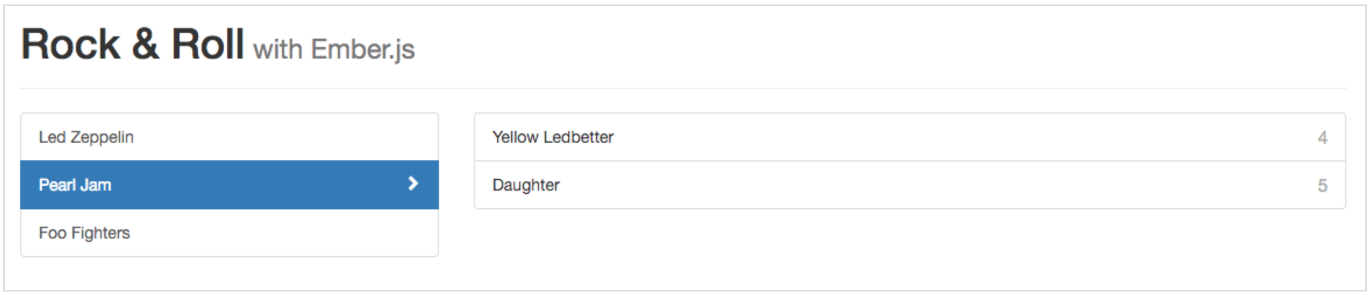
All we currently want is to display the title of each song, along with its rating, so let's create a new template to display the songs:

```
$ ember generate template bands/band/songs
```

It should have the following content:

```hbs
<!-- app/templates/bands/band/songs.hbs -->
<ul class="list-group songs">
  {{#each model.songs as |song|}}
    <li class="list-group-item song">
      {{song.title}}
      <span class="rating pull-right">{{song.rating}}</span>
    </li>
  {{/each}}
</ul>
```

Here is what we should see when we list the songs for one of the bands:



`{{outlet}}` is the default template for routes

I touted Ember in the introduction by stating that one of its principal features is “Only code that is worth writing should be written.” We can see an instance of this principle in this simple example.

There is no common markup for a particular band (yet), so the `bands.band` template just defines an `{{outlet}}` where each child route (and thus child template) can render its content.

Ember goes to great lengths to find sensible defaults; assuming a template with `{{outlet}}` as its content for a route is one such default. Consequently, the `bands.band` template could be deleted, and the app would continue working just as before. We'll not do it now as we'll modify this template in a later chapter.

The road says

Man, the simple `songs` route is no longer, yet you have the corresponding route class, template and unit test lingering. Shouldn't you just get rid of them?

```
$ rm app/routes/songs.js
$ rm app/templates/songs.hbs
$ rm tests/unit/routes/songs-test.js
```

Next song

At this point, the user can freely navigate between bands, but there is no way to meaningfully interact with the application, to create something with it. We'll fix that in the next chapter through the cunning use of actions.

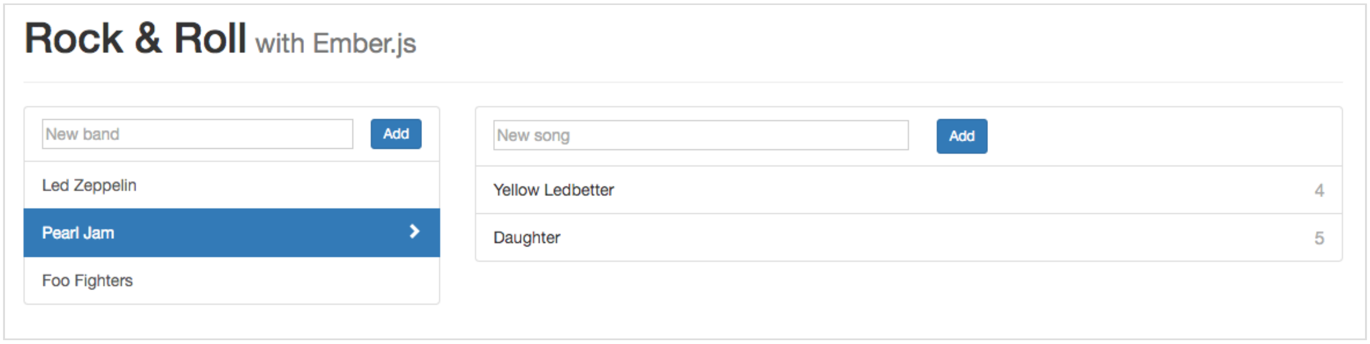
CHAPTER 6

Actions

Tuning

Event handlers transform DOM events to user intent. Clicking a link is interpreted as wanting to go to the URL indicated by the link; clicking an "Add band" button is understood as creating a band where the name is taken from a text field. The classic way that Ember provides for this is through actions.

We'll add two actions to our application so that the user can add bands and songs. Here is what our application will look like by the end of this chapter:



New styles

A couple of rules need to be added in this chapter to add some minimal styling to the inputs:

```
/* app/styles/app.css */
(...)
.new-band {
  width: 82%;
}

.new-band-button {
  float: right;
  padding: 3px 10px;
}

.new-song {
  width: 50%;
}

.new-song-button {
  margin-left: 20px;
}
```

Extracting model classes

So far we've only created a couple of bands and songs and displayed them. In this chapter, we'll also create bands and songs so now is a good time to give them their proper modules.

First, let's just move what we already have in `app/routes/bands.js`, in their respective model files, `app/models/band.js` and `app/models/song.js`, creating these files manually this time:

```js
// app/models/band.js
import EmberObject, { computed } from '@ember/object';

export default EmberObject.extend({
  name: '',

  slug: computed('name', function() {
    return this.get('name').dasherize();
  })
});
```

JS

```js
// app/models/song.js
import EmberObject from '@ember/object';

export default EmberObject.extend({
  title: '',
  rating: 0,
  band: null
});

```

JS

(I replaced the `var Band =` and `var Song =` with `export default` since we'll use these model classes as modules.)

When creating a song, we'll link back to the band by assigning the band object itself to the song's `band` property, so let's hint at this by turning the default value of the property into `null`, instead of an empty string:

```
// app/models/song.js
import EmberObject from '@ember/object';

export default EmberObject.extend({
  title: '',
  rating: 0,
- band: '',
  band: null
});
```

When creating a band, we would like it to be created with an empty array to hold its songs, without our having to explicitly define the array. We want to be able to write this:

```
Band.create({ name: 'Them Crooked Vultures' });
```

instead of this:

```
Band.create({ name: 'Them Crooked Vultures', songs: [] });
```

To achieve this goal, the `songs` property for each band needs to be set up at instantiation time:

```js
// app/models/band.js
import EmberObject, { computed } from '@ember/object';

export default EmberObject.extend({
  name: '',

  init: function() {
    this._super(...arguments);
    if (!this.get('songs')) {
      this.set('songs', []);
    }
  },

  slug: computed('name', function() {
    return this.get('name').dasherize();
  }),
});

```

Having it set up like this makes possible two use cases. The first one, seen above, is when we create a band without needing to pass an empty array for songs. The second one is useful when we pass in an array of song objects, just like we do when seeding the bands with songs that had been created earlier.

Having extracted the `Band` and `Song` classes into their own modules, they now need to be imported (instead of being defined there) in order to create our seed data:

```diff
  // app/routes/bands.js
+ import Band from 'rarwe/models/band';
+ import Song from 'rarwe/models/song';

- var Band = EmberObject.extend({
-    name: '',
-    slug: computed('name', function() {
-      return this.get('name').dasherize();
-    }),
- });
-
- var Song = EmberObject.extend({
-    title: '',
-    rating: 0,
-    band: ''
- });

  export default Route.extend({
    model: function() {
      (...)
    }
  })
```

BACKSTAGE

THE CLASS-LEVEL MUTABLE PROPERTY

You probably wonder why songs cannot be assigned a default value of an empty array at the "class" level, like this:

```js
// app/models/band.js
export default EmberObject.extend({
  name: '',
  songs: [],

  (...)
});
```

JS

The reason is the way accessing properties on the instance works, which I explained in the Class and instance properties section of the Templates and data bindings chapter. If you don't explicitly pass in an empty `songs` array for each band object you create, they will share the underlying array defined on the class (the prototype). That can be a source of weird behavior and hard-to-debug bugs.

Let's see what defining the `songs` array at the class level would do:

```js
var pearlJam = Band.create({ name: 'Pearl Jam' });
var ledZeppelin = Band.create({ name: 'Led Zeppelin' });
var daughter = Song.create({ title: 'Daughter', rating: 5 });

pearlJam.get('songs').pushObject(daughter);
pearlJam.get('songs.firstObject.title'); // => "Daughter"
ledZeppelin.get('songs.firstObject.title') // => "Daughter"
```

JS

Even though we explicitly only added Daughter to Pearl Jam, it doubles as a Led Zeppelin song. We don't want to explicitly pass in an empty array to all created band objects and thus opt to create it at initialization time for each `Band` instance.

The following block runs at instantiation time. Inside the function, `this` refers to the freshly created instance:

```js
var Band = EmberObject.extend({
  (...)
  init: function() {
    this._super(...arguments);
    if (!this.get('songs')) {
      this.set('songs', []);
    }
  },
});
```

JS

Here, `this._super()` calls the same method, `init`, in the superclass, with the same arguments.

Initializing the `songs` array on each instance instead of defining an array on the class instance ensures we don't fall into the "class-level mutable property" trap.

Capturing the event

To keep things simple, we'll add a text field with a button above the list of bands for creating a new band, and do likewise for songs. Let's start with band creation. The `bands` template only needs to change slightly:

```hbs
  <!-- app/templates/bands.hbs -->

  <div class="col-md-4">
    <div class="list-group">
+     <div class="list-group-item">
+       {{input type="text" class="new-band" placeholder="New band" value=name}}
+       <button class="btn btn-primary btn-sm new-band-button" {{action "createBand"}}>Add</button>
+     </div>
      {{#each model as |band|}}
        (...)
      {{/each}}
    </div>
  </div>
  (...)
```

The new snippet is the `<div class="list-group-item>` that wraps the `{{input ...}}` helper and the `<button>`. `{{input}}` creates a text field whose value is bound to the `name` property of the controller. Controllers will be introduced in detail in Chapter 8. The important thing to know here is that templates use properties defined in the controller for displaying and "storing" things (like the name of the new band in this case).

A two-way binding is created between the `name` value of the controller backing the template (`bands`) and the value of the text field. If one of them changes, the other one follows this change.

Let's now turn our attention to the `{{action "createBand"}}` helper embedded in the button. It creates an event listener that acts when the button is clicked (though you can set it up to be triggered by a different event by passing its name, for example `{{action "createBand" on="doubleClick"}}`).

Turning event into intent

When the user inputs "Muse" and then clicks the button, the "createBand" action will be triggered first on the corresponding controller, then on the current route, and finally bubble up the active routes, ending in the `application` route. Not wanting to bring in controllers just yet, we'll handle the action on the `bands` route and do as the user wished by creating a new band. The name of the band comes from the controller's `name` property it was bound to:

```js
  // app/routes/bands.js
  export default Route.extend({
    model: function() {
      (...)
    },

+   actions: {
+     createBand: function() {
+       var name = this.get('controller').get('name');
+       var band = Band.create({ name: name });
+       this.modelFor('bands').pushObject(band);
+       this.get('controller').set('name', '');
+     }
+   }
  });
```

Action handlers reside in an `actions` object and the name of the method has to match the name used in the template. `this.get('controller').get('name')` accesses the controller instance belonging to the route, and then fetches the name property that we know has the value inputted by the user, thanks to the binding. We then proceed by creating a band with that name and then resetting the name. That will also clear the text field so that the user can add another band right away.

Creating songs happens in a very similar fashion. Let's first add the text field and the button with the action helper, just as we did with bands:

```hbs
<!-- app/templates/bands/band/songs.hbs -->
<ul class="list-group songs">
+  <li class="list-group-item">
+    {{input type="text" class="new-song" placeholder="New song" value=title}}
+    <button class="btn btn-primary btn-sm new-song-button" {{action "createSong"}}>Add</button>
+  </li>
  {{#each model.songs as |song|}}
    (...)
  {{/each}}
</ul>
```

We now know the triggered action is called `createSong` and that we can handle it on the current route, `bands.band.songs`:

```
// app/routes/bands/band/songs.js
import Song from 'rarwe/models/song';

export default Route.extend({
  model: function() {
    return this.modelFor('bands.band');
  },

  actions: {
    createSong: function() {
      var controller = this.get('controller');
      var band = this.modelFor('bands.band');
      var title = controller.get('title');

      var song = Song.create({ title: title, band: band });
      band.get('songs').pushObject(song);
      controller.set('title', '');
    }
  }
});
```

JS

With that, we have made our first step toward interacting with the application instead of just observing it.

Next song

It's great that we can add new bands and songs, but it would even be nicer to be able to rate the songs in our collection. Taking our cue from the star rating widget in iTunes, we'll add that next and have it update the song's rating using the concepts of actions we have just learned.

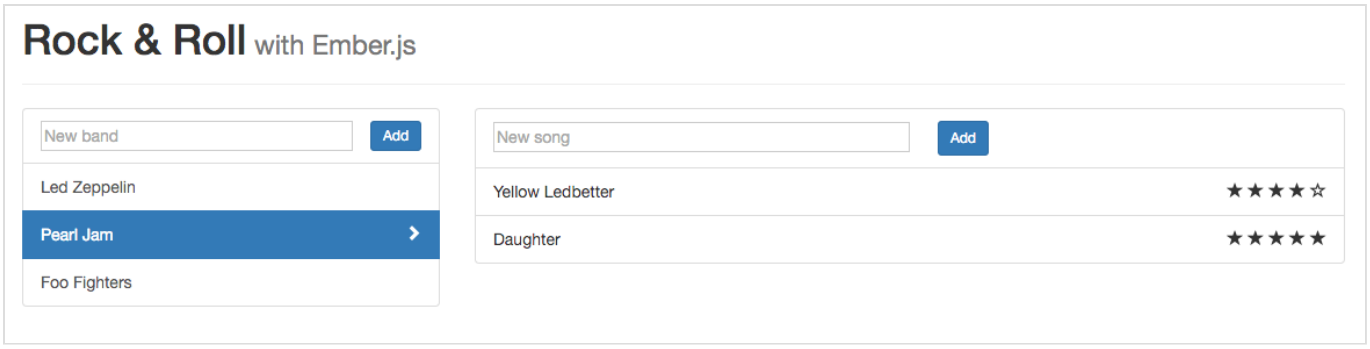
CHAPTER 7

Components

Tuning

HTML only goes so far. More often than not, we find ourselves wanting to extend it to support a richer UI or enable more sophisticated event handling. Ember makes this possible by allowing us to create components. The current task, creating a star-rating widget, is a perfect example of when to reach for them.

Here is what our application should look like by the end of this chapter:



The idea behind components

Ember components are designed to be reusable not just within the same application, but potentially between different projects, too. As a consequence, a component can only access whatever is passed into it at creation time. It knows nothing about the context and can only communicate with the "outer world" by sending actions to it. The resulting isolation fosters reusability.

Component specification

Let's sketch out a spec for our star-rating component, keeping in mind that it should be reusable in other applications, too. Since each situation where a star-rating widget is needed differs, the component needs to be configurable. Configuration happens by passing in key-value pairs of property names and their values, and also by using the block-form of the component (more about that later).

Such a component should know how many stars to draw in total, and how many of these should be rendered as a full star. Since clicking on one of these stars should trigger an action, the name of that action must be customizable, too. With these specifications nailed down, let's get to work.

Implementation

New styles

Before we dive into making the star-rating component, add the following rules to the app's stylesheet, so that the stars will be rendered correctly:

```css
/* app/styles/app.css */

(...)
.rating-panel {
  float: right;
}

a.star-rating {
  color: inherit;
  text-decoration: none;
}
```

Defining the component's properties

The first step is to use an Ember CLI generator to create a new component:

```
$ ember generate component star-rating
```

That creates two files under `app`. One of them contains the component's code in `app/components/star-rating.js`, the other its template under `app/templates/components/star-rating.hbs`.

Components extend the `Component` class defined in the `@ember/component` module. In the class definition, the component should define its properties, CSS classes, and attributes.

Let's modify the generated code of the component according to our specifications:

```
// app/components/star-rating.js
import Component from '@ember/component';

export default Component.extend({
+  tagName: 'div',
+  classNames: ['rating-panel'],
+
+  rating:     0,
+  maxRating:  5
});
```

JS

The `tagName` specifies what HTML tag should be rendered for the component. The classes defined in `classNames` will become CSS classes when rendering the component. `rating` and `maxRating` are custom properties that are not rendered in the HTML representation of the component, at least not directly. `rating` is the actual rating (score) of the item (song, in our example) while `maxRating` is the maximum rating the item can be given.

`classNames` is a concatenated property. You can read more about them in the Encore.

Using the component

Displaying the rating with stars

Most of the time, you will create components from within templates and not from JavaScript code as we saw above. It works very similarly, though, in templates.

Let's replace the simple display of each song's rating with our star-rating widget:

```hbs
  <!-- app/templates/bands/band/songs.hbs -->
  <ul class="list-group songs">
    (...)
    {{#each model.songs as |song|}}
      <li class="list-group-item song">
        {{song.title}}
-       <span class="rating pull-right">{{song.rating}}</span>
+       {{star-rating rating=song.rating}}
      </li>
    {{/each}}
  </ul>
```

We create a star-rating widget for each song in the list by writing `star-rating` and assigning values to its properties. The ones we do not define will take their value from the "class" definition of the component. `maxRating`, for example, will be equal to 5.

By convention, a `star-rating` used in a template will be looked up as `component:star-rating`. Having the code of the component in `app/components/star-rating.js` creates a module that will be found by that name, so no further wiring up is necessary. Component names must have a dash in their name to prevent possible future clashes with HTML tag names.

It is also important to know that passed-in values that are unquoted will establish a binding between the property from the context and the property of the component. Above, the `rating` of the component is assigned `song.rating` (which is the rating of the song) and a two-way binding is created between the two.

Should either end change, the other will change accordingly. Let's leverage that to display the correct number of full stars when the rating of the song changes:

```hbs
<!-- app/templates/components/star-rating.hbs -->
{{#each stars as |star|}}
  <a href="#" {{action "setRating" star.rating}}
     class="star-rating glyphicon {{if star.full 'glyphicon-star'
'glyphicon-star-empty'}}">
  </a>
{{/each}}
```

HBS

You can see that we use the `stars` property in the template to render each star. Each star has a `rating` and a `full` property. `rating` is passed to the action handler when the star is clicked. The `full` property is used in the inline `if` helper. If the condition is truthy, the first value is returned, otherwise the second one. So each star will have the `star-rating` and `glyphicon` classes and either the `glyphicon-star` or the `glyphicon-star-empty` class, depending on whether `star.full` is true or false.

Now, let's see how the `stars` property that the template iterates on is implemented:

```js
// app/components/star-rating.js
import { computed } from '@ember/object';
import Component from '@ember/component';

export default Component.extend({
  tagName: 'div',
  classNames: ['rating-panel'],

  rating:     0,
  maxRating:  5,

  stars: computed('rating', 'maxRating', function() {
    var fullStars = this.starRange(1, this.get('rating'), 'full');
    var emptyStars = this.starRange(this.get('rating') + 1,
this.get('maxRating'), 'empty');
    return fullStars.concat(emptyStars);
  }),

  starRange: function(start, end, type) {
    var starsData = [];
    for (var i = start; i <= end; i++) {
      starsData.push({ rating: i, full: type === 'full' });
    }
    return starsData;
  }
});
```

`stars` first calls the `starRange` function to receive as many full stars as the `rating`. Then, it calls it again to get as many empty stars as the difference is between `maxRating` and `rating`. The full and empty stars will then be joined together, with the full stars in front.

In Ember templates, only properties can be rendered; results of function calls can not (If you'd like to learn more about why that is so, read the "On the utility of explicit dependency definitions" section in the Encore. That means `stars` need to be turned into a (computed) property by calling `computed`, specifying its dependent keys, `rating` and `maxRating` and the function that returns the value of the property. If you need a refresher on computed properties, go back to the Nested routes chapter.

Speaking to the outside world through actions

We now have a fully functioning component that can render itself based on the data that was passed in. It is in splendid isolation, but what it lacks is the ability to communicate with the outside world. So let's round out this chapter by making our component more sociable.

Clicking on one of the stars expresses the user's intent to update the rating of the song. We know from the Actions chapter that the `{{action "createBand"}}` in a route's template will trigger an action called "createBand" when the element is clicked.

An action launched from inside a component acts a bit differently: it looks up that action on the component itself, instead of the "controller, then current route, then active routes" path we saw in the previous chapter.

Components define their own action handlers the same way as we saw for routes, in a top-level `actions` object. Let's set up the action for updating the song's rating, then:

```js
// app/components/star-rating.js
export default Component.extend({
  tagName: 'div',
  classNames: ['rating-panel'],

  rating:     0,
  maxRating:  5,
+ item:       null,
+ onClick:   '',
  (...)
});
```

The item whose rating we want to set and the name of the action we want to trigger also need to be passed in:

```hbs
  <!-- app/templates/bands/band/songs.hbs -->
  <ul class="list-group songs">
    (...)
    {{#each model.songs as |song|}}
      <li class="list-group-item song">
        {{song.title}}
-       {{star-rating rating=song.rating}}
+       {{star-rating rating=song.rating item=song onClick="updateRating"}}
      </li>
    {{/each}}
  </ul>
```

Since the item whose rating can be set via the stars is also passed in to the component, we can go ahead and grab it, and then set its new rating:

```js
  // app/components/star-rating.js
  export default Component.extend({
    (...)
+   actions: {
+     setRating: function(newRating) {
+       this.get('item').set('rating', newRating);
+     }
+   }
  });
```

This works, but doesn't this code dampen the component's reusability? Any time you suspect you are doing something that is specific to the current context the component is used in, stop and assess whether you are relying on the peculiarities of the current case. Here, we *are* doing that, since we use the fact that the song has a rating property. This would prevent the component from being used in another scenario where the item has a differently-named property, like "score" or "points."

Furthermore, handling the updating of the song's rating is best left to the application. In some cases clicking on the star that represents the current rating might be interpreted as wanting to set the rating to zero. In other situations, we might do nothing in that case, leaving the current rating intact.

The way to make the component truly reusable is to gather all data inherent to the component and send an action to the context passing along these data. This is what the assigned `onClick` is for:

```diff
  // app/components/star-rating.js
  export default Component.extend({
    (...)
    actions: {
      setRating: function(newRating) {
-       this.get('item').set('rating', newRating);
+       this.sendAction('onClick', {
+         item: this.get('item'),
+         rating: newRating
+       });
      }
    }
  });
```

JS

`sendAction('onClick', ...)` sends the action name passed in to the component (`updateRating` in the above case) with some optional parameters. That action will be handled by the first handler on the "controller, then current route, then active routes" path. In our case, the `bands.band.songs` route will have the privilege to do so:

```js
// app/routes/bands/band/songs.js
export default Route.extend({
  (...),
  actions: {
    createSong: function() {
      (...)
    },
    updateRating: function(params) {
      var song = params.item,
          rating = params.rating;

      song.set('rating', rating);
    }
  }
});
```

As the song's rating is updated, the binding will trigger an update of the component's `stars` property, and so the component's template will get re-rendered with the appropriate number of full stars.

It is also worth noting that you can call the attribute which stores the action name anything you want to. I called it `onClick`, but you could call it `foo` , `doThisWhenClicked` or anything else. Just don't forget to pass the name you picked when you invoke `this.sendAction`.

Closure actions

From Ember 1.13 onward, there is another way to trigger actions. Instead of passing in an action name (a string) to the component, a function, returned by a call to the `action` helper can be passed in.

Updating a song's rating is currently done through an "old-style" action:

```hbs
<!-- app/templates/bands/band/songs.hbs -->
<ul class="list-group songs">
  (...)
  {{#each model.songs as |song|}}
    <li class="list-group-item song">
      {{song.title}}
      {{star-rating item=song rating=song.rating onClick="updateRating"}}
    </li>
  {{/each}}
</ul>
```

To convert this to a closure action, we need to use the `action` helper in our template:

```hbs
  <!-- app/templates/bands/band/songs.hbs -->
  <ul class="list-group songs">
    (...)
    {{#each model.songs as |song|}}
      <li class="list-group-item">
        {{song.title}}
-       {{star-rating item=song rating=song.rating onClick="updateRating"}}
+       {{star-rating item=song rating=song.rating on-click=(action "updateRating")}}
      </li>
    {{/each}}
  </ul>
```

```
// app/components/star-rating.js
export default Component.extend({
  (...)
- onClick: '',
+ "on-click": null,
  (...)
});
```

(In JavaScript, property names that are not valid identifiers need to be quoted. Here it is the dash that makes this necessary.)

The `action` helper looks up the provided name in the actions object of the current context, in this case the controller and creates an action function from it encapsulating the current context (hence the name, "closure action").

Since we had defined the `updateRating` action on the route, we need to move it to the controller. Controllers will be covered in the next chapter in detail, for now, let's just create one to contain that single action:

```
$ ember generate controller bands/band/songs
```

```
// app/controllers/bands/band/songs.js
export default Controller.extend({
+  actions: {
+    updateRating: function(params) {
+      var song = params.item,
+          rating = params.rating;
+
+      song.set('rating', rating);
+    }
+  }
});
```

In the same step, let's remove the action from the route where it's no longer necessary:

```js
  // app/routes/bands/band/songs.js
  export default Route.extend({
    (...)
    actions: {
      createSong: function() {
        (...)
      },
-     updateRating: function(params) {
-       var song = params.item,
-           rating = params.rating;
-
-       song.set('rating', rating);
-     }
    }
  });
```

Instead of storing an action name (a string) to trigger in the `on-click` attribute, we now have a function to be called . The component's `setRating` action handler needs to reflect that change.

```
  // app/components/star-rating.js
  import Component from '@ember/component';

  export default Component.extend({
    (...)
    actions: {
      setRating: function(newRating) {
-       this.sendAction('onClick', {
+       this.get('on-click')({
          item: this.get('item'),
          rating: newRating
        });
      }
    }
  });
```

Closure actions are clearly an improvement over old-style, bubbling actions. They are easier to reason about and debug because if the named action does not exist on the template's context (probably because the name was mistyped), we'll get a clear error message:

```
Uncaught Error: An action named 'updateRaying' was not found in
<rarwe@controller:bands/band/songs::ember621>.
```

Another advantage of closure actions just being functions is that they have a return value that we can use at the point where we handle (call) the action. That may be leveraged to see if the upstream action was successful or to pass back additional information, something that was not possible with bubbling actions.

The road says

Let's get rid of the generated integration test for our component before it wreaks havoc, 'mkay? I know for a fact that we'll set up a decent integration test for in the Testing chapter, anyways.

```
$ rm tests/integration/components/star-rating-test.js
```

Oh, and please do the same for the controller:

```
$ rm tests/unit/controllers/bands/band/songs-test.js
```

Next song

The app not only has a way to create bands and songs, but also a flashy widget to rate those songs. Missing are a few simple things to make the app more user friendly, like disabling the Add buttons while nothing is typed as the name of the new band or song. Enter controllers.

CHAPTER 8

Controllers

Tuning

We can navigate through the app, create new bands and songs, and rate them via a star rating widget we made. Let's kick our app up a notch by adding a few things to improve the user experience. The building blocks that we will use to achieve this are controllers.

Preventing the creation of bands without a name

Currently, one can create a new band with an empty string as its name by clicking on the "Add" button without typing anything in. To prevent that, we could disable the button as long as the text field is empty. We need to bind the button's disabled property to a property that tracks the emptiness of the field.

Templates can access the properties defined on the corresponding controller, so we need to create a controller for the `bands` route (and thus template):

```
$ ember generate controller bands
```

We then define a `isAddButtonDisabled` property there that will control the `disabled` property of the text input field:

```js
// app/controllers/bands.js
import { isEmpty } from '@ember/utils';
import { computed } from '@ember/object';
import Controller from '@ember/controller';

export default Controller.extend({
+  name: '',
+
+  isAddButtonDisabled: computed('name', function() {
+    return isEmpty(this.get('name'));
+  })
});
```

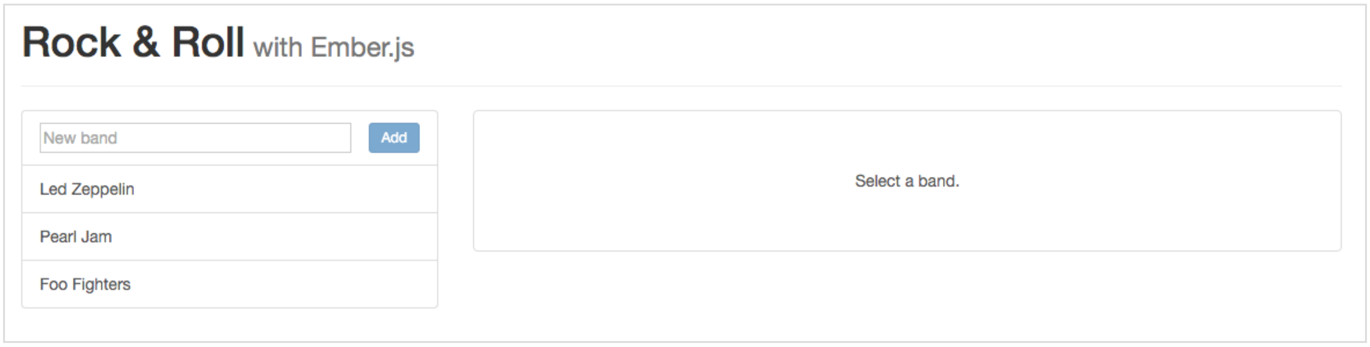
JS

Remember, the `name` property is what the value of the text input is bound to in the template.

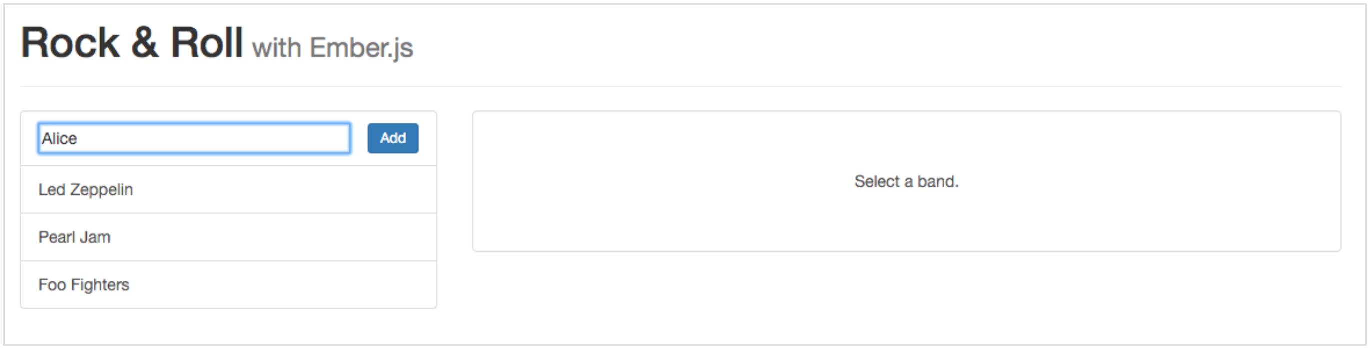
Then, we set up the templates so that the Add button is disabled when the `isAddButtonDisabled` property returns true:

```hbs
<!-- app/templates/bands.hbs -->
<div class="col-md-4">
  <div class="list-group">
    <div class="list-group-item">
      {{input type="text" class="new-band" placeholder="New band" value=name}}
-      <button class="btn btn-primary btn-sm new-band-button" {{action "createBand"}}>
+      <button class="btn btn-primary btn-sm new-band-button" disabled={{isAddButtonDisabled}} {{action "createBand"}}>
        Add
      </button>
    </div>
    {{#each model as |band|}}
      {{#link-to "bands.band.songs" band class="list-group-item band-link"}}
        {{band.name}}
        <span class="pointer glyphicon glyphicon-chevron-right"></span>
      {{/link-to}}
    {{/each}}
  </div>
</div>
(...)
```

When you now go to the `bands` route, you can see that the "Add" button is disabled by default:



As you start to type the name of the band, though, the text field comes to life:



Since the value and disabled attribute of the text field are bound to the same property, `name`, the text field functions as a "self-enabling" input. When something is typed, the value of the field and thus the `name` property of the controller changes, which also makes the `isAddButtonDisabled` property be recomputed and become false.

Defining the `isAddButtonDisabled` property for the text field to create a song happens exactly the same way, and is left as an exercise to the reader (just make sure to use `title` instead of `name`).

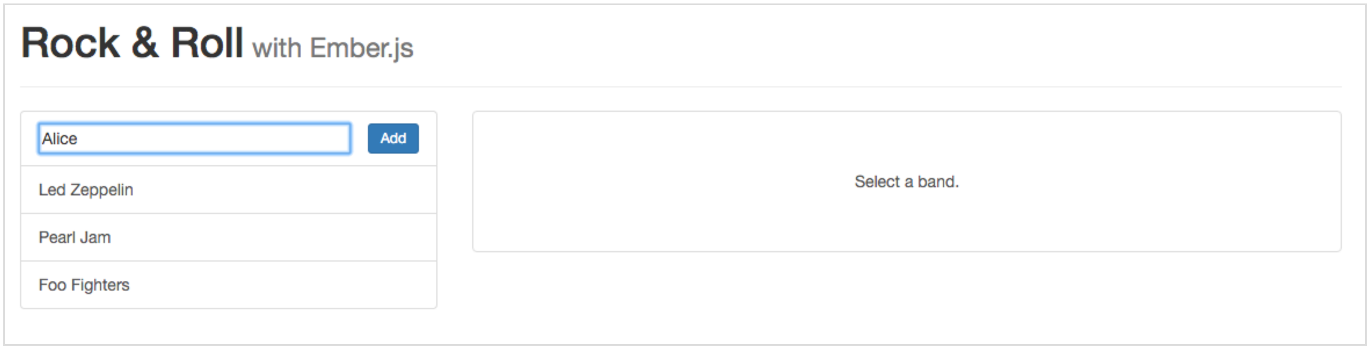
Smooth UI flows

If we think through some typical UI flows, we realize most of the time the creation of a band is going to be instantly followed by adding songs to that band. So if we want to improve user experience we should make that flow smoother by guiding the user through the process.

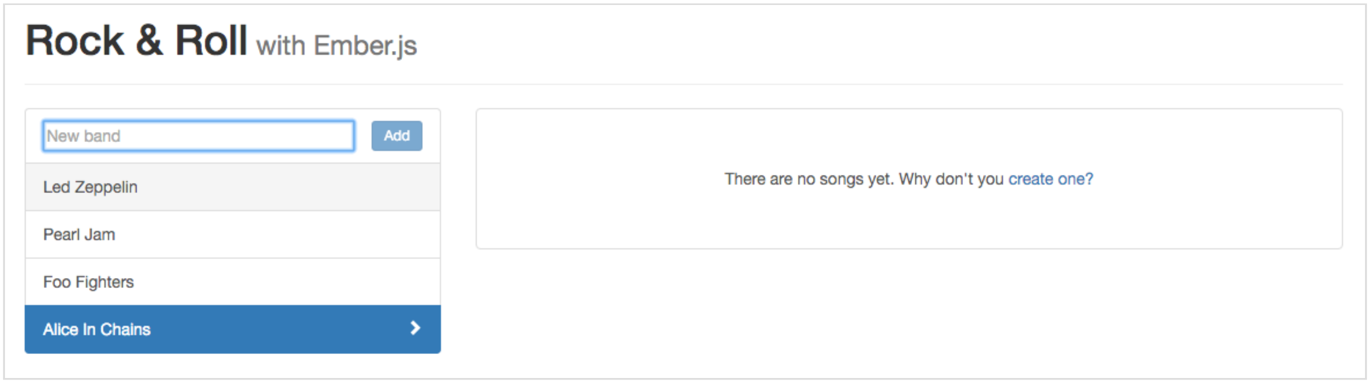
Here is how we could achieve this:

- When a band is created, let's go to the song listing for that band (which will be empty)
- If the list of songs is empty, let's add a blurb with a call to action that nudges the user to create the first song.
- When the user acts on that call by clicking the link, let's display the text field to create the first song.

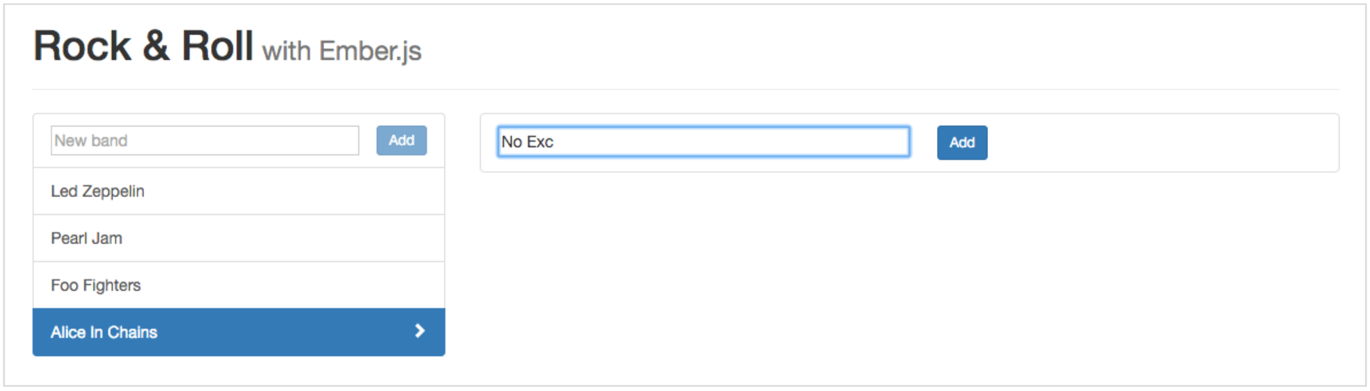
Here is what the process would look like:

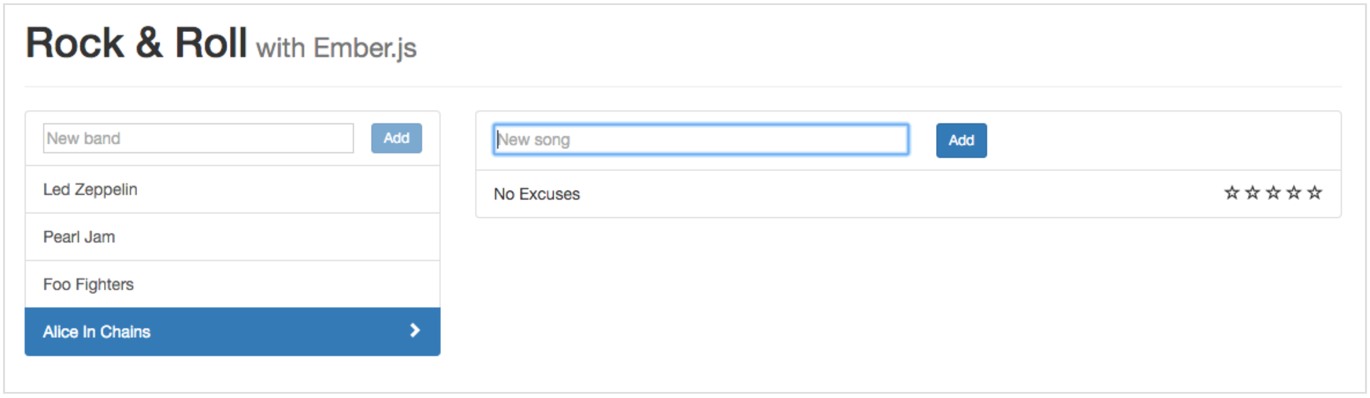


Select a band.



There are no songs yet. Why don't you create one?





Let's see in detail how each of these steps is implemented.

1. Going to the band page after it is created

We know how to navigate between routes in templates using the `link-to` helper. However, this time, we need a way to navigate to another route from inside an action handler of a route:

```js
// app/routes/bands.js
export default Route.extend({
  (...),
  actions: {
    createBand: function() {
      var name = this.get('controller').get('name');
      var band = Band.create({ name: name });
      this.modelFor('bands').pushObject(band);
      this.get('controller').set('name', '');
+     this.transitionTo('bands.band.songs', band);
    }
  }
});
```

We have seen the above snippet before; the only new thing is the `this.transitionTo` line at the end. It starts a transition to the route passed in as the first parameter (`bands.band.songs`), optionally taking extra parameters for the dynamic segments for the target route. In this case, we pass the new band object.

2. Nudging the user to create the first song

Now that the user sees the empty song listing, we want to provide a guided way to create the first song. We only want to show the text field for the song title if the user has performed the action we nudge him to do (see the second screenshot above).

First, let's display the blurb with the call to action when there are no songs:

```
  <!-- app/templates/bands/band/songs.hbs -->
  <ul class="list-group songs">
    (...)
+   {{#if noSongs}}
+     <li class="list-group-item empty-list">
+       <div class="empty-message">
+         There are no songs yet. Why don't you <a href="#" {{action "enableSongCreation"}}>create one?</a>
+       </div>
+     </li>
+   {{/if}}
  </ul>
```

`noSongs` needs to be defined on the controller for the route. We had already created the controller in the previous chapter so let's just add the new property:

```
  // app/controllers/bands/band/songs.js
  export default Controller.extend({
+   noSongs: computed('model.songs.[]', function() {
+     return this.get('model.songs.length') === 0;
+   }),
    (...)
  });
```

Devised this way, our blurb is going to show up only when there are no songs. Once the user creates a song, the `model.songs.length` property changes to 1, `noSongs` gets recomputed (since its dependent key has changed) and becomes false, and the blurb disappears.

The `object.array.[]` syntax for the dependent key invalidates the value of the CP if any of the following happens:

- `this.get('object')` is assigned a new value
- `this.get('object.array')` is assigned a new value
- An item in `this.get('object.array')` is replaced

- An item in `this.get('object.array')` is deleted
- A new item is added to `this.get('object.array')`

3. Showing the text field when the call to action has been acted on

The final piece of the puzzle is to show the text field for song creation only when the user clicks on the "create one" link. We know how to do that by firing an action and setting the necessary properties in an action handler. Here, we should flip a switch that will make the text field appear.

We call that switch `songCreationStarted`:

```hbs
<!-- app/templates/bands/band/songs.hbs -->
<ul class="list-group songs">
  {{#if songCreationStarted}}
    <li class="list-group-item">
      {{input type="text" class="new-song" placeholder="New song" value=title}}
      <button class="btn btn-primary btn-sm new-song-button"
              disabled={{isAddButtonDisabled}}
              {{action "createSong"}}>Add</button>
    </li>
  {{/if}}
  {{#each model.songs as |song|}}
    (...)
  {{/each}}
  {{#if noSongs}}
    <li class="list-group-item empty-list">
      <div class="empty-message">
        There are no songs yet. Why don't you <a href="#" {{action "enableSongCreation"}}>create one?</a>
      </div>
    </li>
  {{/if}}
</ul>
```

We define the property on the controller:

```
// app/controllers/bands/band/songs.js
export default Controller.extend({
+   songCreationStarted: false,
    (...)
});
```

Clicking on "create one" sends the "enableSongCreation" action, so that is where we should flip that switch. We choose to handle the action "locally", on the controller itself:

```
// app/controllers/bands/band/songs.js
export default Controller.extend({
  songCreationStarted: false,
  (...)
  actions: {
+   enableSongCreation: function() {
+     this.set('songCreationStarted', true);
+   },
    (...)
  }
});
```

At first it seems that everything works correctly, we can create a new band, click the link to start the song creation "process" and then create the first song for the band.

However, when we start with a band that already has some songs, we find the "New song" text field gone. That is because we bound the showing of the text field to the `songCreationStarted` property, which is false.

We need to refine our conditionals to show the text field *either* if a band already has songs *or* if the song creation process is currently under way:

```js
  // app/controllers/bands/band/songs.js
  export default Controller.extend({
    songCreationStarted: false,

    (...)
+   canCreateSong: computed('songCreationStarted', 'model.songs.[]', function() {
+     return this.get('songCreationStarted') || this.get('model.songs.length');
+   }),
    (...)
  });
```

We can then use the `canCreateSong` property in the template instead of `songCreationStarted`:

```hbs
  <!-- app/templates/bands/band/songs.hbs -->
  <ul class="list-group songs">
-   {{#if songCreationStarted}}
+   {{#if canCreateSong}}
      <li class="list-group-item">
        {{input type="text" class="new-song" placeholder="New song" value=title}}
        <button class="btn btn-primary btn-sm new-song-button"
                {{action "createSong"}}
                disabled={{isAddButtonDisabled}}>Add</button>
      </li>
    {{/if}}
    (...)
  </ul>
  (...)
```

We should also note that the "There are no songs yet. Why don't you create one?" message should probably be hidden when the user has clicked on it and also when there are already songs for that band. This is precisely the inverse of `canCreateSong`. We can thus "reuse" this property using the `unless` Handlebars helper which is the opposite of `if`. It evaluates to true when the condition passed to it is false:

```
  <!-- app/templates/bands/band/songs.hbs -->
  <ul class="list-group songs">
    (...)
-   {{#if noSongs}}
+   {{#unless canCreateSong}}
      <li class="list-group-item empty-list">
        <div class="empty-message">
          There are no songs yet. Why don't you <a href="#" {{action "enableSongCreation"}}>create one?</a>
        </div>
      </li>
-   {{/if}}
+   {{/unless}}
  </ul>
```

HBS

The `canCreateSong` property now also fulfills what the `noSongs` CP has achieved earlier and thus the latter can be safely removed:

```
  // app/controllers/bands/band/songs.js
  export default Controller.extend({
    songCreationStarted: false,
-   noSongs: computed('model.songs.[]', function() {
-     return this.get('model.songs.length') === 0;
-   }),
    (...)
  });
```

JS

BACKSTAGE

WHERE TO HANDLE ACTIONS?

We have now seen action handlers defined on routes, components, and controllers.

In the case of element actions (as opposed to closure actions) we can opt to handle the action on the controller, the current route, or any of the active routes.

A good rule of thumb is that any action which only modifies the state of the controller itself be handled on the controller. Actions that have a broader- reaching effect should be handled by routes.

Toggling a controller property, like `songCreationStarted` above, is an example of controller action handling. Creating bands and songs, since their effects are "global", are examples of doing that in the routes.

Ember also allows for the same event instance to be handled multiple times. If a handler returns `true`, the action will bubble up, giving a chance for other handlers to catch it and act accordingly. Using this, you might handle a certain action in the controller (modifying one of its properties) and then let the action propagate up the route tree where any route in the path can also handle it (creating a model object, for example).

Closure actions are much more clear-cut. Since the action helper that creates a closure action throws an error if the action is not found in the current context, they need to be defined there. This means the controller belonging to the route's template in simple cases. As closure actions are more often than not preferable to element actions (see the Components chapter for reasons), this tips the balance in favor of "local" action handling.

Keep in mind that the above is a guideline, not a law. The specifics of the particular case you want to solve might very well warrant deviating from this guideline.

Creating new bands and songs by pressing return

To wrap up this chapter, let's make another quick improvement to user experience. To create a band or song, the user currently has to click the Add button. It would be great to be able to do that from the keyboard with the return key.

Submitting the form is the canonical way to achieve this. If any text field on the form has the focus and the return key is pressed, it sends the submit event on the form.

Consequently, we need to wrap the text input into a form and send our `createBand` action when the form is submitted:

```hbs
  <!-- app/templates/bands.hbs -->
  <div class="col-md-4">
    <div class="list-group">
      <div class="list-group-item">
+       <form {{action "createBand" on="submit"}}>
          {{input type="text" class="new-band" placeholder="New band" value=name}}
-         <button class="btn btn-primary btn-sm new-band-button" disabled={{isAddButtonDisabled}} {{action "createBand"}}>
+         <button type="submit" class="btn btn-primary btn-sm new-band-button" disabled={{isAddButtonDisabled}}>Add</button>
+       </form>
      </div>
      {{#each model as |band|}}
        {{#link-to "bands.band.songs" band class="list-group-item band-link"}}
          {{band.name}}
          <span class="pointer glyphicon glyphicon-chevron-right"></span>
        {{/link-to}}
      {{/each}}
    </div>
  </div>
  <div class="col-md-8">
    {{outlet}}
  </div>
```

I attached a `submit` event handler onto the form by specifying `on="submit"` to the action helper. If the `on` option is not defined, a click handler will be added, which was sufficient up until this point. By making the button of type "submit," I could also remove the action handler, since the button will trigger the form's submit event when clicked.

Doing the same for adding a new song follows exactly the same pattern:

```
  <!-- app/templates/bands/band/songs.hbs -->
  <ul class="list-group songs">
    {{#if canCreateSong}}
      <li class="list-group-item">
+       <form class="new-song-form" {{action "createSong" on="submit"}}>
          {{input type="text" class="new-song" placeholder="New song" value=title}}
-         <button class="btn btn-primary btn-sm new-song-button" disabled={{isAddButtonDisabled}} {{action "createSong"}}>Add</button>
+         <button type="submit" class="btn btn-primary btn-sm new-song-button" disabled={{isAddButtonDisabled}}>Add</button>
        </form>
      </li>
    {{/if}}
    (...)
  </ul>
  (...)
```

The road says

Hey, another lingering controller test. What'cha gonna do about it?

```
$ rm tests/unit/controllers/bands-test.js
```

Next song

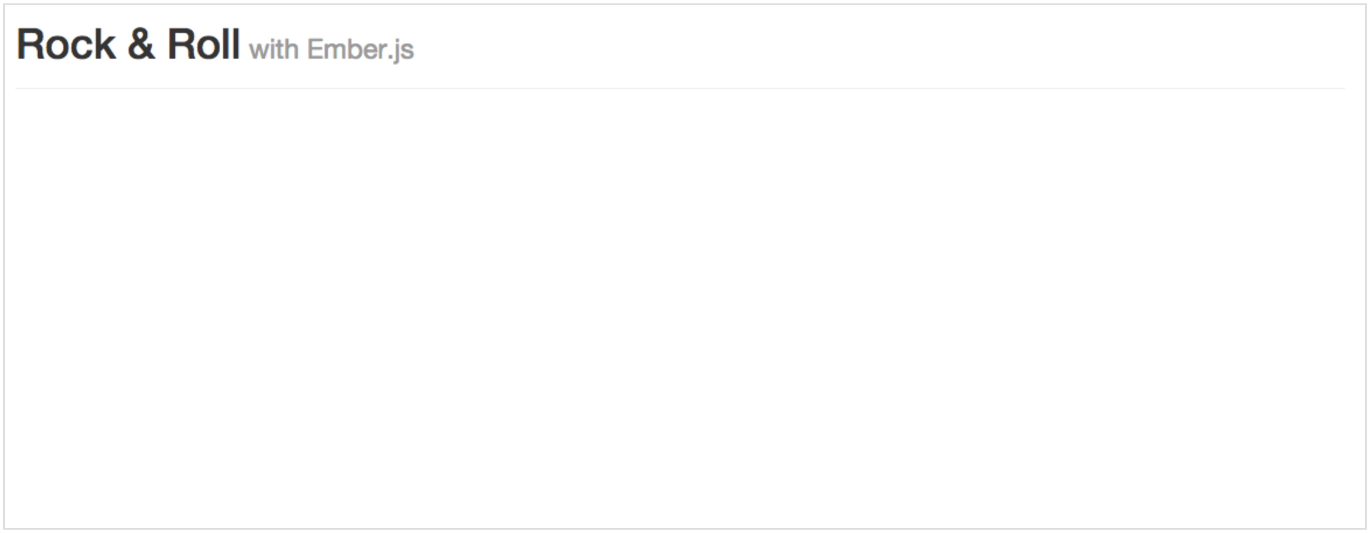
Believe it or not, we've only scratched the surface of what routes can accomplish in Ember. In the next chapter, we will take a more thorough look at route callback functions, "hooks", and make our application rock even harder!

CHAPTER 9

Advanced routing

Tuning

When we visit the root of our application, nothing is shown except the header:



One way to remedy that is to move our routes "one level up", so that our `bands` route becomes the root (the `index`) route and the list of bands is shown. Another possibility is to have the top-level `index` route redirect to the `bands` route.

Not only does that keep our routing structure intact, but it gives us the flexibility of showing another view there in the future, such as a dashboard that shows the latest and most popular bands and songs. To implement the redirection, we'll have to dive deeper into router transitions and in-route hooks specifically.

Route hooks

We saw in the Nested routes chapter how the router traverses each route level when transitioning to the destination route. It was a simplified model, since it only mentioned the `model` hook being called for each route. The truth is somewhat more complex: it's not only the `model` hook but several model hooks that are called in succession.

The first one is `beforeModel`. Since it gets called before the resolved model is known, the only parameter it receives is the current transition:

```js
beforeModel: function(transition) {...}
```

Next up is the familiar `model` hook. In addition to the transition, it also receives the params from the URL, which are parsed according to the routing table:

```js
model: function(params, transition) {...}
```

Once the model is resolved, the `afterModel` hook is called with the model and the transition object:

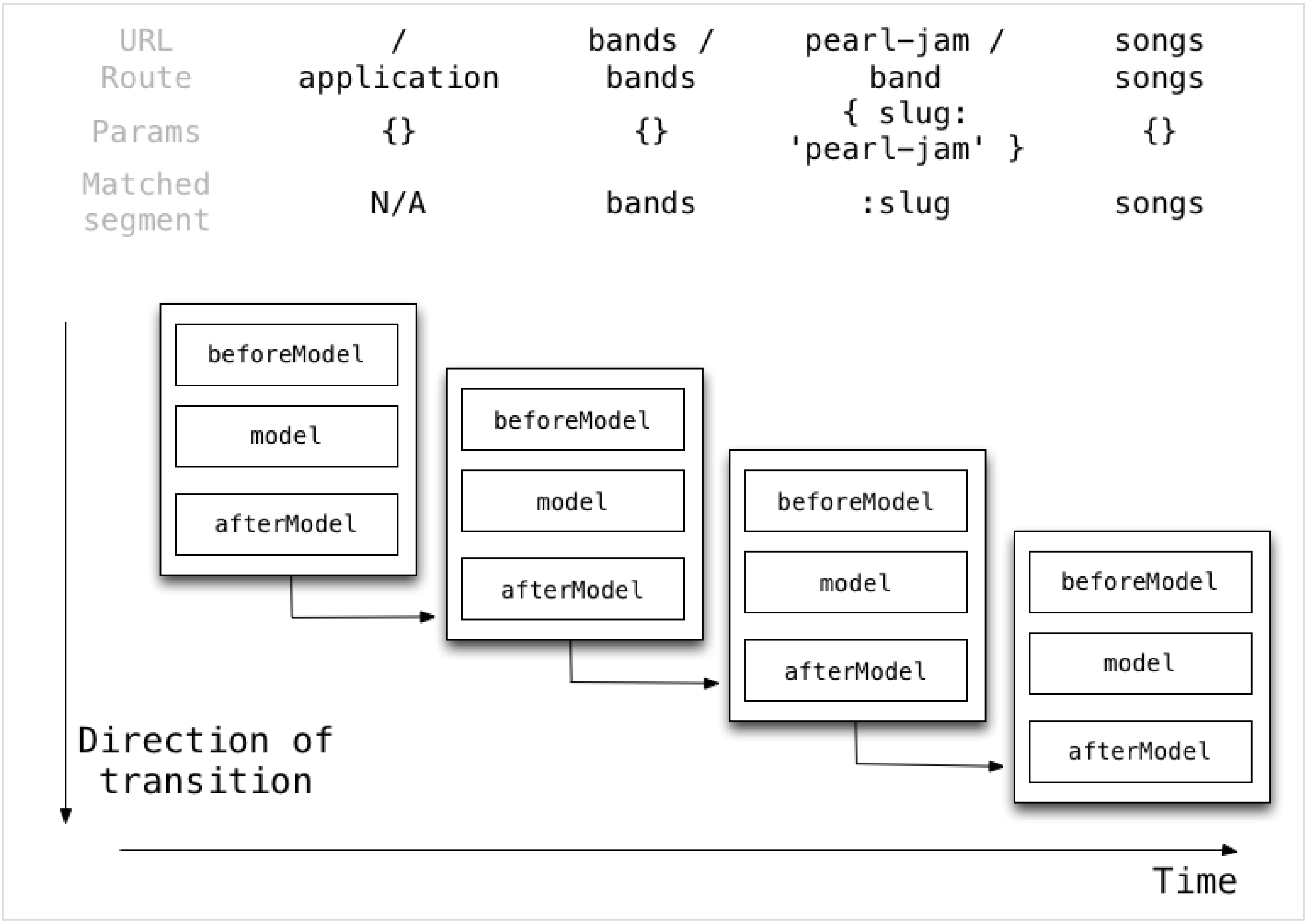
```js
afterModel: function(model, transition) {...}
```

So at each level of a route transition, the beforeModel-model-afterModel hooks are called for the particular route (with one exception that we are going to see later). Only when these have all run for all levels of the transition is the next hook, called `setupController`, entered:

```js
setupController: function(controller, model, transition) {...}
```

As its title suggests, its main job is to set up the controller belonging to the route. The default implementation sets the controller's `model` property to the model that was previously resolved - if this is the desired behavior, the `setupController` hook does not need to be provided.

The following graph helps to visualize how the router calls each model hook for each route:



Once these hooks are called and the transition is complete, the `setupController` hook is called for each route, starting from the top-most route, `application`, down to `bands.band.songs`.

Route actions

Above the route hooks, there also exist "route actions" that are triggered at the beginning and the end of a transition. They are called `willTransition` and `didTransition`. Since they are actions, they need to be defined inside the `actions` object of the route:

```js
actions: {
  willTransition: function(transition) {...},
  didTransition: function() {...}
}
```

As their names suggest, `willTransition` is triggered on the source route before the transition starts and `didTransition` is triggered on the destination route once the transition has finished.

Theory is great, but practical examples make concepts stick a lot better than just talking about them, so we'll use (almost) all of the above hooks and actions to improve our application.

New styles

Several css rules need to be added in this chapter so that things look decent. Here they are:

```
/* app/styles/app.css */

(...)
.page-header a:hover {
  text-decoration: none;
}

.band-info {
  margin-top: 20px;
}

.nav-tabs > li > a.active,
.nav-tabs > li > a.active:hover,
.nav-tabs > li > a.active:focus {
  color: #555;
  cursor: default;
  background-color: #fff;
  border: 1px solid #ddd;
  border-bottom-color: transparent;
}

.band-description-header {
  margin-bottom: 20px;
}
```

Redirection

The classical use of the `beforeModel` and `afterModel` hooks is redirection. The difference is that when `beforeModel` runs, the model is not yet resolved. If we can decide whether to redirect or not (or which route to redirect to) without the model, `beforeModel` should be used, otherwise `afterModel` is appropriate.

Redirecting before the model is known

If we load up our application on the `index` route (the related path being '/') then only the "Rock & Roll" header is shown, since we did not define an `index` template or route. So let's redirect the user to the list of bands instead.

We first need to generate the `index` route:

```
$ ember generate route index
```

Since this redirection is not tied to knowing the model, we can make it as early as possible, in the `beforeModel` hook:

```js
  // app/routes/index.js
  export default Route.extend({
+   beforeModel: function() {
+     this.transitionTo('bands');
+   }
  });
```

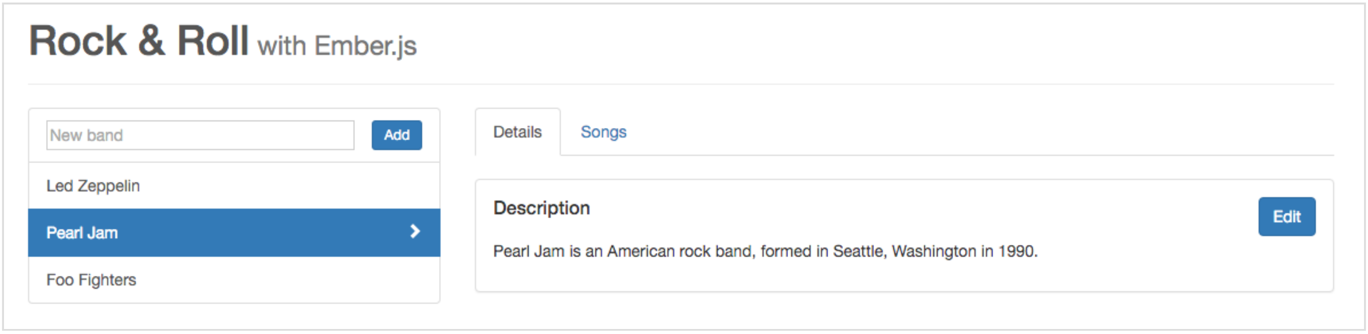
The route object's `transitionTo` method takes a route name, implicitly aborts the current transition (if there is one), and attempts a transition to the new route. Here, the `bands` route serves as our dashboard with the list of bands, so that's where we want to redirect to.

While we are here, let's add a link to the `index` route in the header, so that we can navigate to the 'home' route from everywhere in the application:

```hbs
<!-- app/templates/application.hbs -->
<div class="container">
  <div class="page-header">
+   {{#link-to 'index'}}
      <h1>Rock & Roll<small> with Ember.js</small></h1>
+   {{/link-to}}
  </div>
  <div class="row">
    {{outlet}}
  </div>
</div>
```

Redirecting after the model is known

To demonstrate how `afterModel` accomplishes redirection when taking into account the resolved model, let's introduce a tabbed navigation for each band. One will list the songs (what we already have) while another one will be called "Details" and show a short description of the band:


Rock & Roll with Ember.js

New band Add

Led Zeppelin

Pearl Jam

Foo Fighters

Details Songs

Description Edit

Pearl Jam is an American rock band, formed in Seattle, Washington in 1990.

First, we introduce a new route for the Details tab. Just like `songs`, it should be a subroute of the `bands.band` route:

```
$ ember generate route bands/band/details
```

Make sure that the route was not added to a separate block, like this has been:

```js
  // app/router.js
  Router.map(function() {
    this.route('bands', function() {
      this.route('band', { path: ':slug' }, function() {
        this.route('songs');
+       this.route('details');
      });
    });
  });
```

If it was not, move the `this.route('details')` line under the already existing `band` route, as shown above.

Since we want to show some properties of the band on the Details tab, the model of the route should be the band itself. Using the `modelFor` method introduced in the Nested routes chapter, we can easily accomplish that:

```js
  // app/routes/bands/band/details.js
  export default Route.extend({
+   model: function() {
+     return this.modelFor('bands.band');
+   }
  });
```

The "Only code that is worth writing should be written" principle is employed again. Since reusing the model of the parent route in child routes is a common pattern, beginning in Ember 1.5.0 child routes inherit the model of their parent by default. That means we could remove the above file altogether and the page would still work. We'll now only remove the `model` hook since we'll need to add code to this route later in this chapter. (Note that we can also remove the `model` hook of the `bands.band.songs` route following the same logic.)

Let's say that when a band is selected we want to show the description if there is one, and show the list of songs otherwise. There are a few things we need to modify to make this work.

First, let's add a description property to the band and create one of our "seed" bands with a description so that we can test this feature on it:

```js
// app/models/band.js
export default EmberObject.extend({
  name: '',
+ description: '',
  (...)
});
```

```js
// app/routes/bands.js
export default Route.extend({
  model: function() {
    (...)
    var ledZeppelin = Band.create({
      name: 'Led Zeppelin',
      songs: [blackDog]
    });
    var pearlJam = Band.create({
      name: 'Pearl Jam',
+     description: 'Pearl Jam is an American rock band, formed in Seattle, Washington in 1990.',
      songs: [yellowLedbetter, daughter]
    });
    var fooFighters = Band.create({
      name: 'Foo Fighters',
      songs: [pretender]
    });
  },
  (...)
});
```

Second, the band links currently point to the `bands.band.songs` route. We want the top-level route for the `bands.band` route to act as a redirect hub, so we need to modify those links to point to `bands.band`:

```
  <!-- app/templates/bands.hbs -->
  <div class="col-md-4">
    <div class="list-group">
      (...)
      {{#each model as |band|}}
-       {{#link-to "bands.band.songs" band class="list-group-item band-link"}}
+       {{#link-to "bands.band" band class="list-group-item band-link"}}
          {{band.name}}
          <span class="pointer glyphicon glyphicon-chevron-right"></span>
        {{/link-to}}
      {{/each}}
    </div>
  </div>
  (...)
```

Third, we need to make the redirection in the `bands.band` route. We can't use the `beforeModel` hook in this case since the model (the band in question) is not yet known at that point. The `afterModel` hook, however, is a perfect place for this:

```js
// app/routes/bands/band.js
export default Route.extend({
  model: function(params) {
    var bands = this.modelFor('bands');
    return bands.findBy('slug', params.slug);
  },

+  afterModel: function(band) {
+    var description = band.get('description');
+    if (isEmpty(description)) {
+      this.transitionTo('bands.band.songs');
+    } else {
+      this.transitionTo('bands.band.details');
+    }
+  }
});
```

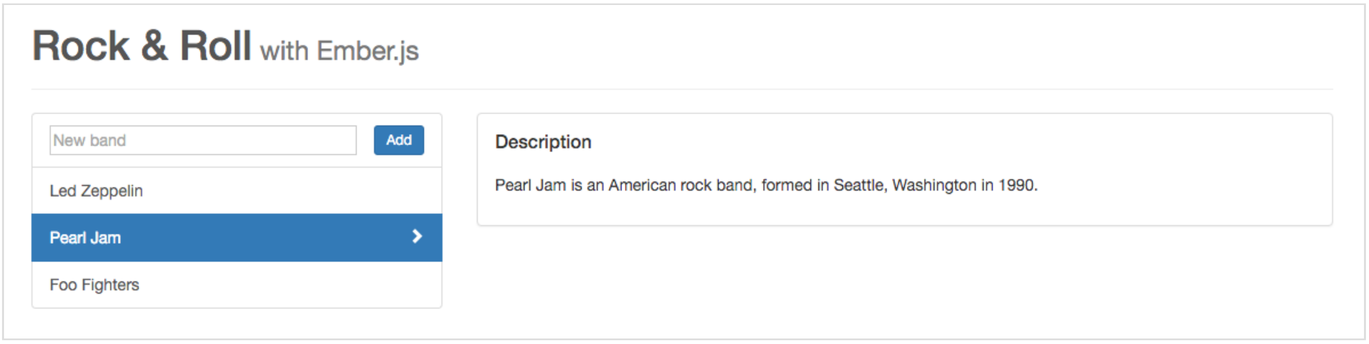
JS

Now, let's display the band's description on the Details page:

```hbs
<!-- app/templates/bands/band/details.hbs -->
<div class="panel panel-default band-panel">
  <div class="panel-body">
    <h4 class="panel-title band-description-header">Description</h4>
    <p>{{model.description}}</p>
  </div>
</div>
```

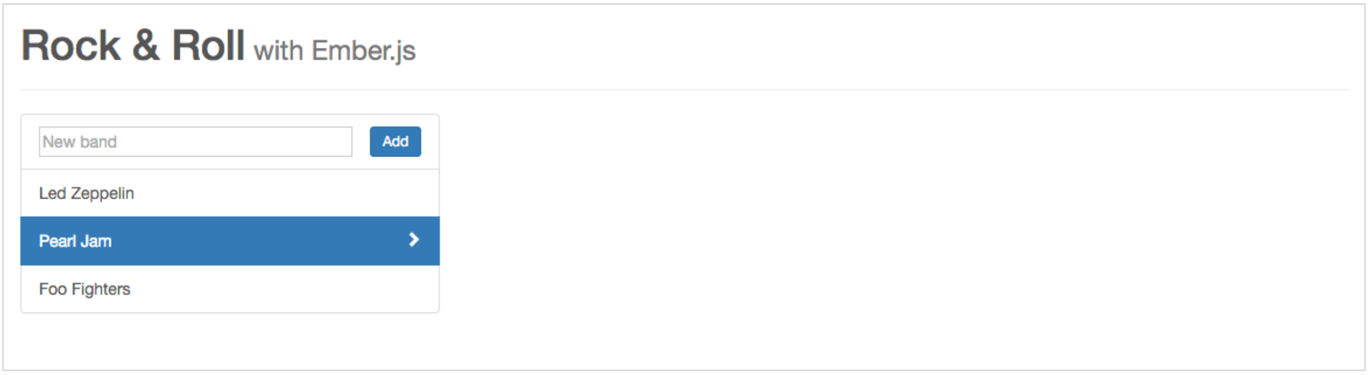
HBS

If you load up the application on `/bands/pearl-jam` everything works as expected and the app is transitioned to the `bands.band.details` route (while the URL changes to `/bands/pearl-jam/details`) since the band has a description:



Pearl Jam is an American rock band, formed in Seattle, Washington in 1990.

However, if you click on the Pearl Jam link again, you see an empty panel for the band instead of the band's description.



The Ember router can only transition to leaf routes so clicking the above `{{#link-to "bands.band" ...}}` will actually attempt a transition from `bands.band.details` to `bands.band.index`. Since it does not need to pass by the common parent route `bands.band` to complete this transition, the redirection code defined in the `bands.band` route is not executed.

This subtle bug can be squashed by moving the redirection to `bands.band.index`, so let's create that route:

```
$ ember generate route bands/band/index
```

And then cut-paste the code into it from the `bands.band` route:

```js
// app/routes/bands/band/index.js
export default Route.extend({
  afterModel: function(band) {
    var description = band.get('description');
    if (isEmpty(description)) {
      this.transitionTo('bands.band.songs');
    } else {
      this.transitionTo('bands.band.details');
    }
  }
});
```

Now everything works correctly! If we click the link for Pearl Jam, the only band that has a description, the description tab is shown; for all the other bands, it is the list of songs.

Finally, let's add the tabbed navigation in the template for the `bands.band` route so that we can manually switch between the `details` and the `songs` subroutes:

```hbs
<!-- app/templates/bands/band.hbs -->
<ul class="nav nav-tabs">
  <li>{{link-to "Details" "bands.band.details" model}}</li>
  <li>{{link-to "Songs"   "bands.band.songs"   model}}</li>
</ul>
<div class="band-info">
  {{outlet}}
</div>
```

SKIPPING MODEL RESOLUTION

BACKSTAGE

Since we are talking about subtleties in the route's model hooks, let me present you with a rookie trap.

If you add a console.log to the model hook of the `band` route and start the application on the `bands/pearl-jam` URL, you will see it printed out, as expected:

```js
export default Route.extend({
  model: function(params) {
    console.log('Model hook called for `bands.band` called with', params.slug);
    var bands = this.modelFor('bands');
    return bands.findBy('slug', params.slug);
  }
});
```

JS

However, if you subsequently click on any of the band links, nothing is printed, as if the model hook was not called. The reason for this is the following.

If the route has a dynamic segment (`bands.band` in our case has `:slug`) and the context object (the band object) is passed in, the model hook is skipped.

This might trip up a lot of people, but it makes sense. The main role of the model hook is to fetch the model object for the subsequent template rendering. If it is already known before starting the transition, there is no need to run the possibly time-consuming function.

So, in what ways can the context object be passed in to skip the model resolution? We have already seen one in the band links:

```
<!-- app/templates/bands.hbs -->
{{#each model as |band|}}
  {{#link-to "bands.band" band class="list-group-item
band-link"}}
    {{band.name}}
    <span class="pointer glyphicon
glyphicon-chevron-right"></span>
  {{/link-to}}
{{/each}}
```

HBS

Another way to pass in the context object is with "application code." We could, for example, transition to the first band in the `bands` route if we realize there is only one band:

```
// app/routes/bands.js
export default Route.extend({
  model: function() {
    return bands;
  },

  afterModel: function(model) {
    var bands = model;
    if (bands.length === 1) {
      this.transitionTo('bands.band', bands.get('firstObject'));
    }
  },
  (...)
});
```

JS

The takeaway is that you shouldn't place any code in the model hook of routes with a dynamic segment that you expect to be run every time, regardless of how the route was transitioned to.

If for whatever reason, you do want the call to be made, you can use the following trick. Instead of the actual object, just pass in its identifier for the dynamic route segment - in our case, the band slug:

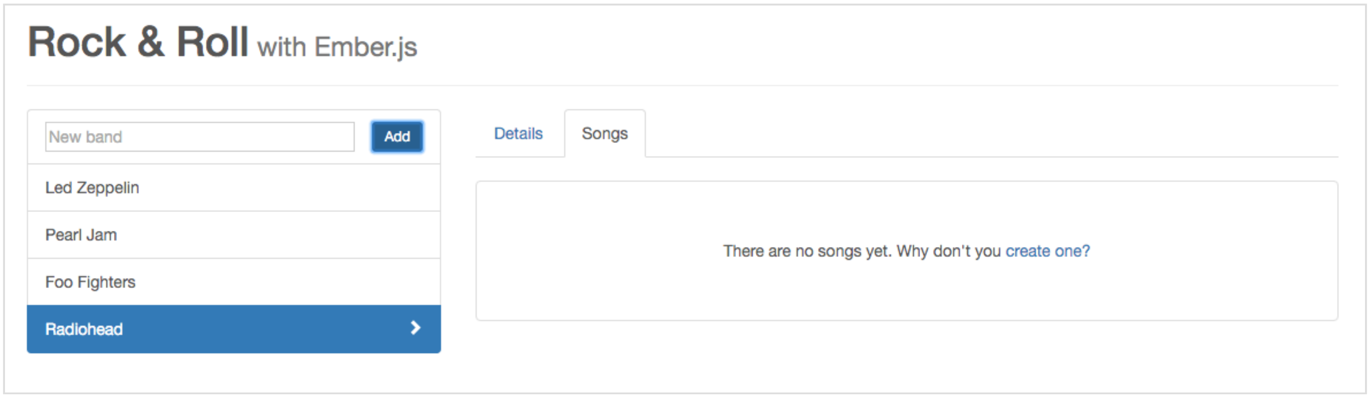
```hbs
<!-- app/templates/bands.hbs -->
{{#each model as |band|}}
  {{#link-to "bands.band" band.slug class="list-group-item
band-link"}}
    {{band.name}}
    <span class="pointer glyphicon
glyphicon-chevron-right"></span>
  {{/link-to}}
{{/each}}
```

HBS

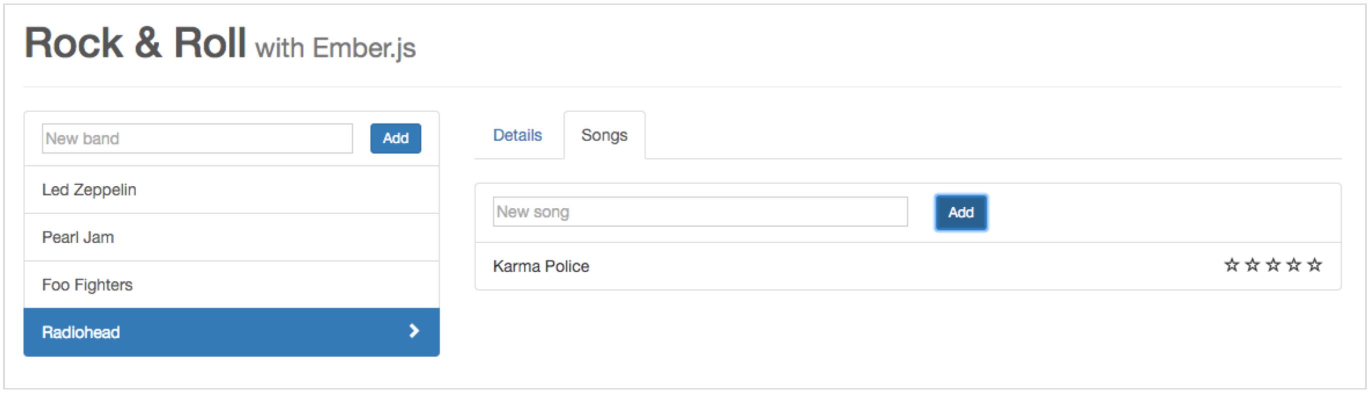
Resetting controller properties

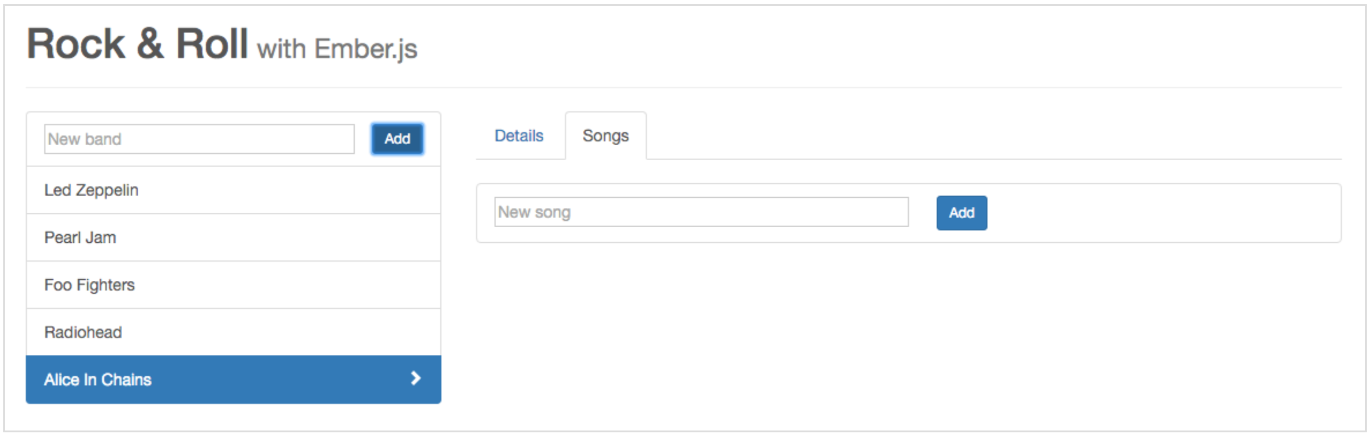
There are some other, less known and used route hooks, like `activate`, `deactivate` and `resetController`. We will use the latter to fix the following subtle bug.

If you create a new band and then go through the above two-step song creation process (first clicking on the "create one" link, then typing in a title and clicking the Save button), the next time you create a band, you will find the input box for the new song instead of the "Why don't you create one?" link:



There are no songs yet. Why don't you create one?





The source of this bug is a fact that is a main source of bugs in Ember and even advanced Ember developers can introduce bugs of this kind.

Controllers in Ember are singletons and thus are not torn down when their route is navigated away from or when their model changes. That means their properties persist while the app lives and anything that needs to be reset needs to be done manually.

What happens in the above case is that `songCreationStarted` flag is set to true when we click the "create one" link the first time and this value persists for the next time we want to create a song for another band. We thus need to reset that controller property if we want to allow restarting the 2-step song creation process.

Each route has a `resetController` hook that will be called when the route is being exited or when the model for the route changes:

```js
resetController: function(controller, isExiting, transition) { ... }
```

`isExiting` is true if we move to another route, false if only the model changes. In our case, we want to reset the `songCreationStarted` to false in both cases:

```
  // app/routes/bands/band/songs.js
  export default Route.extend({
    model: function() {
      return this.modelFor('bands.band');
    },

+   resetController: function(controller) {
+     controller.set('songCreationStarted', false);
+   },
    (...)
  });
```

We can now create a song in two steps several times.

Leaving routes and arriving at them

We saw how different route hooks can be leveraged to modify the flow of transition and redirect mid-way to another route. Another category of tasks is enabled by actions that are fired on routes. We'll see two of them here, `didTransition` and `willTransition`.

Setting descriptive page titles

Changing the title of the browser tab as the user navigates the screens of a Single Page Application shows extra attention to detail.

The `didTransition` action is a suitable place to have this functionality. It fires on the destination route when the transition has completed.

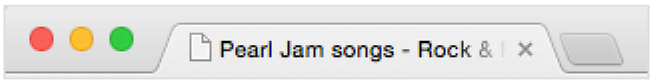
We'll first set a single, static text as the title for the `bands` route:

```js
  // app/routes/bands.js
  export default Route.extend({
    (...)
    actions: {
+     didTransition: function() {
+       document.title = 'Bands - Rock & Roll';
+     },
      (...)
  });
```

For the `bands.band.songs` route, though, let's also include the band name in the title:

```js
  // app/routes/bands/band/songs.js
  export default Route.extend({
    (...)
    actions: {
+     didTransition: function() {
+       var band = this.modelFor('bands.band');
+       document.title = `${band.get('name')} songs - Rock & Roll`;
+     },
      (...)
    }
  });
```

We now have browser tab titles that are clearly distinguishable at a glance:



Warning about losing data

`willTransition` is `didTransition`'s pair. It fires on the current route when a transition is about to begin. It is most often used to warn a user about potentially losing any unsaved changes.

To demonstrate the use of `willTransition` for this case, let's make the band description editable when you click a button on the Details tab:

```
  <!-- app/templates/bands/band/details.hbs -->
  <div class="panel panel-default band-panel">
    <div class="panel-body">
-     <h4 class="panel-title band-description-header">Description</h4>
-     <p>{{model.description}}</p>
+     <h4 class="panel-title band-description-header pull-left">Description</h4>
+     {{#if isEditing}}
+       <button class="btn btn-primary pull-right" {{action "save"}}>Save</button>
+     {{else}}
+       <button class="btn btn-primary pull-right" {{action "edit"}}>Edit</button>
+     {{/if}}
+     <div class="clearfix"></div>
+     {{#if isEditing}}
+       <div class="form-group">
+         {{textarea class="form-control" value=model.description}}
+       </div>
+     {{else}}
+       <p>{{model.description}}</p>
+     {{/if}}
    </div>
  </div>
```

`isEditing` is a flag on the controller that controls what is rendered. When it is false, the current description is shown, along with an Edit button that enables switching to 'edit mode'.

When `isEditing` becomes true, the button is turned into a Save button while the description paragraph is turned into a textarea where it can be edited. Clicking "Save" brings us all the way back to where we started.

Let's create the controller for `bands.band.details`:

```
$ ember generate controller bands/band/details
```

, and paste in the following content:

```js
// app/controllers/bands/band/details.js
export default Controller.extend({
  isEditing: false,

  actions: {
    edit: function() {
      this.set('isEditing', true);
    },
    save: function() {
      this.set('isEditing', false);
    }
  }
});
```

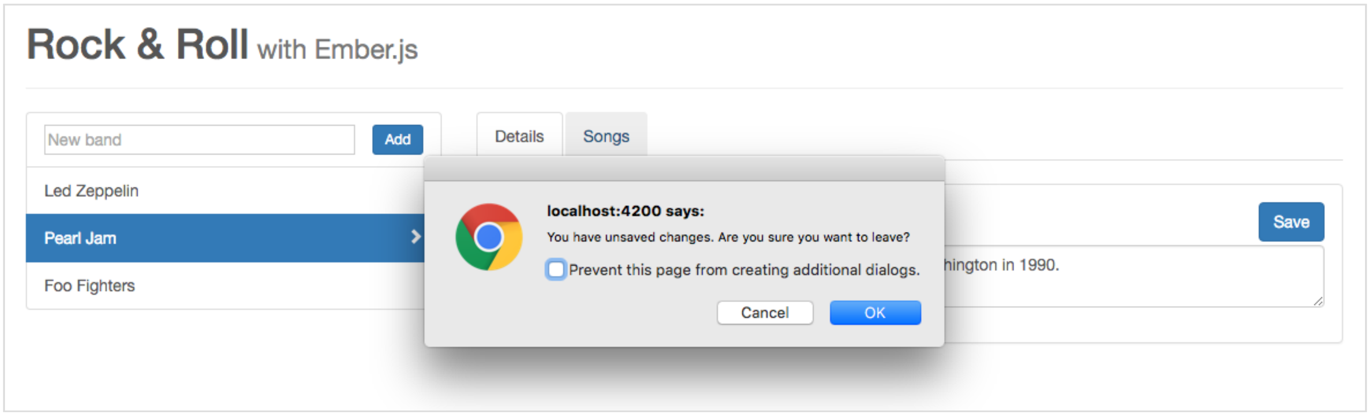
We want to alert the user when they are about to leave the page while they're in the middle of editing the description. In such a case, `willTransition` will be fired on the current route, `bands.band.details`. All we have to do is verify that the controller was in edit mode and have the user confirm they really meant to leave the page:

```diff
  // app/routes/bands/band/details.js


  export default Route.extend({
-   model: function() {
-     return this.modelFor('bands.band');
-   },

+   actions: {
+     willTransition: function(transition) {
+       var controller = this.get('controller'),
+           leave;
+
+       if (controller.get('isEditing')) {
+         leave = window.confirm("You have unsaved changes. Are you sure you want to leave?");
+         if (leave) {
+           controller.set('isEditing', false);
+         } else {
+           transition.abort();
+         }
+       }
+     }
+   }
  });
```

The important piece is `transition.abort()`. It aborts the current transition, staying on the current page.

Note that the current transition is also aborted in the redirection cases. The difference is that there it's done implicitly, by starting a new transition to another route (this is essentially what a redirect does), whereas here it needs to be spelled out explicitly since there is no new route to go to.

It is also worth noting that no changes are lost when navigating away from the page without saving the new description. Since the value of the textarea is synced to the description of the band, the same description is going to be there when the user comes back to the band details page. It would be more correct to warn about losing changes when the tab is about to be closed. There, however, the user might no longer be on the page and our warning could thus seem out of context. So while factually our warning is not correct, it might make more sense than elsewhere.



localhost:4200 says:
You have unsaved changes. Are you sure you want to leave?
Prevent this page from creating additional dialogs.

The astute reader might rightfully note that there are actually no unsaved changes since the binding between the value of the textarea and the description property of the band keeps these values in sync. In another sense, because at this point everything is only saved in memory, everything is an unsaved change.

In the next chapter we'll persist these records on the backend so this will make more sense then but the warning might be better issued when the user is about to close the tab.

Clean-up

The `ember generate route index` command generated the route that we used to redirect to `bands` but it also generated a corresponding template. We don't need this so it's better to remove it:

```
$ rm app/templates/index.hbs
```

The same is true for the `bands/band/index` template. We generated the `bands.band.index` route only for redirection purposes so the template should be removed:

```
$ rm app/templates/bands/band/index.hbs
```

There is also a huge number of test files that we can remove:

```
$ rm tests/unit/routes/index-test.js
$ rm tests/unit/routes/bands/band/index-test.js
$ rm tests/unit/routes/bands/band/details-test.js
$ rm tests/unit/controllers/bands/band/details-test.js
```

Next song

Our app now improves the user experience, saving the user a few clicks by redirecting to the right place. It sets the document title so that the user can see which page is active just by glancing at the tab. It also prevents the user from losing unsaved changes by warning them when they're about to leave the page.

Well, actually, they'll lose all of their work when they close the tab or reload the page. That is because everything is just stored in memory; nothing is written to disk. In the next chapter we'll hook up our application to a backend to make our changes stick around.

CHAPTER 10

Talking to a backend

Tuning

Our application is really starting to take form, and feels less and less "Let's build a blog in 10 minutes"-like. There is still one aspect which makes it look like a toy application, though: nothing is persisted; everything lives only until we close the browser tab.

In this chapter, we'll make our app talk to an API, and our models will be persisted on a backend and loaded from there. The library that we use for this is Ember Data.

The concept behind Ember Data

Ember Data is a model persistence library for Ember.js. It manages communication between the front-end application and an API that persists and loads model data.

It abstracts away from loading and saving data to the backend, instead storing and retrieving model objects on the frontend and minimizing round-trips to the server. It also manages relationships between our model classes. Last, but not least, it is highly customizable, so it is relatively easy to make Ember Data work with the API which is important if we don't have control over it.

The API

Since this is a book on Ember.js, we can treat the API that sends data for the front-end application to consume as a black box. I made that black box send back (and accept) data in the format that Ember Data works with out-of-the-box, without any customizations.

The API follows the JSON API convention, a specification that became stable in May 2015. It is an anti-bikeshedding weapon since it lays down a standard way to send data back and forth so teams no longer have to spend precious development time arguing over the minutiae of API design.

Above the core functionalities of an API, JSON API also deals with relationships, sparse fieldsets, included resources, pagination and so on. It is a fantastic tool to work with and if you start a project today, I totally recommend using it.

At each step we take to connect our app to the API, I'll explain the shape of the data that the API, which is available at http://json-api.rockandrollwithemberjs.com, sends or expects in detail.

Model classes

Model classes descend from `DS.Model` where `DS` is the top-level name exported by Ember Data (short for Data Store).

Our Band class needs to be altered:

```
  // app/models/band.js
+ import DS from 'ember-data';

- export default EmberObject.extend({
+ export default DS.Model.extend({
-   name:         '',
-   description:  '',
+   name:         DS.attr('string'),
+   description:  DS.attr(),
+   songs:        DS.hasMany('song'),

-   init: function() {
-     this._super(...arguments);
-     if (!this.get('songs')) {
-       this.set('songs', []);
-     }
-   },

    slug: computed('name', function() {
      return this.get('name').dasherize();
    }),
  });
```

`DS.attr` specifies that the property is an attribute of the model. The first argument is the attribute type (more precisely, the name of the transform applied to this attribute when sending to/receiving from the server), which can be omitted if it's a string.

By defining `songs` as a hasMany property, we tell Ember Data to put the Song objects belonging to a particular band in an array that can be accessed as `band.get('songs')`.

It would be possible to create a model class using a generator, too:

```
$ ember generate model band
```

However, since the `Band` class already existed, we would have had to prevent it from being overwritten by the generator and do the modifications ourselves anyway.

Let's see the other model class, `Song`:

```
  // app/models/song.js
+ import DS from 'ember-data';

- export default EmberObject.extend({
+ export default DS.Model.extend({
-   title: '',
-   rating: 0,
-   band: null
+   title:  DS.attr('string'),
+   rating: DS.attr('number'),
+   band:   DS.belongsTo('band')
  });
```

JS

`Band` and `Song` are connected to each other through a has-many relationship from the perspective of Band. It has (potentially) many songs but a song belongs to exactly one band. By setting up the above relationship in our model classes and having the backend send its response in the right format, the instances of Band can access their songs via `band.get('songs')`. In the same vein, any `Song` instance gets a handle to its band by calling `song.get('band')`.

Transforming the app to use Ember Data

Instead of giving an abstract run-down of Ember Data features, we'll adjust our app piece by piece and I'll explain the relevant parts of the library. This follows the structure of the whole book so far, where as we add a new feature to the application, we leverage a certain Ember building block (for example a route, a controller, or a component) and then reveal how it works its magic.

Loading bands

Let's start adjusting our application in a top-down manner, in the same manner we did for nested routes.

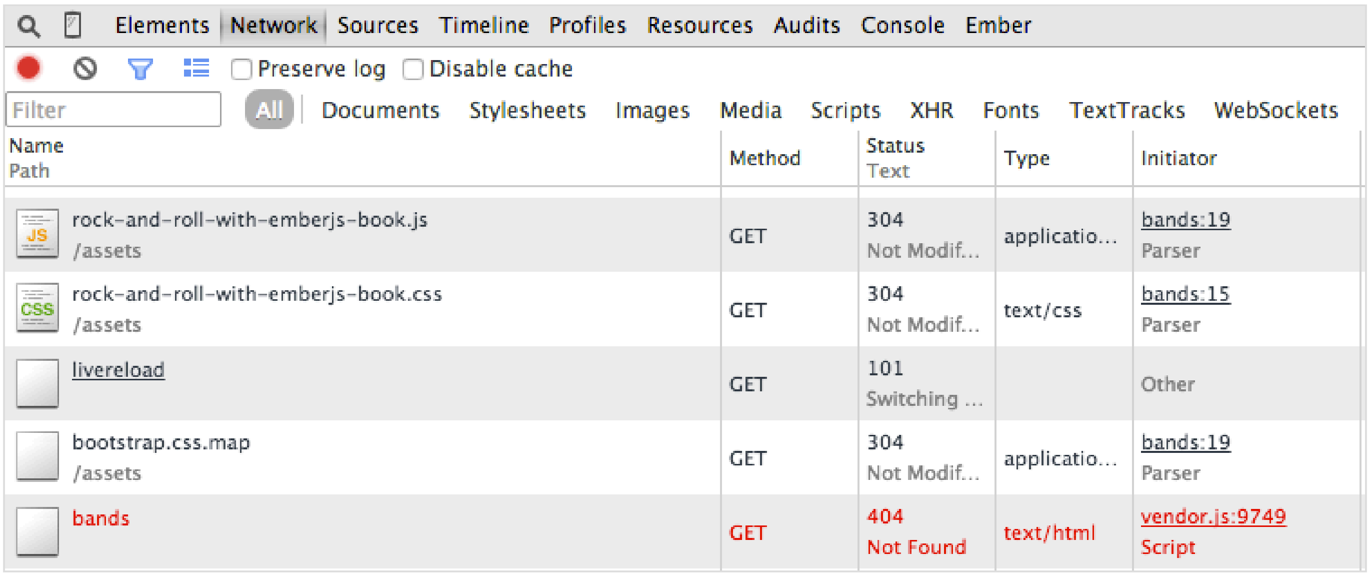
When the application is entered via the URL, the model hook of the `bands` route is called. We now return the contents of the array we manually pushed the created bands into. This is not how Ember Data works. The application developer does not have to deal with maintaining a collection of each type of objects, let alone pushing and removing from these collections. Instead, we interface with an object called the `store` which does this for us - and more, as we will see shortly.

So let's delete all the seed code to create bands and songs and let them be returned by the backend.

We tell the store to fetch all bands:

```diff
  // app/routes/bands.js
- import Band from 'rarwe/models/band';
- import Song from 'rarwe/models/song';

  export default Route.extend({
    model: function() {
-     // All band and song creation code was here
-     return [ledZeppelin, pearlJam, fooFighters];
+     return this.store.findAll('band');
    },
    (...)
  });
```

However, when the app is reloaded, we got a 404 when trying to fetch the bands:



Name Path	Method	Status Text	Type	Initiator
rock-and-roll-with-emberjs-book.js /assets	GET	304 Not Modif...	applicatio...	bands:19 Parser
rock-and-roll-with-emberjs-book.css /assets	GET	304 Not Modif...	text/css	bands:15 Parser
livereload	GET	101 Switching ...		Other
bootstrap.css.map /assets	GET	304 Not Modif...	applicatio...	bands:19 Parser
bands	GET	404 Not Found	text/html	vendor.js:9749 Script

Ember Data is doing its best to find out where data for all bands should be loaded from. By default, it assumes the xhr requests are served from the same host and port. In our case, though, the API server runs at `http://json-api.rockandrollwithemberjs.com`, so we should override this default assumption.

Let's kill the `ember serve` process that runs our application and restart it using the following command:

```
$ ember serve --proxy=http://json-api.rockandrollwithemberjs.com
```

The `--proxy` option establishes a proxy that all xhr requests will be sent through to the URL defined as the option's value. So the above command sends the ajax requests to our API host in a way that also gets around the CORS issues (Cross Origin Resource Sharing) that would have been possible had we sent the requests directly to the API.

Now, `store.findAll('band')` sends the request to fetch all bands to the right place through the proxy. Let's see what the format of the response should look like so that Ember Data is able to create the front-end objects and everything works:

```
"data": [
  {
      "id": "1",
      "type": "bands",
      "links": {
        "self": "http://localhost:4200/bands/1"
      },
      "attributes": {
        "name": "Pearl Jam",
        "description": "Pearl Jam is an American rock band, formed in Seattle, Washington in 1990."
      },
      "relationships": {
        "songs": {
          "links": {
            "self": "http://localhost:4200/bands/1/relationships/songs",
            "related": "http://localhost:4200/bands/1/songs"
          }
        }
      }
  },
  {
      "id": "2",
      "type": "bands",
      "links": {
        "self": "http://localhost:4200/bands/2"
      },
      "attributes": {
        "name": "Led Zeppelin",
        "description": null
      },
      (...)
  },
  (...)
]
```

JS

A few things to note:

1. JSON API sends back data with a top-level key called `data`, which is an array of items since we asked for a collection.
2. Each item has an `id` and a `type` where the type key is in plural (although this is not mandatory, you can also use the singular as long as you are consistent)
3. The attributes for each item live under an `attributes` hash.
4. Since we did not ask for related resources to be included in the response, those are reachable through the links in the `relationships` hash.

Fetching a single band

The next step is to retrieve a single band for the next level, the `band` route.

As seen when fetching all bands, Ember Data retrieves data via the store, which is present on all route objects as `this.store`. The only difference now is that we only need one band object:

```js
// app/routes/bands/band.js
export default Route.extend({
  model: function(params) {
-   var bands = this.modelFor('bands');
-   return bands.get('content').findBy('slug', params.slug);
+   return this.store.findRecord('band', params.id);
  }
});
```

The call is quite similar, too. The only difference is that we passed the id of the band we would like to retrieve as the second argument.

We changed from using the slug of the band to identify it to using its id. Ember Data could be made to work with slugs, too, but to keep the amount of new information low and the app more robust, we've switched to ids. That change also makes it necessary to amend the appropriate route definition:

```js
  // app/router.js
  (...)
  Router.map(function() {
    this.route('bands', function() {
-     this.route('band', { path: ':slug' }, function() {
+     this.route('band', { path: ':id' }, function() {
        this.route('songs');
        this.route('details');
      });
    });
  });
```

We can also get rid of the `slug` computed property in `Band` which then will have the following content:

```js
  // app/models/band.js
  import DS from 'ember-data';
- import { computed } from "@ember/object";

  export default DS.Model.extend({
    name:         DS.attr('string'),
    description:  DS.attr(),
    songs:        DS.hasMany('song'),

-   slug: computed('name', function() {
-     return this.get('name').dasherize();
-   })
  });
```

If we watch the network calls in the browser when we initiate the app on `/bands/1/songs`, we see that two ajax requests go out, one to `/bands` and another one to `/bands/1`. The second one is due to how Ember Data 2.0 fetches records in the background. By default, if a record is found by its type and id, the store will return that record but it will also fetch the record from the backend in the background. This ensures that a record is returned immediately whenever it is available but also that the client-side records are kept in sync with those on the backend without blocking UI rendering.

Loading songs for a band

The good news is that the `bands.band.songs` route does not need any changes to keep working. Before migrating to Ember Data, each band object had an array for its songs that we pushed objects into manually. With Ember Data, declaring the has-many relationship and returning the data in the right format correctly sets up the `songs` property to contain the songs belonging to the band.

The code does not change one bit:

```js
// app/routes/bands/band/songs.js
export default Route.extend({
  model: function() {
    return this.modelFor('bands.band');
  },
  (...)
});
```

With that, our loading concerns are done. All that's left is saving bands and songs.

Creating a band

Currently, creating a band happens with the following code snippet:

```js
// app/routes/bands.js
import Route from '@ember/routing/route';
import Band from 'rarwe/models/band';

export default Route.extend({
  (...)
  actions: {
    createBand: function() {
      var name = this.get('controller').get('name');
      var band = Band.create({ name: name });
      bands.get('content').pushObject(band);
      this.get('controller').set('name', '');
      this.transitionTo('bands.band.songs', band);
    }
  }
});
```

JS

The `Band` class had previously been imported from its module and then an instance of it pushed to the content of the global `bands` array. As you might expect, this is not the Ember Data way. We'll turn to the store again to create a band object and then save it:

```js
  // app/routes/bands.js
  import Route from '@ember/routing/route';
- import Band from 'rarwe/models/band';

  export default Route.extend({
    (...)
    actions: {
      (...)
      createBand: function() {
-       var name = this.get('controller').get('name');
-       var band = Band.create({ name: name });
-       bands.get('content').pushObject(band);
-       this.get('controller').set('name', '');
-       this.transitionTo('bands.band.songs', band);

+       var route = this;
+       var controller = this.get('controller');
+
+       var band = this.store.createRecord('band', controller.getProperties('name'));
+       band.save().then(function() {
+         controller.set('name', '');
+         route.transitionTo('bands.band.songs', band);
+       });
      }
    }
  });
```

`createRecord` takes a type name as its first argument and an object from which the properties are built as its second. `getProperties`, present on all `EmberObjects`, is a handy method to create such an object with passed-in keys, where the values are the object's property values belonging to the keys. So `controller.getProperties('name')` results in something like `{ name: "Tool" }`, which is exactly what `createRecord` needs.

The call to `save` sends a POST request to `/bands` with the serialized object as the payload and returns a promise (see Backstage section). When the backend operation has successfully completed, the function passed to `then` is going to be called. We do just as before: by setting the name property to an empty string we clear the text input so that a new band can be created, and then we go to the songs page of the new band.

The format of the server's response is the following:

```json
{
  "data": {
    "id": "7",
    "type": "bands",
    "links": {
      "self": "http://localhost:4200/bands/7"
    },
    "attributes": {
      "name": "Tool",
      "description": null
    },
    "relationships": {
      "songs": {
        "links": {
          "self": "http://localhost:4200/bands/7/relationships/songs",
          "related": "http://localhost:4200/bands/7/songs"
        }
      }
    }
  }
}
```

JS

The server responds with the newly minted resource. Note that the value of the top-level `data` key is now a single object since the request targeted a single resource.

Creating and saving a song is almost identical so I'll just quickly show the diff:

```
  // app/routes/bands/band/songs.js
  import Route from '@ember/routing/route';
- import Song from 'rarwe/models/song';

  export default Route.extend({
    (...),
    actions: {
      (...)
      createSong: function() {
        var controller = this.get('controller');
        var band = this.modelFor('bands.band');

-       var title = controller.get('title');
-       var song = Song.create({ title: title, band: band });
-       band.get('songs').pushObject(song);
-       controller.set('title', '');
+       var song = this.store.createRecord('song', {
+         title: controller.get('title'),
+         band: band
+       });
+       song.save().then(function() {
+         controller.set('title', '');
+       });
      },
    }
  });
```
JS

BACKSTAGE

PROMISES

Promises are a way of dealing with asynchronous operations. A promise is a construct that "eventually" produces a value and is either fulfilled (usually associated with success) or rejected (implying failure).

The way to interact with a promise is through its `then` method. It takes a fulfillment handler and – optionally – a rejection handler and calls the one corresponding to the result of the promise resolution. So if resolving the promise was successful, the fulfillment handler is called with the result; if there was an error during promise resolution or we want to indicate a failure scenario, the rejection handler is called with the error reason.

```js
import { Promise as EmberPromise } from 'rsvp';

var yayOrNay = function() {
  return new EmberPromise(function(resolve, reject) {
    if (Math.random() > 0.5) {
      resolve(42);
    } else {
      reject('Sorry, try again.');
    }
  });
}

function yay(value) { console.log('Yay, success: ' + value) };
function nay(error) { console.log(error) };

yayOrNay().then(yay, nay);
```

JS

If we are lucky and p is resolved with success, we see "Yay, success: 42" printed out to the console. Otherwise, `nay` is called and the message "Sorry, try again." is the output.

A very common use of promises is ajax requests. Sending a request returns a promise. We then pass a function that deals with the returned data as the fulfillment handler and another function that handles the error case:

```js
ajaxRequest.then(function(data) {
  // Process data
}, function(error) {
  // Handle error
});
```

In Ember, promises are very strongly relied upon. One such scenario is returning promises from a model hook. Until the promises created in the model hook are resolved (either with success or with failure), template rendering will not take place. This means that the user will not see a half-rendered view where data pops in as it's returned asynchronously. The promises must fetch and return all their data in order for template rendering to start.

Actually, we've already leveraged this. All `find` methods of Ember Data that we used in the model hooks (`findAll` and `findRecord`) of our routes returns a promise. Consequently, rendering the list of bands or the songs/details of a single band did not happen until the relevant data came in.

If you would like to know more about promises, I suggest you read my introductory blog post, a more advanced level article or the Promises/A+ spec which might sound intimidating but is fairly readable and short.

Updating a band's description

Let's end the chapter by persisting update operations, namely a band's description and song ratings. I'll also show you a trick in the process.

Editing the description happens in the `bands.band.details` controller and the save action is already implemented. All it does at the moment, though, is flip back the `isEditing` switch to false so that the textarea becomes a static text again:

```js
// app/controllers/bands/band/details.js
export default Controller.extend({
  isEditing: false,

  actions: {
    (...)
    save: function() {
      this.set('isEditing', false);
    }
  }
});
```

What needs to happen in addition to that is saving the updated description to the backend. In other words, updating the band object.

We saw that when an action is fired, Ember first looks up the action on the controller. Then, if it does not find it, it keeps looking on the current route, moving up the route tree. If the action is not even found on the application route, an error occurs.

I mentioned earlier that Ember allows handling of an action multiple times by returning `true` from a handler. We'll leverage this technique here:

```js
// app/controllers/bands/band/details.js
export default Controller.extend({
  isEditing: false,

  actions: {
    (...)
    save: function() {
      this.set('isEditing', false);
+     return true;
    }
  }
});
```

By doing this, the action is sent to the current route, `bands.band.details`. There, we can save the band object:

```js
// app/routes/bands/band/details.js
export default Route.extend({
  actions: {
    save: function() {
      var controller = this.get('controller'),
          band = controller.get('model');

      return band.save();
    },
    (...)
  }
});
```

We could have put all of the action handling code in one place, either the controller or the route. The advantage of this split is that the code that implements the change in the controller state lives in the controller, while the code that introduces a state change in the model resides in the route.

Calling `save` on an object that has been loaded from the backend sends a PATCH request to the resource URL for the object, in this case, `bands/1`, to which the backend replies with the representation of the whole resource:

```
{
  "data": {
    "id": "1",
      "type": "bands",
      "links": {
        "self": "http://localhost:4200/bands/1"
      },
      "attributes": {
        "name": "Pearl Jam",
        "description": "Pearl Jam is an American rock band, formed in Seattle, Washington in 1990."
      },
      "relationships": {
        "songs": {
          "links": {
            "self": "http://localhost:4200/bands/1/relationships/songs",
            "related": "http://localhost:4200/bands/1/songs"
          }
        }
      }
  }
}
```

Updating song ratings

Finally, let's also make sure that song ratings are persisted. We only have to add a `song.save()` line to the end of `updateRating` so that a PATCH request is sent to the backend for the appropriate song:

```
   1  // app/controllers/bands/band/songs.js
   2  export default Controller.extend({
   3    (...)
   4    actions: {
   5      (...)
   6      updateRating: function(params) {
   7        var song = params.item,
   8            rating = params.rating;
   9
+ 10        if (song.get('rating') === rating) {
+ 11          rating = 0;
+ 12        }
  13        song.set('rating', rating);
+ 14        return song.save();
  15      }
  16    }
  17  });
```

Lines 12 to 14 provide a way to reset our rating for a particular song. If the same star was clicked as the current rating, the current rating will be the same as the new one to set and we interpret it as the user wishing to set the rating to zero.

Next song

Our app has come a long way in this chapter, and it starts to take shape. It would be a shame if something happened to it. To prevent this and make sure we don't break anything as we develop our app even further, we'll add a couple of acceptance tests to safeguard the critical paths of our application.

CHAPTER 11

Testing

Tuning

We established that we want to write a few acceptance tests to prevent breaking something that had previously worked correctly when refactoring code or adding new features.

We will also, in the next chapter, drive the development of a new feature by writing the test for it first. This is called "Test Driven Development" (or "Test First Development").

For high-level (as opposed to unit) tests, the main benefit of writing the test before the implementation is not having to go through the somewhat tedious process of setting up the context in which the new feature runs and can be tested. Instead of logging in, clicking a link, filling out a form, and clicking the submit button, we can have a few lines of code that automates this. This way, we save a lot of time and decrease our level of boredom. If we decided to write the test – to protect against regression – for this feature anyway, this is clearly a win.

Clarifying the vocabulary

Different people have come up with different names for exactly what each kind of test means. Some call 'acceptance tests' what others call 'integration tests,' and they disagree on what the differences are. Others use the term 'end-to-end tests' to refer to integration tests (or acceptance tests).

For that reason, I will spell out exactly what I (and Ember) mean by each kind of tests before we start the actual writing of tests.

Acceptance tests

What I call an acceptance test is an automated test that exercises the whole system. These tests automate user actions to cut down on testing time, to prevent user error in the test process, and to compose a test suite that can prevent regression.

It might or might not mock out external dependencies, which mostly means API calls in the case of a front-end application.

Mocking means replacing the actual calling of a service (our API server, Twitter, etc.) with just returning the data that we expected the service to return. It should be noted, though, that if we decide to mock out dependencies, the real application no longer behaves exactly as the application in testing mode does. This is because the exact method of calling the service and the data it returns is detached from the actual service.

To mock or not to mock?

The main advantage of not mocking in acceptance tests is that they provide a real guarantee that your application works as it should. Their main drawback is that they are the heaviest of the bunch, requiring a network connection and all the dependencies being available at the time of running.

To give an example, to test our application locally you would need to run the Ruby backend API on your machine.

Mocking makes tests lighter. Since we can mock out http requests, the tests run a lot faster and don't need access to external services. Their main drawback stems from the same fact: since they don't connect to the real services, the tests might pass even if there is a change in a service that can break the real application. If your app connects to a Twitter endpoint to fetch some data and that endpoint is retired, you'll falsely believe that your app runs just fine, and only realize it doesn't in production.

There is one more advantage to mocking in tests: they don't change the state of the environment once they have run. A non-mocking test that creates an actual band record in the backing database does not leave the environment as it found it, and is thus not a good steward of it. This is less of an issue for backend applications, because database tables can be truncated or recreated relatively easily there. However, in our front-end app we would have to make the backend clear its data by, for example, triggering an endpoint from the front-end app.

For the Rock & Roll application, that tips the balance in favor of mocking.

Unit tests

Acceptance tests verify whether the whole application works correctly. Unit tests verify that a certain unit, completely isolated from its environment, behaves as we expect it to.

What's the use of making sure each cog works fine if we know – because we have acceptance tests – that the whole machine works correctly?

One possible answer, the one given by followers of the BDD (Behavior Driven Development) school, is that the rationale for writing unit tests is not to guarantee their correctness. They serve the purpose of driving the design of the whole system.

By focusing on testing the communication between the components of the system, BDD people claim, the emerging system will have a very flexible, decoupled architecture. The system will be a network of small components where each component only does one thing, and consults the others to get the information it needs.

Another answer to justify the need for unit tests is less architectural. When a high-level test fails, you only have a high-level idea of what went wrong. If you have a suite of unit tests that back up the acceptance test suite, there can be two cases.

If there is also a failing unit test (or several), you'll ideally know which unit is responsible for the error. Since this happens at a lower level, you have a more concrete idea of where the problem lies, which in turn reduces the time spent debugging.

If the unit tests all pass, you can be fairly certain that the error that causes the acceptance test to fail is due to a communication problem between the units. In this case, the problem space got narrower, which also reduces the time needed for debugging.

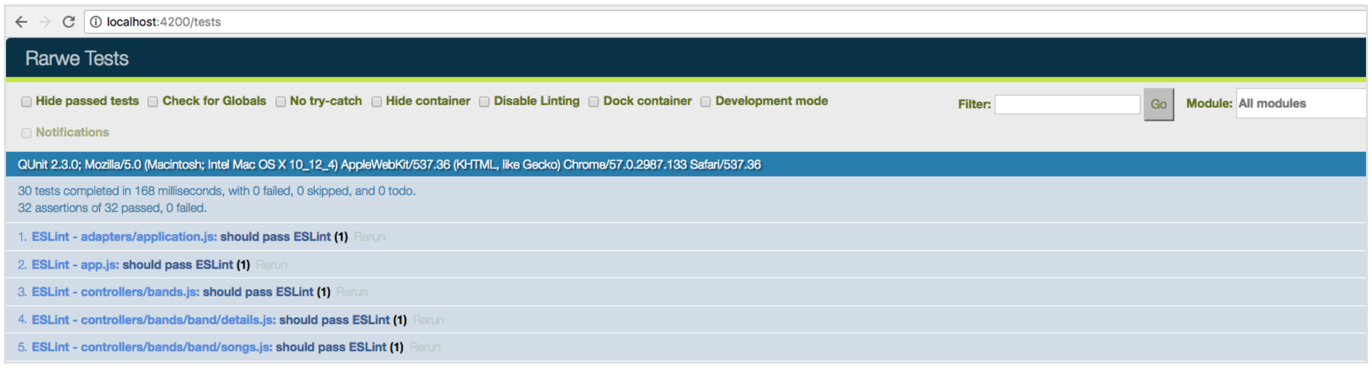
Integration tests

Integration tests verify the interface of an object and are somewhere between acceptance and unit tests. An actual application instance is not spun up for the test, but we do not have to define each collaborator of the object under test for the test to work, as we have to with unit tests.

Adding an acceptance test

With the theory nailed down, it's time to roll up our sleeves and get to work.

All ember-cli projects have a couple of tests included that verify that all project files pass the linter (eslint) tests. They can be run by directing our browser to `http://localhost:4200/tests`:



```
QUnit 2.3.0; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36
30 tests completed in 168 milliseconds, with 0 failed, 0 skipped, and 0 todo.
32 assertions of 32 passed, 0 failed.
1. ESLint - adapters/application.js: should pass ESLint (1) Rerun
2. ESLint - app.js: should pass ESLint (1) Rerun
3. ESLint - controllers/bands.js: should pass ESLint (1) Rerun
4. ESLint - controllers/bands/band/details.js: should pass ESLint (1) Rerun
5. ESLint - controllers/bands/band/songs.js: should pass ESLint (1) Rerun
```

By default all tests are run, but any single test file can be selected with the multi-select box on the right. If you want to drill down further, you can click on the Rerun link next to each test case.

We can see that all tests passed, so we can continue by adding our first acceptance test.

Ember CLI has so-called "blueprints". These blueprints allow us to generate project files – or modify existing ones – with predefined content, customized by the name we pass to the command. There is an acceptance-test blueprint, so creating an acceptance test file is as easy as:

```
$ ember generate acceptance-test bands
```

That generates a file at `tests/acceptance/bands-test.js`. If we take a look inside, here is what we find:

```
// tests/acceptance/bands-test.js
import { test } from 'qunit';
import moduleForAcceptance from 'rarwe/tests/helpers/
module-for-acceptance';

moduleForAcceptance('Acceptance | bands');

test('visiting /bands', function(assert) {
  visit('/bands');

  andThen(function(assert) {
    assert.equal(currentURL(), 'bands');
  });
});
```
JS

Ember CLI uses the QUnit testing framework. Each test file has a module declaration with a name, a `beforeEach`, and an `afterEach` function. These functions, as their name suggests, will run before and after each test case, respectively. Their role is to provide a clear testing environment so that no two test cases influence each other and all tests run with a blank slate.

To provide that clean slate for Ember tests, a new application is set up and started in `beforeEach`, and the same application is destroyed in `afterEach`.

These functions are hidden in the `module-for-acceptance.js` module:

```js
// tests/helpers/module-for-acceptance.js
import { module } from 'qunit';
import { resolve } from 'rsvp';
import startApp from '../helpers/start-app';
import destroyApp from '../helpers/destroy-app';

export default function(name, options = {}) {
  module(name, {
    beforeEach() {
      this.application = startApp();

      if (options.beforeEach) {
        return options.beforeEach.apply(this, arguments);
      }
    },

    afterEach() {
      let afterEach = options.afterEach &&
options.afterEach.apply(this, arguments);
      return resolve(afterEach).then(() =>
destroyApp(this.application));
    }
  });
}
```

JS

Let's run our test by visiting `http://localhost:4200/tests` in our browser.

So far so good, but let's scrap the generated test and create a more valuable one to our app. Instead of inspecting the current route of the application, we are going to assert that all the bands are rendered in the list.

Describing what steps the test should take, and what assertions it should have, gives us the following:

```js
// tests/acceptance/bands-test.js
(...)
- test('visiting /bands', function(assert) {
+ test('List bands', function(assert) {
  visit('/bands');

  andThen(function() {
-   assert.equal(currentURL(), 'bands');
+   assert.equal(find('.band-link').length, 2, 'All band links are rendered');
+   assert.equal(find('.band-link:contains("Radiohead")').length, 1, 'First band link contains the band name');
+   assert.equal(find('.band-link:contains("Long Distance Calling")').length, 1, 'The other band link contains the band name');
  });
});
```

We reasonably expect that this test will fail because we haven't created the bands that we assert are displayed. Since we decided to write acceptance tests and stub out http requests, let's resolve the above error by stubbing out this request.

Analyzing an Ember acceptance test

Let's stop for a moment to see how an acceptance test in Ember is built up.

The `test` function takes a name for the test that is displayed when the test is run. It also takes a function which wraps the test setup and the assertions. Its sole argument, `assert` is the object with all the built-in qunit assertion methods (`equal`, `ok`, etc.).

Our first test case is pretty simple: `visit` makes the router load the application at the given path, setting up all controllers and rendering templates just as if a user interacted with the real application. This should be the entry point for all tests.

`andThen` closes the test by going through all the things that need to be verified. `find` looks up a DOM element with a CSS selector, and `equal` is a QUnit helper that verifies whether the first two arguments are equal and fails if they are not. The three assertions above thus make sure that both bands are rendered with the correct CSS class and with the band names as their content.

When the test reruns it still fails, though. The XHR request to `/bands` is not stubbed out and thus the list of bands is not returned and displayed, so that should be the next step.

There are several libraries that allow stubbing out XHR requests, like jquery-mockjax, sinon.js or ic-ajax. We will use a new kid on the block, Pretender. It provides a very nice DSL to stub out the requests, similar to Sinatra or Express. It is simple to use, yet flexible enough to allow for simulating error responses, customizing unhandled request actions, and a lot more.

So let's install the Ember CLI addon:

```
$ ember install ember-cli-pretender
```

This adds the package to `package.json` and also runs the default blueprint provided with ember-cli-pretender. Let's restart the server since we have installed a new package.

This sets us (rock &) rolling.

To stub out a request in Pretender, we define the handlers (the stubs) inside a `new Pretender()` block:

```js
import Pretender from 'pretender';

new Pretender(function() {
  this.httpVerb(path, function(request) {
    return [statusCode, responseHeaders, data];
  });
});
```

(This is the general form of a Pretender handler, you don't have to write it anywhere. We'll add such handlers shortly.)

Being able to take a regex for the path and access the request object gives a lot of flexibility with regards to setting up the stub. Let's go ahead and add one for `/bands` to make our test pass.

Making the first test pass

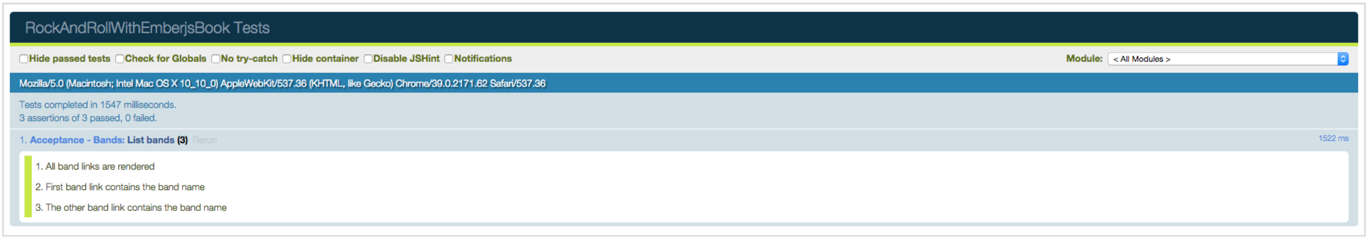
Popping open our acceptance test, we thus add the Pretender block in the test itself:

```js
  // tests/acceptance/bands-test.js
  import { test } from 'qunit';
  import moduleForAcceptance from '../helpers/module-for-acceptance';
+ import Pretender from 'pretender';

- moduleForAcceptance('Acceptance | bands');
+ moduleForAcceptance('Acceptance | bands', {
+    afterEach() {
+      server.shutdown();
+    }
+ });

+ var server;

  test('List bands', function(assert) {
+   server = new Pretender(function() {
+     this.get('/bands', function() {
+       var response = {
+         data: [
+           {
+             id: 1,
+             type: 'bands',
+             attributes: {
+               name: 'Radiohead'
+             }
+           },
+           {
+             id: 2,
+             type: 'bands',
+             attributes: {
+               name: 'Long Distance Calling'
+             }
+           },
+         ]
+       };
+       return [200, { 'Content-Type': 'application/vnd.api+json' }, JSON.stringify(response)];
+     });
+   });

visit('/bands');

andThen(function() {
assert.equal(find('.band-link').length, 2, 'All band links are rendered');
assert.equal(find('.band-link:contains("Radiohead")').length, 1, 'First band link contains the band name');
assert.equal(find('.band-link:contains("Long Distance Calling")').length, 1, 'The other band link contains the band name');
});
  });
```

In the handler, we just take two bands, write their JSON API representation and send it back with a `200 OK` http status code indicating success (`application/vnd.api+json` is the Content-Type for JSON API).

Pretender works by modifying the original `window.XMLHttpRequest` object and capturing XHR requests. Once we are done with a test the original `XMLHttpRequest` has to be restored. This is done by `server.shutdown()` in the `afterEach` block.

The test now passes in its entirety:



```
RockAndRollWithEmberjsBook Tests
☐Hide passed tests ☐Check for Globals ☐No try-catch ☐Hide container ☐Disable JSHint ☐Notifications        Module: < All Modules >
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.62 Safari/537.36
Tests completed in 1547 milliseconds.
3 assertions of 3 passed, 0 failed.
1. Acceptance - Bands: List bands (3) Rerun                                                         1522 ms
  1. All band links are rendered
  2. First band link contains the band name
  3. The other band link contains the band name
```

Safeguarding critical user scenarios

Creating a band

We now have the main elements for writing acceptance tests in place, so let's add a couple more of them to prevent features from breaking in the future.

The first should test creating a band:

```js
// tests/acceptance/bands-test.js
test('Create a new band', function(assert) {
  server = new Pretender(function() {
    this.get('/bands', function() {
      var response = {
        data: [
          {
            id: 1,
            type: 'bands',
            attributes: {
              name: 'Radiohead'
            }
          }
        ]
      };
      return [200, { 'Content-Type': 'application/vnd.api+json' }, JSON.stringify(response)];
    });

    this.post('/bands', function() {
      var response = {
        data: {
          id: 2,
          type: 'bands',
          attributes: {
            name: 'Long Distance Calling'
          }
        }

      };
      return [200, { 'Content-Type': 'application/vnd.api+json' }, JSON.stringify(response)];
    });

    this.get('/bands/2/songs', function() {
      var response = {
        data: []
```

JS

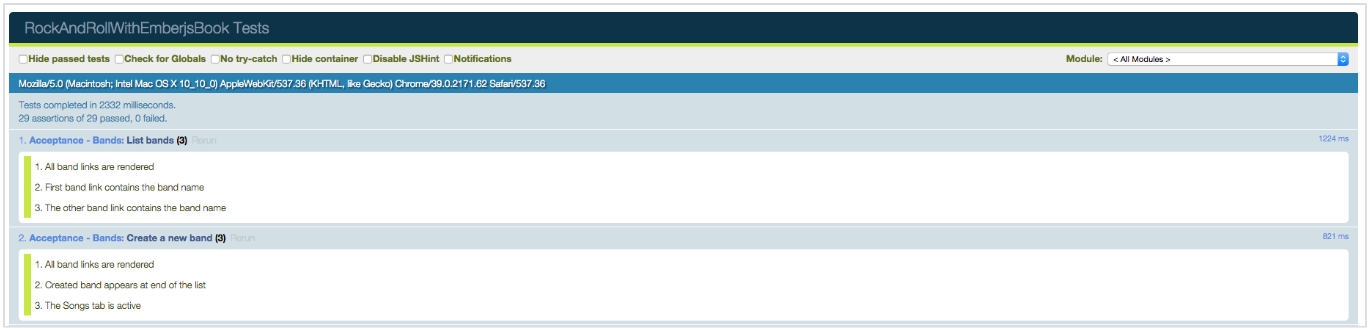
```
      };
      return [200, { 'Content-Type': 'application/vnd.api+json' }, JSON.stringify(response)];
    });
  });

  visit('/bands');
  fillIn('.new-band', 'Long Distance Calling');
  click('.new-band-button');

  andThen(function() {
    assert.equal(find('.band-link').length, 2, 'All band links are rendered');
    assert.equal(find('.band-link:last').text().trim(), 'Long Distance Calling', 'Created band appears at the end of the list');
    assert.equal(find('.nav a.active:contains("Songs")').length, 1, 'The Songs tab is active');
  });
});
```

The second request handler in the Pretender block makes sure to return the right response for creating a band (POST to `/bands`).

There are also two new test helpers that help setting up the test. `fillIn` sets the value of a text box targeted by a CSS selector that is specified in the first argument. `click` simulates a click event on the element that is again targeted by a CSS selector.

What we want to verify is whether the new band appears in the band listing (1st assertion in the `andThen` block), with the correct text (2nd assertion), and whether we transitioned to the Songs for the new band so that songs can be added right away (3rd and final assertion).

Since we stubbed out the POST request correctly, the new test passes, too:



```
RockAndRollWithEmberjsBook Tests
☐Hide passed tests ☐Check for Globals ☐No try-catch ☐Hide container ☐Disable JSHint ☐Notifications        Module: < All Modules >
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.62 Safari/537.36
Tests completed in 2332 milliseconds.
29 assertions of 29 passed, 0 failed.
1. Acceptance - Bands: List bands (3) Rerun                                                     1224 ms
  1. All band links are rendered
  2. First band link contains the band name
  3. The other band link contains the band name
2. Acceptance - Bands: Create a new band (3) Rerun                                              821 ms
  1. All band links are rendered
  2. Created band appears at end of the list
  3. The Songs tab is active
```

Creating a song

Let's close this section by adding a test for creating a song. Remember that for a band that does not have a song yet, creating a song is a two-step process. After selecting the band, first the "Why don't you create one?" link has to be clicked, and only then can the new song be added.

```
test('Create a new song in two steps', function(assert) {
  server = new Pretender(function() {
    this.get('/bands', function() {
      var response = {
        data: [
          {
            id: 1,
            type: 'bands',
            attributes: {
              name: 'Radiohead'
            }
          }
        ]
      };
      return [200, { 'Content-Type': 'application/vnd.api+json' }, JSON.stringify(response)];
    });
    this.get('/bands/1', function() {
      var response = {
        data: {
          id: 1,
          type: 'bands',
          attributes: {
            name: 'Radiohead'
          }
        }
      };
      return [200, { 'Content-Type': 'application/vnd.api+json' }, JSON.stringify(response)];
    });
    this.post('/songs', function() {
      var response = {
        data: {
          id: 1,
          type: 'songs',
          attributes: {
            name: 'Killer Cars'
```

JS

```
          }
        }
      };
      return [200, { 'Content-Type': 'application/vnd.api+json' }, JSON.stringify(response)];
    });

    this.get('/bands/1/songs', () => {
      return [200, { 'Content-Type': 'application/vnd.api+json' }, JSON.stringify({ data: [] })];
    });
  });

  visit('/');
  click('.band-link:contains("Radiohead")');
  click('a:contains("create one")');
  fillIn('.new-song', 'Killer Cars');
  triggerEvent('.new-song-form', 'submit');

  andThen(function() {
    assert.equal(find('.songs .song:contains("Killer Cars")').length, 1, "Creates the song and displays it in the list");
  });
});
```

The shape of this test is very similar to the one that verifies creating a band.

The only real difference is that instead of clicking the Add button at the end, we chose to trigger the submit event of the form. That is made possible by the `triggerEvent` helper:

```
triggerEvent(selector, eventName)
```

One thing you might have noticed is that we stub out three requests. Above the one that fetches all the bands and the one that responds to the song creation request, there is also one that returns data for an individual band (`/bands/1`). The reason for this is that when the link containing Radiohead is clicked, the corresponding `band` record is fetched as a result of `this.store.findRecord('band', params.id)`. Although the record is already in the store, Ember Data will refetch it in the background. If we did not stub out this request, the test would fail.

We now have a test for each of the main features of our application: listing bands, creating a new one, and adding a song to a band.

Writing your own test helpers

Ember makes it really easy to add your own helpers, both synchronous and asynchronous.

One reason to do this might be to add higher-level helpers that automate a whole process, for example, logging in. Another reason could be to make assertions more concise and descriptive, and decrease duplication in these assertions.

Registering a synchronous helper

A synchronous helper does not need to deal with asynchronicity. Since most things in the browser (requests, user events, etc.) happen in an asynchronous fashion, synchronous helpers are mostly used for bundling assertions or making them more concise and descriptive.

Our assertions are quite noisy, so let's add a few methods to clean them up.

Let's create a file under `tests/helpers` that stores these assertion helpers:

```js
// tests/helpers/asserts.js
import { registerHelper } from '@ember/test';

function assertTrimmedText(app, assert, selector, text, errorMessage) {
  var element = findWithAssert(selector);
  var elementText = element.text().trim();
  assert.equal(elementText, text, errorMessage);
}

function assertLength(app, assert, selector, length, errorMessage) {
  assert.equal(find(selector).length, length, errorMessage);
}

function assertElement(app, assert, selector, errorMessage) {
  assert.equal(find(selector).length, 1, errorMessage);
}

registerHelper('assertTrimmedText', assertTrimmedText);
registerHelper('assertLength', assertLength);
registerHelper('assertElement', assertElement);
```
JS

It is important that the `registerHelper` calls run before the test helpers are injected into the app, otherwise they will not be available in tests. Since test helpers are injected in `tests/helpers/start-app.js`, that's where we import the file that contains our new helpers:

```js
// tests/helpers/start-app.js
import Application from '../../app';
import config from '../../config/environment';
import { merge } from '@ember/polyfills';
import { run } from '@ember/runloop';
import './asserts';

export default function startApp(attrs) {
  let attributes = merge({}, config.APP);
  attributes = merge(attributes, attrs); // use defaults, but you can override;

  return run(() => {
    let application = Application.create(attributes);
    application.setupForTesting();
    application.injectTestHelpers();
    return application;
  });
}
```

The function exported by the `start-app.js` module returns a new Application instance that has been set up for testing. Since the helpers are imported right at the start, they are going to be defined, and thus usable in our tests. We can now clean up our assertions by using our newly-defined helpers:

```
 test('List bands', function(assert) {
   (...)
   andThen(function() {
-    assert.equal(find('.band-link').length, 2, 'All band links are rendered');
-    assert.equal(find('.band-link:contains("Radiohead")').length, 1, 'First band link contains the band name');
-    assert.equal(find('.band-link:contains("Long Distance Calling")').length, 1, 'The other band link contains the band name');
+    assertLength(assert, '.band-link', 2, 'All band links are rendered');
+    assertLength(assert, '.band-link:contains("Radiohead")', 1, 'First band link contains the band name');
+    assertLength(assert, '.band-link:contains("Long Distance Calling")', 1, 'The other band link contains the band name');
   });
 });

 test('Create a new band', function(assert) {
   (...)
   andThen(function() {
-    assert.equal(find('.band-link').length, 2, 'All band links are rendered');
-    assert.equal(find('.band-link:last').text().trim(), 'Long Distance Calling', 'Created band appears at the end of the list');
-    assert.equal(find('.nav a.active:contains("Songs")').length, 1, 'The Songs tab is active');
+    assertLength(assert, '.band-link', 2, 'All band links are rendered');
+    assertTrimmedText(assert, '.band-link:last', 'Long Distance Calling', 'Created band appears at the end of the list');
+    assertElement(assert, '.nav a.active:contains("Songs")', 'The Songs tab is active');
   });
 });

 test('Create a new song in two steps', function(assert) {
   (...)
   andThen(function() {
-   assert.equal(find('.songs .song:contains("Killer Cars")').length,
 1, "Creates the song and displays it in the list");
+   assertElement(assert, '.songs .song:contains("Killer Cars")',
 'Creates the song and displays it in the list');
   });
 });
```

To stop eslint complaining about our helpers not being defined, let's add them to `tests/.eslintrc.js`:

```js
// tests/.eslintrc.js
module.exports = {
  env: {
    embertest: true
  },
+ globals: {
+   assertTrimmedText: false,
+   assertLength: false,
+   assertElement: false,
+ }
};
```

Modifying the `.eslintrc.js` file needs an `ember serve` restart to take effect and make the warnings go away.

Registering an async helper

An async helper will wait for all asynchronous requests to finish. This makes the tests bullet-proof, since the test only goes on to the next step if everything has settled and been rendered. This prevents false negatives in tests since assertions that verify if a certain element has been rendered (or that its content is what we expect) only run after asynchronous operations have run to completion and updated DOM content.

The provided helpers we used all fall in this category: `visit`, `fillIn`, `click` and `triggerEvent`.

We'll create such a helper for visiting the root of our application and selecting a band by its name. Let's put these async helpers into their own test support file:

```js
// tests/helpers/async-helpers.js
import { registerAsyncHelper } from '@ember/test';

function selectBand(app, name) {
  visit('/')
  .click('.band-link:contains("' + name + '")');

  return app.testHelpers.wait();
}

registerAsyncHelper('selectBand', selectBand);
```

Let's also extract the `triggerEvent('.new-song-form', 'submit')` into its own helper, to increase the expressiveness of our test:

```js
// tests/helpers/async-helpers.js
import { registerAsyncHelper } from '@ember/test';

function selectBand(app, name) {
  visit('/')
  .click('.band-link:contains("' + name + '")');

  return app.testHelpers.wait();
}

+ function submit(app, selector) {
+   return triggerEvent(selector, 'submit');
+ }

registerAsyncHelper('selectBand', selectBand);
+ registerAsyncHelper('submit', submit);
```

Returning the result of calling the `wait` helper is paramount. This is what guarantees that all async operations have finished and we can proceed to the next step.

Just as with sync helpers, these need to be imported before they are injected into the application. This is done with the `import '../helpers/async-helpers'` line in the `start-app.js` file:

```js
// tests/helpers/start-app.js
import Application from '../../app';
import config from '../../config/environment';
import { merge } from '@ember/polyfills';
import { run } from '@ember/runloop';
import './asserts';
+ import './async-helpers';

export default function startApp(attrs) {
  (...)
}
```

Using this helper, we can rewrite our test for creating a new song:

```
// tests/acceptance/bands-test.js
test('Create a new song in two steps', function(assert) {
  server = new Pretender(function() {
    this.get('/bands', function() {
      var response = {
        data: [
          {
            id: 1,
            type: 'bands',
            attributes: {
              name: 'Radiohead'
            }
          }
        ]
      };
      return [200, { 'Content-Type': 'application/vnd.api+json' },
JSON.stringify(response)];
    });
    this.get('/bands/1', function() {
      var response = {
        data: {
          id: 1,
          type: 'bands',
          attributes: {
            name: 'Radiohead'
          }
        }
      };
      return [200, { 'Content-Type': 'application/vnd.api+json' },
JSON.stringify(response)];
    });
    this.post('/songs', function() {
      var response = {
        data: {
          id: 1,
          type: 'songs',
          attributes: {
```
JS

```
              name: 'Killer Cars'
            }
          }
        };
        return [200, { 'Content-Type': 'application/vnd.api+json' }, JSON.stringify(response)];
      });

      this.get('/bands/1/songs', () => {
        return [200, { 'Content-Type': 'application/vnd.api+json' }, JSON.stringify({ data: [] })];
      });
    });

-   visit('/');
-   click('.band-link:contains("Radiohead")');
+   selectBand('Radiohead');
    click('a:contains("create one")');
    fillIn('.new-song', 'Killer Cars');
-   triggerEvent('.new-song-form', 'submit');
+   submit('.new-song-form');

    andThen(function() {
      assertElement(assert, '.songs .song:contains("Killer Cars")', 'Creates the song and displays it in the list');
    });
  });
```

As the last step, let's add our new test helpers, `selectBand` and `submit` to the `tests/.eslintrc.js` file to keep eslint happy. Don't forget to relaunch the `ember serve` process in order for the linter to stop complaining.

```
// tests/.eslintrc.js
module.exports = {
  env: {
    embertest: true
  },
  globals: {
    assertTrimmedText: false,
    assertLength: false,
    assertElement: false,
+   selectBand: false,
+   submit: false
  }
};
```

Refactoring stubs

Let's finish the section about acceptance tests by cleaning up the stub definitions. We're not going to use any Ember features, just good old-fashioned refactoring.

There's a lot of repetition in setting up the stubs. We always return a successful json response, but more importantly the format of the returned data is repeated across multiple tests. This is error-prone, and if the format changes, it will have to be modified in each of these tests. Moreover, setting up the stubbed data is very noisy:

```
test('Create a new band', function(assert) {
  server = new Pretender(function() {
    this.get('/bands', function() {
      var response = {
        data: [
          {
            id: 1,
            type: 'bands',
            attributes: {
              name: 'Radiohead'
            }
          }
        ]
      };
      return [200, { 'Content-Type': 'application/vnd.api+json' },
JSON.stringify(response)];
    });

    this.post('/bands', function() {
      var response = {
        data: {
          id: 2,
          type: 'bands',
          attributes: {
            name: 'Long Distance Calling'
          }
        }
      };
      return [200, { 'Content-Type': 'application/vnd.api+json' },
JSON.stringify(response)];
    });

    this.get('/bands/2/songs', function() {
      var response = {
        data: []
      };
      return [200, { 'Content-Type': 'application/vnd.api+json' },
```

JS

```
JSON.stringify(response)];
    });
  });

  (...)
});
```

A straightforward way to make the test setup more robust and less verbose is to just pass the test object (or objects) that needs to be returned for each endpoint. So let's create a file under `tests/helpers` that we'll import into our test file:

```
// tests/helpers/http-stubs.js
function songsUrlForBand(id) {
  return '/bands/' + id + '/songs';
}

function responseItemForBand(data, id) {
  var bandId = id || data.id;
  return {
    id: bandId,
    type: 'bands',
    attributes: data.attributes,
    relationships: {
      songs: {
        links: {
          related: songsUrlForBand(bandId)
        }
      }
    }
  };
}

function responseItemForSong(data, id) {
  var songId = id || data.id;
  return {
    id: songId,
    type: "songs",
    attributes: data.attributes,
  };
}

export default {
  stubBands: function(pretender, data) {
    let responseForBands = [];
    data.forEach(function(band) {
      let responseForBand = responseItemForBand(band);
      pretender.get('/bands/' + responseForBand.id, function() {
        return [200, {'Content-Type': 'application/vnd.api+json'},
JSON.stringify({ data: responseForBand }) ];
      });
      responseForBands.push(responseForBand);
    });
    pretender.get('/bands', function() {
      return [200, {'Content-Type': 'application/vnd.api+json'},
JSON.stringify({ data: responseForBands }) ];
    });
  },

  stubSongs: function(pretender, bandId, data) {
    var response = data.map(function(song) {
      return responseItemForSong(song);
    });
    pretender.get(songsUrlForBand(bandId), function() {
      return [200, {'Content-Type': 'application/vnd.api+json'},
JSON.stringify({ data: response }) ];
    });
  },

  stubCreateBand: function(pretender, newId) {
    pretender.post('/bands', function(request) {
      var response =
responseItemForBand(JSON.parse(request.requestBody).data, newId);
      return [200, {'Content-Type': 'application/vnd.api+json'},
JSON.stringify({ data: response }) ];
    });
  },

  stubCreateSong: function(pretender, newId) {
    pretender.post('/songs', function(request) {
      var response =
responseItemForSong(JSON.parse(request.requestBody).data, newId);
      return [200, {'Content-Type': 'application/vnd.api+json'},
JSON.stringify({ data: response }) ];
    });
  },
};
```

The `responseItemForSong` and `responseItemForBand` functions take an id so that they can be used both when the band id is part of the request (typically GET requests) and when it needs to be created on the "server-side" (when creating resources).

A trick I employ is that I use the passed list of bands in `stubBands` to also set up the stubs for the individual band responses (`/bands/1`, `/bands/2`, etc.). This makes it unnecessary to stub out responses for individual band requests from the tests as it happens as part of `stubBands`.

All that's left is calling these functions from the test cases. I'll show how the test for creating a band has changed, and leave the others as an exercise for the reader:

```
  // tests/acceptance/bands-test.js
  import { test } from 'qunit';
  import moduleForAcceptance from '../helpers/module-for-acceptance';
  import Pretender from 'pretender';
+ import httpStubs from '../helpers/http-stubs';
  (...)

  test('Create a new band', function(assert) {
    server = new Pretender(function() {
-     this.get('/bands', function() {
-       var response = {
-         data: [
-           {
-             id: 1,
-             type: 'bands',
-             attributes: {
-               name: 'Radiohead'
-             }
-           }
-         ]
-       };
-       return [200, { 'Content-Type': 'application/vnd.api+json' },
  JSON.stringify(response)];
-     });
-
-     this.post('/bands', function() {
-       var response = {
-         data: {
-             id: 2,
-             type: 'bands',
-             attributes: {
-               name: 'Long Distance Calling'
-             }
-           }
-       };
-       return [200, { 'Content-Type': 'application/vnd.api+json' },
  JSON.stringify(response)];
-     });
-     this.get('/bands/2/songs', function() {
-       var response = {
-         data: []
-       };
-       return [200, { 'Content-Type': 'application/vnd.api+json' }, JSON.stringify(response)];
-     });
+     httpStubs.stubBands(this, [
+       {
+         id: 1,
+         attributes: {
+           name: 'Radiohead'
+         }
+       }
+     ]);
+     httpStubs.stubCreateBand(this, 2);
+     httpStubs.stubSongs(this, 2, []);
    });

    visit('/bands');
    fillIn('.new-band', 'Long Distance Calling');
    click('.new-band-button');

    andThen(function() {
      assertLength(assert, '.band-link', 2, 'All band links are rendered');
      assertTrimmedText(assert, '.band-link:last', 'Long Distance Calling', 'Created band appears at end of the list');
      assertElement(assert, '.nav a.active:contains("Songs")', 'The Songs tab is active');
    });
  });
```

Integration tests in Ember

Components in Ember are tested via integration tests. Let's type the following command:

```
$ ember generate component-test star-rating
```

Ember CLI has generated the following test stub for us:

```js
// tests/integration/components/star-rating-test.js
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';

moduleForComponent('star-rating', 'Integration | Component | star rating', {
  integration: true
});

test('it renders', function(assert) {
  // Set any properties with this.set('myProperty', 'value');
  // Handle any actions with this.on('myAction', function(val) { ... });

  this.render(hbs`{{star-rating}}`);

  assert.equal(this.$().text().trim(), '');

  // Template block usage:
  this.render(hbs`{{#star-rating}}template block text{{/star-rating}}`);

  assert.equal(this.$().text().trim(), 'template block text');
});
```

We'll replace the test case with something more meaningful for our component:

```js
// tests/integration/components/star-rating-test.js
import EmberObject from '@ember/object';
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';

moduleForComponent('star-rating', 'Integration | Component | star-rating', {
  integration: true
});

test('Renders the full and empty stars correctly', function(assert) {
  assert.expect(6);

  var song = EmberObject.create({ rating: 4 });
  this.set('song', song);
  this.set('maxRating', 5);

  this.render(hbs`{{star-rating item=song rating=song.rating maxRating=maxRating}}`);

  assert.equal(this.$('.glyphicon-star').length, 4, 'The right amount of full stars is rendered');
  assert.equal(this.$('.glyphicon-star-empty').length, 1, 'The right amount of empty stars is rendered');

  this.set('maxRating', 10);

  assert.equal(this.$('.glyphicon-star').length, 4, 'The right amount of full stars is rendered after changing maxRating');
  assert.equal(this.$('.glyphicon-star-empty').length, 6, 'The right amount of empty stars is rendered after changing maxRating');

  this.set('song.rating', 2);
  assert.equal(this.$('.glyphicon-star').length, 2, 'The right amount of full stars is rendered after changing rating');
  assert.equal(this.$('.glyphicon-star-empty').length, 8, 'The right
```
JS

```
amount of empty stars is rendered after changing rating');
});
```

The great thing about an Ember integration test is that it makes it very convenient to test components the same way as they are used in our application. `this` inside the test refers to the context of the component, so we can set values that are passed to the component via `this.set` (see `this.set('song', song)` above). Furthermore, the component can be rendered via a template invocation, courtesy of the `hbs` template string. `this.$()` refers to the component's top-level tag, so writing assertions stays very straightforward.

Unseen in this example is the advantage that if the component depends on other modules in our application, they do not need to be specified through listing them in the `needs` array (as it is the case with unit tests). All components in the hbs template string in the test will be instantiated and rendered as if we were in the application (with the exception of the `link-to` helper which needs a running router service).

Unit tests in Ember

To see an example of a unit test in Ember, we'll turn to controllers since Ember CLI creates unit tests for them:

```
$ ember generate controller-test bands/band/songs
```

We want to verify whether the wiring up of the `canCreateSong` is according to the spec: it is true either if the user has started the song creation process explicitly (before the band has any songs) or if the band has some songs.

While with acceptance and integration tests our assertions related to what can be observed "from the outside" (CSS selectors matching a certain number of elements on the page or an element with a particular value), here we'll inspect the internals of the unit under test (UUT), the controller. In concrete terms, we'll set the `songCreationStarted` property of the controller and the songs of the model and assert on the value of the `canCreateSong` property.

```js
// tests/unit/controllers/bands/band/songs-test.js
import EmberObject from '@ember/object';
import { moduleFor, test } from 'ember-qunit';

moduleFor('controller:bands/band/songs', 'Unit | Controller | bands/band/songs', {
});

test('canCreateSong', function(assert) {
  assert.expect(3);

  var controller = this.subject();
  var band = EmberObject.create();
  controller.set('model', band);

  controller.set('songCreationStarted', false);

  assert.ok(!controller.get('canCreateSong'), "Can't create song if process has not started and no songs yet");

  controller.set('songCreationStarted', true);

  assert.ok(controller.get('canCreateSong'), 'Can create song if process has started');

  controller.set('songCreationStarted', false);
  var songs = [
    EmberObject.create({ id: 1, title: 'Elephants', rating: 5 }),
  ];

  band.set('songs', songs);
  assert.ok(controller.get('canCreateSong'), 'Can create song if process has not started but there are already songs');
});
```

JS

ember-qunit tests have `this.subject()` automatically set to the UUT, so in this case, the `bands.band.songs` controller. We then create the mock of a Band object (just a simple EmberObject) and set it as the `model` of the controller. We verify that if the band has no songs, the value of `canCreateSong` follows that of `songCreationStarted` (see first two assertions).

Finally, we set `songCreationStarted` back to false and make sure that if the band already has some songs, creting a song is allowed (`canCreateSong` is true). The Song object we use is again a mock of the model object used in the real app. Mocking is permissible in unit tests as we are not testing these model classes and just expect these mocks to expose the same behavior (namely have the appropriate properties, like `title` and `rating`).

Next song

We can smugly lean back in our chairs, for we have achieved no less in this chapter than establishing a test suite that verifies that the main features of our application work correctly – with the caveats expressed in the "To mock or not to mock" section – and protects us from inadvertently breaking something when further working on the application. This is no small feat, and most projects would love to have this!

I'll hold myself to my word, and add the next feature with a test-first approach in the following chapter.

CHAPTER 12

Sorting and searching with query parameters

Tuning

Our application has a decent look, but there's no easy way to see the highest-ranked song for a band, or find a particular song in a long list.

The next feature to implement is thus sorting the list of songs and searching the list with text the user has provided. Let's also say we would like to share the sorted (and possibly filtered) lists with our friends. The easiest way to do this is to have the sorting option and search term be reflected in the URL. That brings us to another Ember feature, query parameters.

Query parameters

Query parameters (QPs, for short) are key-value pairs tacked onto the end of the URL. They're very common in server-side programming, and provide a simple way to either modify what needs to be shown on the page, or to just provide additional information that gets stored.

Pagination is an example of the first task. If you go to the programming subreddit and click the "next" button, the URL changes to this:

```
http://www.reddit.com/r/programming/?count=25&after=t3_2ou7md
```

Here, the `count` QP specifies how many posts should be shown on a page while the `after` QP tells the server from which post should these 25 ones be returned.

An example of the second use is the query parameters used in a Google custom campaign, which have the purpose of telling Google how each visitor arrived at the page. That information is extracted from the URL and sent to Google Analytics.

The link back to my blog in one of my guest posts had the following query parameters:

http://balinterdi.com/rock-and-roll-with-emberjs?utm*source=hackhands-ember-promises&utm*campaign=guest-blogging&utm_medium=referral

In most cases, this does not alter the page in any way, and so is an example of the second use for query parameters.

Query parameters in Ember

After a couple of rewrites, Ember settled on a query parameter implementation that covered all use cases and was robust enough to be integrated in the framework.

The query parameters need to be defined as properties on the controller. A binding is automatically set up between such a property and the corresponding query parameter in the URL.

To set up a QP called `search` on our controller, we write the following:

```js
export default Controller.extend({
  queryParams: ['search'],
  search: '',
})
```

The property that is bound to the QP has to be explicitly defined. Its value becomes the default value for the QP to ensure that if the property has that value, the QP does not appear in the URL. Otherwise, having an empty search term would produce an URL like: `/bands/1/songs?search=` , which is unnecessary and inelegant.

In some cases, there is a mismatch between the name we would like to use as a property, and the name of the QP in the URL. Ember supports this out of the box by allowing us to define the query parameters as an object instead of an array.

The search term appears almost universally as `s` (or `q`) in the URL, so let's name it that:

```js
export default Controller.extend({
  queryParams: {
    search: 's',
  },
  search: '',
})
```

This definition implies that the 'search' query parameter can be referred to as `search` on the controller, while it appears as `s` in the URL.

This is sufficient knowledge to implement searching and sorting songs, so let's get right to it.

New styles

There will be an awful lot of extra markup in this chapter, so let's add the necessary styling beforehand:

```css
/* app/styles/app.css */
(...)

.song-filter-search-panel {
  padding: 10px 15px;
  margin-bottom: -1px;
  background-color: #fff;
  border: 1px solid #ddd;
  display: block;
}

.song-filter-search-panel .active .sorting-button {
  z-index: 2;
  color: #fff;
  background-color: #428bca;
  border-color: #428bca;
}

.sorting-link:hover {
  text-decoration: none;
}

.song-filter-search-panel .sorting-link + .sorting-link {
  margin-left: 5px;
}

.sorting-panel {
  float: left;
}

@media (max-width: 690px) {
  .sorting-panel {
    margin-bottom: 5px;
  }
}

.search-panel {
  float: right;
}

.search-panel input {
  padding: 4px 12px 6px 12px;
}

.search-panel .input-group-btn {
  display: inline-block;
}

.search-panel .input-group-btn .search-button {
  margin-left: -5px;
}
```

Sorting the list of songs

As I promised at the end of the last chapter, we will implement this feature using a test-first approach. Let's write the test that verifies that the songs are sorted in the order selected by the user, and that the order is reflected in the URL:

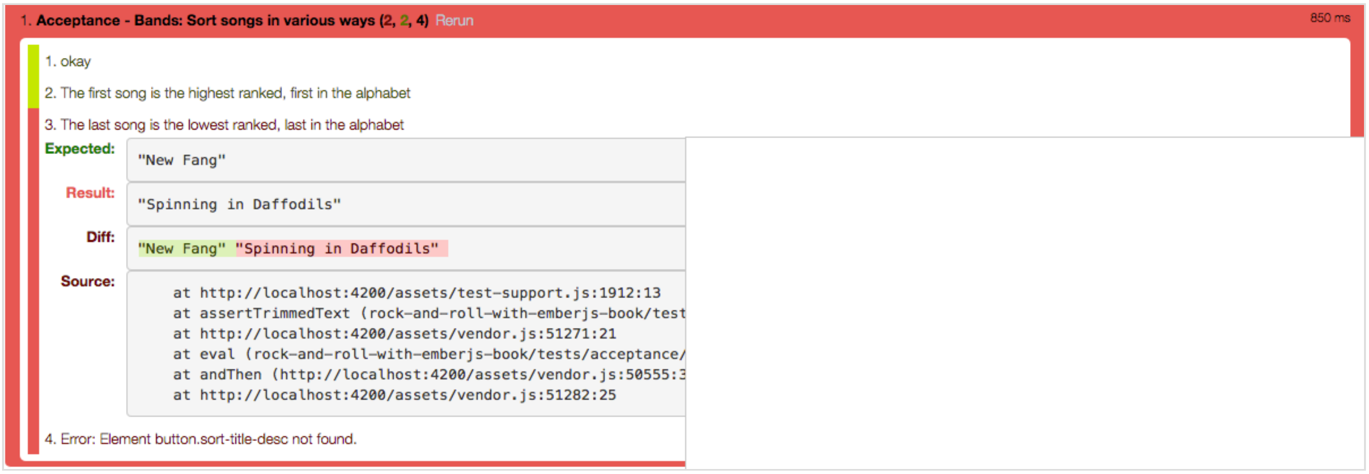
```js
// tests/acceptance/bands-test.js

test('Sort songs in various ways', function(assert) {
  server = new Pretender(function() {
    httpStubs.stubBands(this, [
      {
        id: 1,
        attributes: {
          name: 'Them Crooked Vultures',
        }
      }
    ]);
    httpStubs.stubSongs(this, 1, [
      {
        id: 1,
        attributes: {
          title: 'Elephants',
          rating: 5
        }
      },
      {
        id: 2,
        attributes: {
          title: 'New Fang',
          rating: 4
        }
      },
      {
        id: 3,
        attributes: {
          title: 'Mind Eraser, No Chaser',
          rating: 4
        }
      },
      {
        id: 4,
        attributes: {
          title: 'Spinning in Daffodils',
          rating: 5
        }
      }
    ]);
  });

  selectBand('Them Crooked Vultures');

  andThen(function() {
    assert.equal(currentURL(), '/bands/1/songs');
    assertTrimmedText(assert, '.song:first', 'Elephants', 'The first song is the highest ranked, first in the alphabet');
    assertTrimmedText(assert, '.song:last', 'New Fang', 'The last song is the lowest ranked, last in the alphabet');
  });

  click('button.sort-title-desc');

  andThen(function() {
    assert.equal(currentURL(), '/bands/1/songs?sort=titleDesc');
    assertTrimmedText(assert, '.song:first', 'Spinning in Daffodils', 'The first song is the one that is the last in the alphabet');
    assertTrimmedText(assert, '.song:last', 'Elephants', 'The last song is the one that is the first in the alphabet');
  });

  click('button.sort-rating-asc');

  andThen(function() {
    assert.equal(currentURL(), '/bands/1/songs?sort=ratingAsc');
    assertTrimmedText(assert, '.song:first', 'Mind Eraser, No Chaser', 'The first song is the lowest ranked, first in the alphabet');
    assertTrimmedText(assert, '.song:last', 'Spinning in Daffodils', 'The last song is the highest ranked, last in the alphabet');
  });
});
```

There are more song fixtures than usual in order to be able to assert the right ordering. The first block of assertions makes sure that by default (with no query parameters) the songs are sorted in descending order of their rating. In case of a tie, the song whose title comes first in the alphabet wins.

We assume that sorting happens by clicking on sorting buttons. The test assumes (at this point there is no implementation) that the one sorts the songs in descending order of their title has a `sort-title-desc` class.

The second and third block of assertions verify whether the list of songs has been sorted according to the clicked button.

When we go to `http://localhost:4200/tests` to run the tests, the new test case fails:



```
1. Acceptance - Bands: Sort songs in various ways (2, 2, 4) Rerun                850 ms
  1. okay
  2. The first song is the highest ranked, first in the alphabet
  3. The last song is the lowest ranked, last in the alphabet
  Expected: "New Fang"
  Result:   "Spinning in Daffodils"
  Diff:     "New Fang" "Spinning in Daffodils"
  Source:
        at http://localhost:4200/assets/test-support.js:1912:13
        at assertTrimmedText (rock-and-roll-with-emberjs-book/test
        at http://localhost:4200/assets/vendor.js:51271:21
        at eval (rock-and-roll-with-emberjs-book/tests/acceptance/
        at andThen (http://localhost:4200/assets/vendor.js:50555:3
        at http://localhost:4200/assets/vendor.js:51282:25
  4. Error: Element button.sort-title-desc not found.
```

The songs are not yet sorted, so this was expected. In fact, this is the very essence of test-driven development. We always have a failing test first, before we write the code that makes the test pass.

One of the great things about Ember's query parameter implementation is that we can implement a feature by using "normal" controller properties, and then bind them to query parameters once everything works. That will make the state that the query parameters describe persist in the URL so that it becomes very simple to reconstruct that state.

Sorting an array of items

We'll first do the sorting without worrying about query parameters. We have a list of songs in the controller's `model` property that needs to be sorted according to another property on the same controller. Let's call this latter property `sortBy`.

For the sorting, we'll leverage the built-in feature `sort`, which is a so-called 'computed property macro.' We haven't talked about them yet, so let's take a short break and look at what's happening backstage.

BACKSTAGE

COMPUTED PROPERTY MACROS

A computed property macro (CPM) is a computed property where the dependencies and the function body have been "precompiled." CPMs thus mean less repetition, more readable, and more robust code.

Imagine you are writing a to-do management app (to invent a random example) and you want to keep count of the incomplete to-dos.

Let's assume you have all the to-dos in the `model` property of the controller. You first have to filter out the ones whose `isCompleted` property is false, and then count them. You'd write something like this:

```js
export default Controller.extend({
  (...)
  pendingTodos: computed('model.@each.isComlpeted', function() {
    return this.get('model').filter(function(todo) {
      return todo.get('isCompleted') === false;
    });
  }),

  pendingTodosCount: computed('pendingTodos.length', function()
{
    return this.get('pendingTodos').length;
  }),
});
```

JS

See what I did there? By mistyping the dependent key of `pendingTodos` (I wrote `isComlpeted` instead of `isCompleted`) I risk tearing out a good chunk of my hair trying to figure out why the count does not update when a to-do is completed.

Ember computed macros are a great way for Ember developers to keep more of their hair and cut down on boilerplate code. Here is what it would look like with CPMs:

```js
import { filterBy, alias } from "@ember/object/computed";

export default Controller.extend({
  (...)
  pendingTodos: filterBy('model', 'isCompleted', false),
  pendingTodosCount: alias('pendingTodos.length'),
});
```

Mucho mejor, ¿no?

Ember has a large number of predefined CPMs for common computed property patterns, and you can also add your new ones.

Now, back to sorting. The computed macro for sorting is `sort`. You pass it the property (path) that holds the items to be sorted, and a property that describes the criterion for sorting.

In the case at hand, we want to first sort by rating (higher rating first) and then in alphabetic order of the titles, where the item whose title comes first in the alphabet has to come first in the array:

```js
  // app/controllers/bands/band/songs.js
+ import { sort } from '@ember/object/computed';

  export default Controller.extend({
    (...)
+   sortProperties: ['rating:desc', 'title:asc'],
+   sortedSongs: sort('model.songs', 'sortProperties'),
    (...)
  })
```

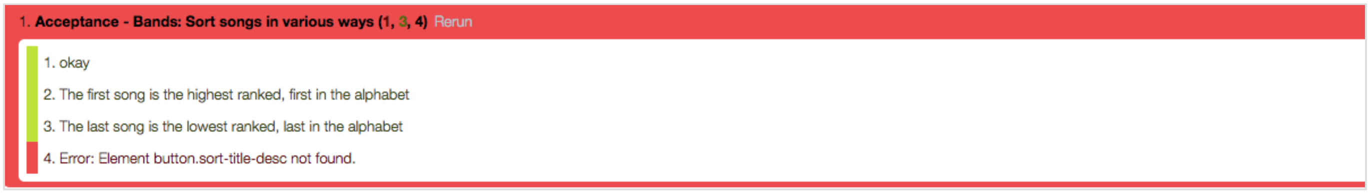
The general form of the above shorthand notation in `sortProperties` is `property:direction`. So `rating:desc` means sort the items by their `rating` property in descending order (highest rating first). `title:asc` instructs Ember to sort the items in ascending order of their `title`. The order in sortProperties matters: songs will first be ordered by their rating and then, if there is a tie, their title is used to break it.

Our template now only needs to iterate through sortedSongs instead of model.songs.

```hbs
  <!-- app/templates/bands/band/songs.hbs -->
  (...)
-   {{#each model.songs as |song|}}
+   {{#each sortedSongs as |song|}}
      <li class="list-group-item song">
        {{song.title}}
        {{star-rating item=song rating=song.rating on-click=(action "updateRating")}}
      </li>
    {{/each}}
  (...)
```

HBS

That makes our first test block, three assertions, pass!



```
1. Acceptance - Bands: Sort songs in various ways (1, 3, 4) Rerun
  1. okay
  2. The first song is the highest ranked, first in the alphabet
  3. The last song is the lowest ranked, last in the alphabet
  4. Error: Element button.sort-title-desc not found.
```

Changing the sorting criteria

The next step is to allow the user to change the sorting criteria by clicking a few buttons. This step does not involve new techniques. We make the buttons trigger a setSorting action with the appropriate sorting order. Next, we handle the action and change the sortProperties property. Since sortProperties is what changes the ordering of the items, there is nothing else to be done.

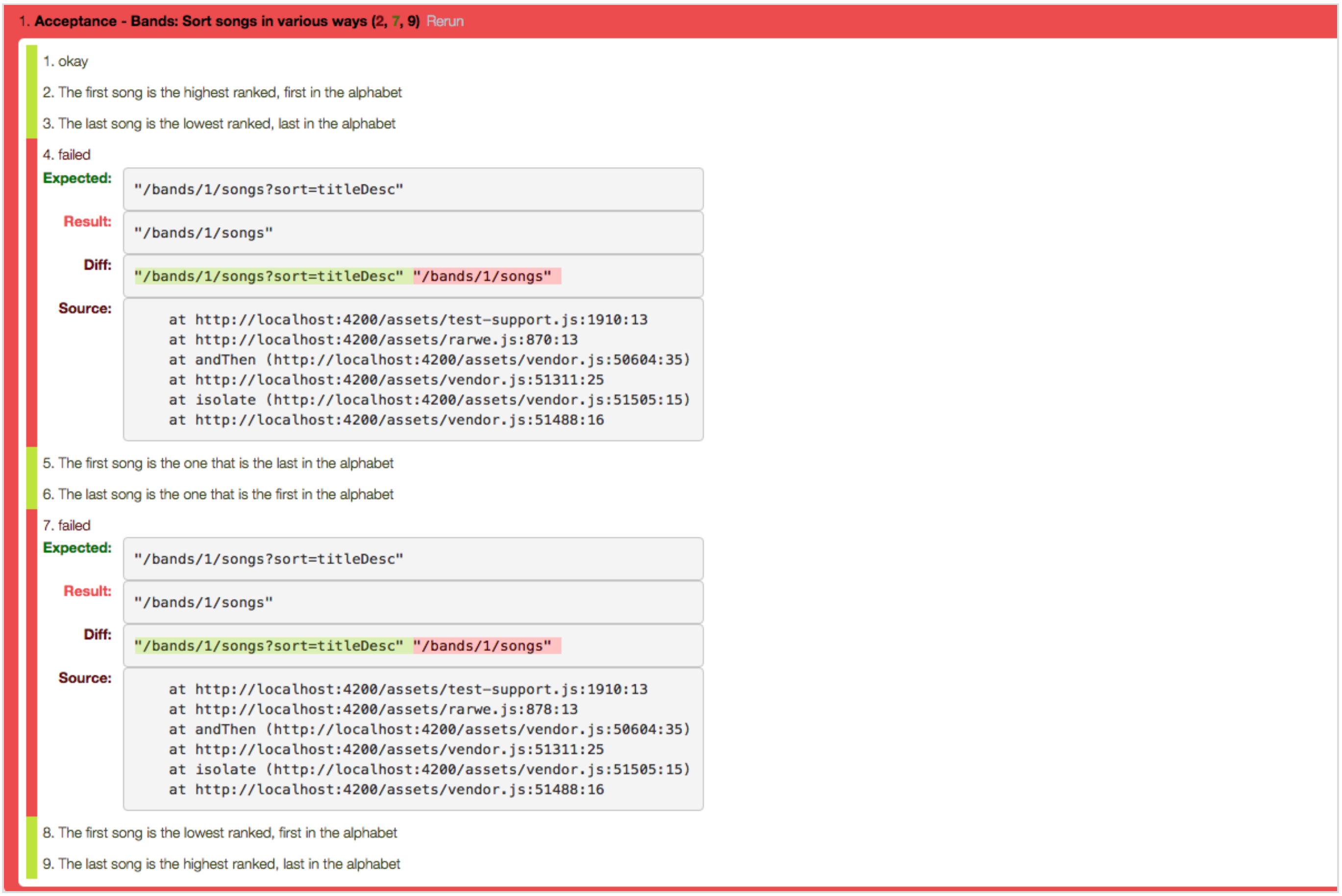
```
  <!-- app/templates/bands/band/songs.hbs -->
  <ul class="list-group songs">
    {{#if canCreateSong}}
      (...)
+     <li class="btn-group song-filter-search-panel" role="group">
+       <div class="sorting-panel">
+         <button type="button" class="btn btn-default sort-rating-desc sorting-button" {{action "setSorting" "ratingDesc"}}>
+           Rating
+           <span class="glyphicon glyphicon-arrow-down"></span>
+         </button>
+         <button type="button" class="btn btn-default sort-rating-asc sorting-button" {{action "setSorting" "ratingAsc"}}>
+           Rating
+           <span class="glyphicon glyphicon-arrow-up"></span>
+         </button>
+         <button type="button" class="btn btn-default sort-title-desc sorting-button" {{action "setSorting" "titleDesc"}}>
+           Title
+           <span class="glyphicon glyphicon-arrow-down"></span>
+         </button>
+         <button type="button" class="btn btn-default sort-title-asc sorting-button" {{action "setSorting" "titleAsc"}}>
+           Title
+           <span class="glyphicon glyphicon-arrow-up"></span>
+         </button>
+       </div>
+       <div class="clearfix"></div>
+     </li>
    {{/if}}
    {{#each sortedSongs as |song|}}
      <li class="list-group-item song">
        {{song.title}}
        {{star-rating item=song rating=song.rating on-click=(action "updateRating")}}
      </li>
    {{/each}}
```
HBS

```
  (...)
</ul>
```

The controller gains some new code to handle the action and to transform the sorting string (like "titleDesc") into the appropriate format for `sortProperties` (note that `sortProperties` needs to be overwritten from the previous code block):

```
  // app/controllers/bands/band/songs.js
  export default Controller.extend({
    (...)
-   sortProperties: ['rating:desc', 'title:asc'],
+   sortBy: 'ratingDesc',
+   sortProperties: computed('sortBy', function() {
+     var options = {
+       'ratingDesc': 'rating:desc,title:asc',
+       'ratingAsc': 'rating:asc,title:asc',
+       'titleDesc': 'title:desc',
+       'titleAsc': 'title:asc',
+     };
+     return options[this.get('sortBy')].split(',');
+   }),

    sortedSongs: sort('model.songs', 'sortProperties'),

    (...)
    actions: {
      (...)
+     setSorting: function(option) {
+       this.set('sortBy', option);
+     },
    }

  })
```

(Be sure not to leave spaces in the string values for the `options` variable above).


```
1. Acceptance - Bands: Sort songs in various ways (2, 7, 9) Rerun
1. okay
2. The first song is the highest ranked, first in the alphabet
3. The last song is the lowest ranked, last in the alphabet
4. failed
Expected: "/bands/1/songs?sort=titleDesc"
Result:   "/bands/1/songs"
Diff:     "/bands/1/songs?sort=titleDesc" "/bands/1/songs"
Source:
    at http://localhost:4200/assets/test-support.js:1910:13
    at http://localhost:4200/assets/rarwe.js:870:13
    at andThen (http://localhost:4200/assets/vendor.js:50604:35)
    at http://localhost:4200/assets/vendor.js:51311:25
    at isolate (http://localhost:4200/assets/vendor.js:51505:15)
    at http://localhost:4200/assets/vendor.js:51488:16
5. The first song is the one that is the last in the alphabet
6. The last song is the one that is the first in the alphabet
7. failed
Expected: "/bands/1/songs?sort=titleDesc"
Result:   "/bands/1/songs"
Diff:     "/bands/1/songs?sort=titleDesc" "/bands/1/songs"
Source:
    at http://localhost:4200/assets/test-support.js:1910:13
    at http://localhost:4200/assets/rarwe.js:878:13
    at andThen (http://localhost:4200/assets/vendor.js:50604:35)
    at http://localhost:4200/assets/vendor.js:51311:25
    at isolate (http://localhost:4200/assets/vendor.js:51505:15)
    at http://localhost:4200/assets/vendor.js:51488:16
8. The first song is the lowest ranked, first in the alphabet
9. The last song is the highest ranked, last in the alphabet
```

With that, all our tests pass except the ones which assert on the URL. This makes sense as we haven't yet set up the query parameters, so the URL does not change as we change the sorting criteria. We'll come back to this at the end of the chapter and now move on to searching.

Filtering songs by a search term

The second and final feature will allow the user to enter a text fragment and have the list contain only the songs that have this text in their title.

Let's stick to writing tests first in this chapter:

```js
// tests/acceptance/bands-test.js
test('Search songs', function(assert) {
  server = new Pretender(function() {
    httpStubs.stubBands(this, [
      {
        id: 1,
        attributes: {
          name: 'Them Crooked Vultures',
        }
      }
    ]);

    httpStubs.stubSongs(this, 1, [
      {
        id: 1,
        attributes: {
          title: 'Elephants',
          rating: 5
        }
      },
      {
        id: 2,
        attributes: {
          title: 'New Fang',
          rating: 4
        }
      },
      {
        id: 3,
        attributes: {
          title: 'Mind Eraser, No Chaser',
          rating: 4
        }
      },
      {
        id: 4,
        attributes: {
          title: 'Spinning in Daffodils',
          rating: 5
        }
      },
      {
        id: 5,
        attributes: {
          title: 'No One Loves Me & Neither Do I',
          rating: 5
        }
      }
    ]);
  });

  visit('/bands/1');
  fillIn('.search-field', 'no');

  andThen(function() {
    assertLength(assert, '.song', 2, 'The songs matching the search term are displayed');
  });

  click('button.sort-title-desc');
  andThen(function() {
    assertTrimmedText(assert, '.song:first', 'No One Loves Me & Neither Do I', 'A matching song that comes later in the alphabet appears on top');
    assertTrimmedText(assert, '.song:last', 'Mind Eraser, No Chaser', 'A matching song that comes sooner in the alphabet appears at the bottom ');
  });
});
```

The test prescribes a text field with a class of `search-field` to be present on the form. Writing in that field should trigger the search immediately. Furthermore, searching and sorting should be possible at the same time, and the consequent clicking of the sorting button and assertions make sure this is the case.

There needs to exist a property on the controller that tracks the value of the search field. Taking that value, the list of songs needs to be narrowed down to the ones whose title have it as a substring (allowing for differences in case):

```js
  // app/controllers/bands/band/songs.js
  export default Controller.extend({
    (...)
+   searchTerm: '',
+
+   matchingSongs: computed('model.songs.@each.title', 'searchTerm', function() {
+     var searchTerm = this.get('searchTerm').toLowerCase();
+     return this.get('model.songs').filter(function(song) {
+       return song.get('title').toLowerCase().indexOf(searchTerm) !== -1;
+     });
+   }),
    (...)
  })
```

We have to make sure that sorting happens on the filtered songs. It would be wasteful (on larger lists) to first sort all the items, only to have most of them filtered out by the searched expression.

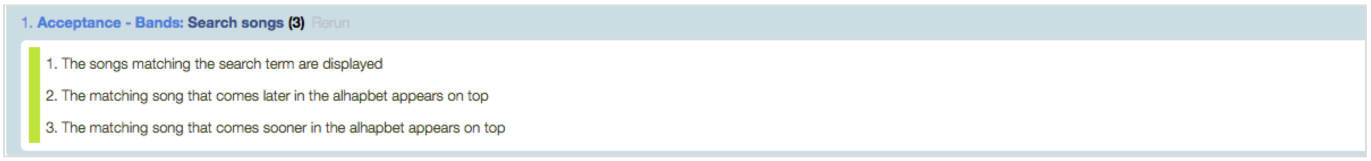
So let's modify the sorting to happen on `matchingSongs`:

```js
  // app/controllers/bands/band/songs.js
  export default Controller.extend({
    (...)
-   sortedSongs: sort('model.songs', 'sortProperties'),
+   sortedSongs: sort('matchingSongs', 'sortProperties'),
    (...)
  });
```

The only reason for the template to change is to incorporate the search panel:

```
  <!-- app/templates/bands/band/songs.hbs -->
  <ul class="list-group songs">
    {{#if canCreateSong}}
      (...)
      <div class="btn-group song-filter-search-panel" role="group">
        <div class="sorting-panel">
          (...)
        </div>
+       <div class="search-panel">
+         {{input type="text" placeholder="Start typing" value=searchTerm class="search-field"}}
+         <div class="input-group-btn">
+           <button class="btn btn-default search-button">
+             <span class="glyphicon glyphicon-search" aria-label="search" aria-hidden="true"></span>
+           </button>
+         </div>
+       </div>
        <div class="clearfix"></div>
      </div>
    {{/if}}
    (...)
    {{#each sortedSongs as |song|}}
      <li class="list-group-item song">
        {{song.title}}
        {{star-rating item=song rating=song.rating on-click=(action "updateRating")}}
      </li>
    {{/each}}
    (...)
  </ul>
```

With that, our "Search songs" test now passes:


1. Acceptance - Bands: Search songs (3) Rerun

1. The songs matching the search term are displayed
2. The matching song that comes later in the alhapbet appears on top
3. The matching song that comes sooner in the alhapbet appears on top

Adding a dash of query parameters

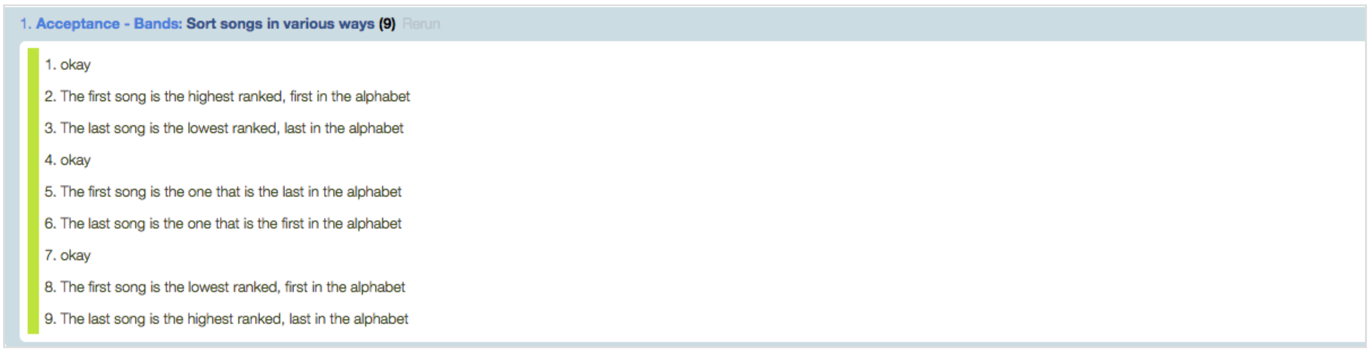
The final touch is to have the sorting order appear in the URL. Something like `/bands/1/songs?sort=titleAsc&s=sir` would do.

Surprising as it might seem, the only thing we have to do is define the query parameter mapping:

```js
// app/controllers/bands/band/songs.js
export default Controller.extend({
+  queryParams: {
+    sortBy: 'sort',
+    searchTerm: 's',
+  },
  songCreationStarted: false,
  sortBy: 'ratingDesc',
  searchTerm: '',
  (...)
```

That's it, three extra lines. One, if we favor a small LOC over readability. This is as easy as adding a dash of salt at the end of cooking a dish. Ember makes JavaScript oh so tasty.

Ladies and gentlemen, all our sorting related tests are now coming back green, including the ones that verify the query parameters:



1. Acceptance - Bands: Sort songs in various ways (9) Rerun

```
1. okay
2. The first song is the highest ranked, first in the alphabet
3. The last song is the lowest ranked, last in the alphabet
4. okay
5. The first song is the one that is the last in the alphabet
6. The last song is the one that is the first in the alphabet
7. okay
8. The first song is the lowest ranked, first in the alphabet
9. The last song is the highest ranked, last in the alphabet
```

Using links instead of buttons

We can improve user experience by making the buttons be links to the route with the appropriate sorting option. Since the sorting options and search term are now embedded in the URL, changing the sorting option is the same thing as transitioning to the same route with the `sort` query parameter changed.

In order to do that, only the template needs to change:

```
  {{!-- app/templates/bands/band/songs.hbs --}}
  <ul class="list-group songs">
    {{#if canCreateSong}}
      (...)
      <li class="btn-group song-filter-search-panel" role="group">
        <div class="sorting-panel">
-         <button type="button"
-           class="btn btn-default sort-rating-desc sorting-button"
-           {{action "setSorting" "ratingDesc"}}>
-           Rating
-           <span class="glyphicon glyphicon-arrow-down"></span>
-         </button>
-         <button type="button"
-           class="btn btn-default sort-rating-asc sorting-button"
-           {{action "setSorting" "ratingAsc"}}>
-           Rating
-           <span class="glyphicon glyphicon-arrow-up"></span>
-         </button>
-         <button type="button"
-           class="btn btn-default sort-title-desc sorting-button"
-           {{action "setSorting" "titleDesc"}}>
-           Title
-           <span class="glyphicon glyphicon-arrow-down"></span>
-         </button>
-         <button type="button"
-           class="btn btn-default sort-title-asc sorting-button"
-           {{action "setSorting" "titleAsc"}}>
-           Title
-           <span class="glyphicon glyphicon-arrow-up"></span>
-         </button>
+         {{#link-to (query-params sortBy="ratingDesc") class="sorting-link"}}
+           <button type="button" class="btn btn-default sort-rating-desc sorting-button">Rating
+             <span class="glyphicon glyphicon-arrow-down"></span>
+           </button>
+         {{/link-to}}
+         {{#link-to (query-params sortBy="ratingAsc") class="sorting-link"}}
+           <button type="button" class="btn btn-default sort-rating-asc sorting-button">Rating
+             <span class="glyphicon glyphicon-arrow-up"></span>
+           </button>
+         {{/link-to}}
+         {{#link-to (query-params sortBy="titleDesc") class="sorting-link"}}
+           <button type="button" class="btn btn-default sort-title-desc sorting-button">Title
+             <span class="glyphicon glyphicon-arrow-down"></span>
+           </button>
+         {{/link-to}}
+         {{#link-to (query-params sortBy="titleAsc") class="sorting-link"}}
+           <button type="button" class="btn btn-default sort-title-asc sorting-button">Title
+             <span class="glyphicon glyphicon-arrow-up"></span>
+           </button>
+         {{/link-to}}
      </div>
    </li>
    (...)
  {{/if}}
  (...)
</ul>
```

Note the relevant parts: the `link-to` helper can take a query-params "subexpression" (a helper call wrapped in parentheses) with key-value pairs for the query parameters.

Also note that I omitted the route name argument from the `link-to`-s. Instead of writing this:

```
{{#link-to 'bands.band.songs' (query-params sortBy="ratingDesc")
class="sorting-link"}}
```

, I wrote that:

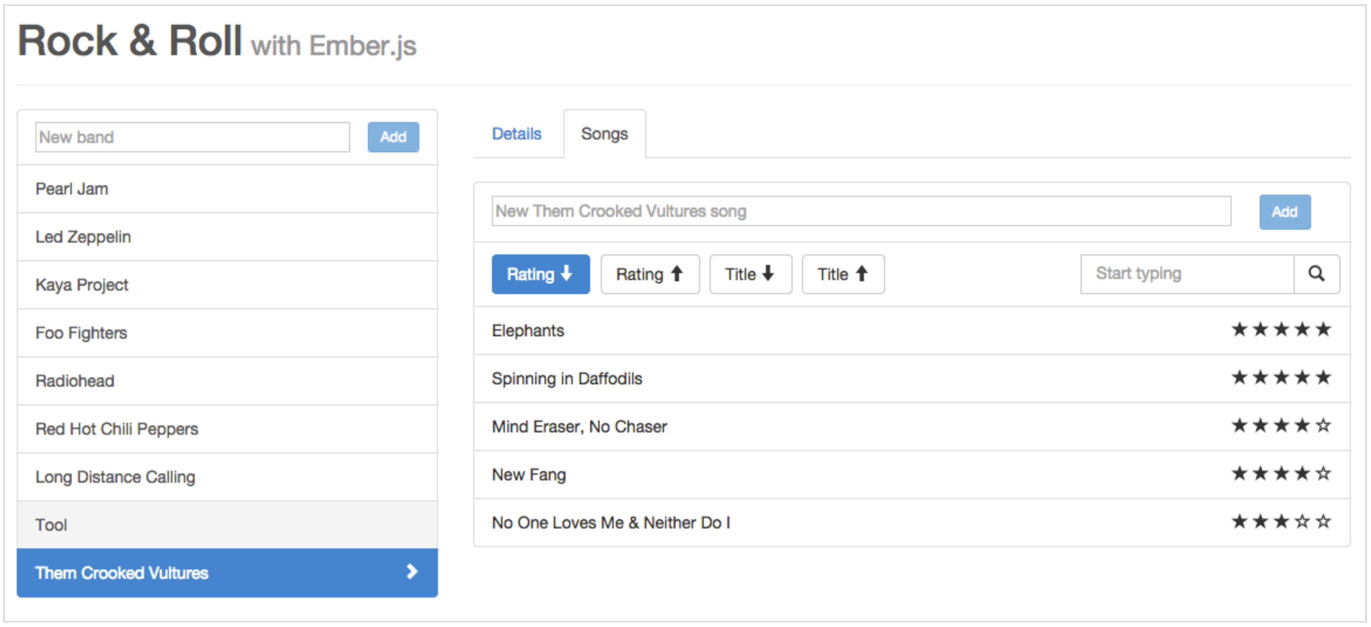
```
{{#link-to (query-params sortBy="ratingDesc") class="sorting-link"}}
```

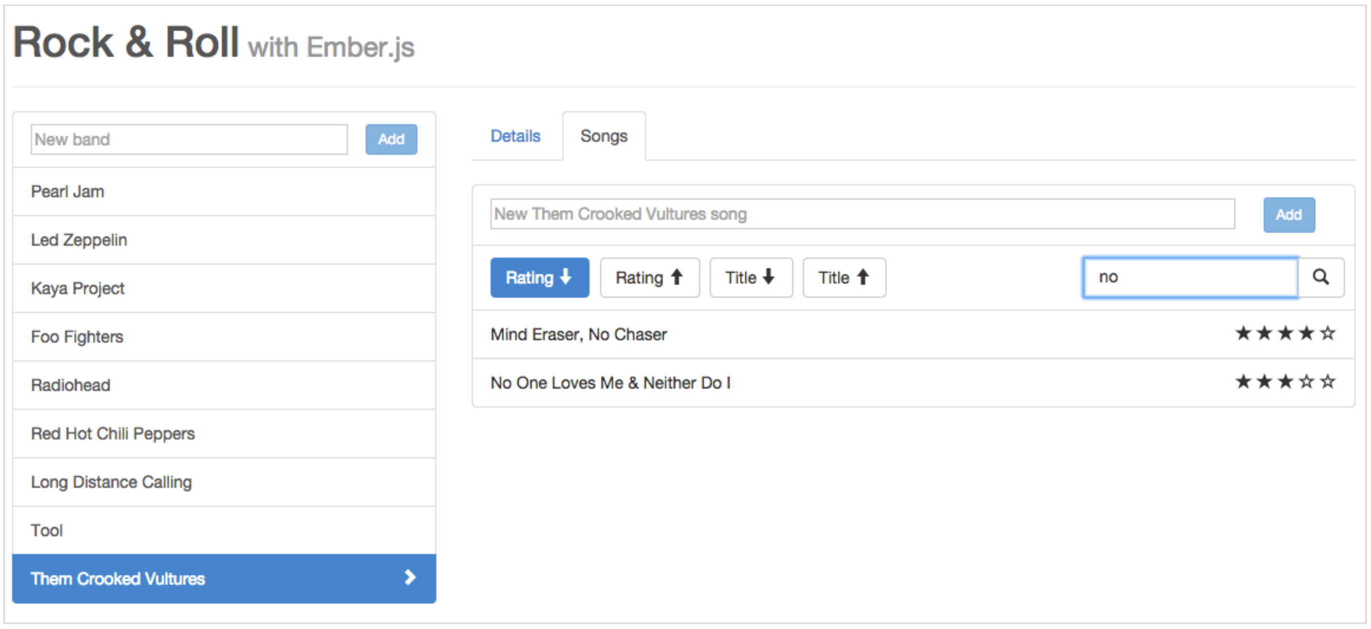
, since it is not necessary to pass the route name if the destination route is the current one.

Having the sorting implemented through links has the advantage that no actions need to be defined (so `setSorting` can be removed from the controller) and that when one hovers over the buttons, the URL is displayed just as with "ordinary" links.

Taking a look at what we made

The disadvantage of driving the implementation of the feature with testing is that we don't see the actual page in its full glory. So after we have come back from the kitchen with a chocolate bar (beer, bag of chips, to each their own reward) in hand, let's look at it and rejoice:





Mind Eraser, No Chaser ★★★★☆
No One Loves Me & Neither Do I ★★★☆☆

Next song

Cool, but what if the backend is down or just takes a lot of time to load our data? The user should have an idea about what is going on. This brings us to loading and error routes and templates.

CHAPTER 13

Loading and error routes

Tuning

When the app "boots up", a request is sent to the backend to fetch the needed data. That might take some time, during which the user of the application is not aware of what's going on.

With the request being asynchronous, the user doesn't even see a spinner in the browser tab, so it's all the more important to display something while data is being fetched.

Showing errors is similar to showing progress. When something goes wrong in an app rendered on the backend, signaling this to the user is relatively easy.

In the case of browser applications, the issue is more complex. An extensive error-handling strategy would handle errors depending on their severity and where they were produced. Here, we will focus on errors that are thrown during the router's transition to a destination route.

New styles

These new styles will give clothing to the loading and error panes implemented in this chapter:

```
/* app/styles/app.css */
(...)
.loading-pane {
  margin: 25px auto;
}

.loading-message {
  text-align: center;
  color: #aaa;
}

.error-pane {
  margin: 25px auto;
  background-color: rgba(255, 176, 176, 0.33);
}

.error-message {
  text-align: center;
  color: #e22626;
  line-height: 100px;
}

.spinner {
  background: url('/images/ajax-loader.gif') no-repeat;
  height: 31px;
  width: 31px;
  margin: 0 auto;
}
```

You can download the same spinner image I use from https://s3-eu-west-1.amazonaws.com/rarwe-book/ajax-loader.gif. After you have downloaded it, place it in the `public/images` folder of your application.

Loading routes

Let's assume, just as we did in the Nested routes chapter, that the router transitions from the top all the way to `band.songs`. For each route, it calls the route's `model` hook to resolve the data needed to render the template.

If the data returned in the model hook is not resolved immediately, Ember is designed to trigger a 'loading' event on the route that is being entered. By default, that loading event will render a template named `loading` at the same level as the route itself.

An example is worth a thousand words, so suppose we are entering the `bands` route, which has the following model hook:

```js
// app/routes/bands.js
export default Route.extend({
  model: function() {
    return this.store.findAll('band');
  },
});
```

JS

As we saw, that promise sends an ajax request to `/bands` to the API. If the data does not arrive right away, the loading event is fired on the `bands` route, which in turn renders a template at the same level as the `bands` route.

Let's create a top-level loading template:

```
$ ember generate template loading
```

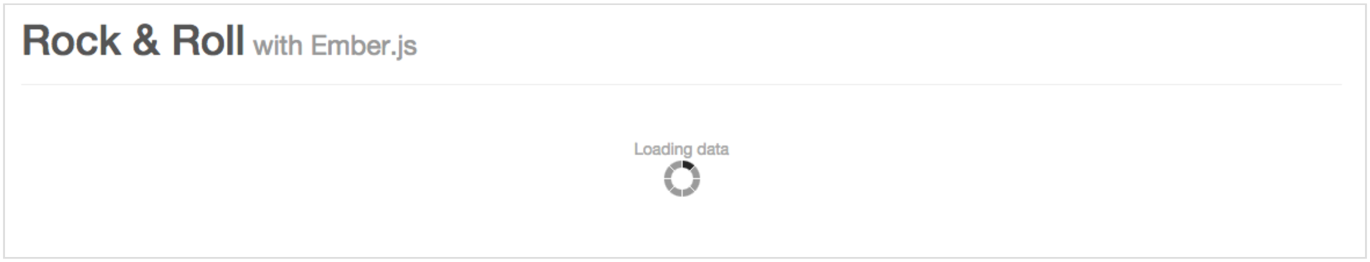
And add some "loading content" into it:

```
<!-- app/templates/loading.hbs -->
<div class="loading-pane">
  <div class="loading-message">
    Loading data
    <div class="spinner"></div>
  </div>
</div>
```

To simulate network latency, we can use the following function:

```
  // app/routes/bands.js
+ import { Promise as EmberPromise } from 'rsvp';

+ function wait(promise, delay) {
+   return new EmberPromise(function(resolve) {
+     setTimeout(function() {
+       promise.then(function(result) {
+         resolve(result);
+       });
+     }, delay);
+   });
+ }
  export default Route.extend({
    model: function() {
-     return this.store.findAll('band');
+     var bands = this.store.findAll('band');
+     return wait(bands, 3 * 1000);
    },
  });
```

Indeed, when the `bands` route is entered, we see the loading screen as intended:



The important thing to notice here is that the loading template is looked up as a sibling of the route being entered, or in other words, at the same level.

That implies each level has its own loading route. We can thus descend one level and create a loading route for the `band` route. The beauty of this concept is that we only need the template to indicate that the app is loading data:

```
$ ember generate template bands/band/loading
```

```
<!-- app/templates/bands/band/loading.hbs -->
<div class="loading-pane">
  <div class="loading-message">
    Fetching band data
  </div>
  <div class="spinner"></div>
</div>
```
HBS

Before we add an artificial delay on the next level of routes, too, let's extract and enhance the `wait` helper we used above.

```js
// app/utils/wait.js
import { resolve, Promise as EmberPromise } from 'rsvp';

export default function wait(value, delay) {
  var promise = value.then && typeof value.then === 'function' ?
      value :
      resolve(value);

  return new EmberPromise(function(resolve) {
    setTimeout(function() {
      promise.then(function(result) {
        resolve(result);
      });
    }, delay);
  });
}
```

JS

The only thing we added is the line that makes sure `promise` is actually a promise. It does this by checking whether the `then` property of the object is a function, and if it's not, it wraps the value in an 'always successfully resolving' promise.

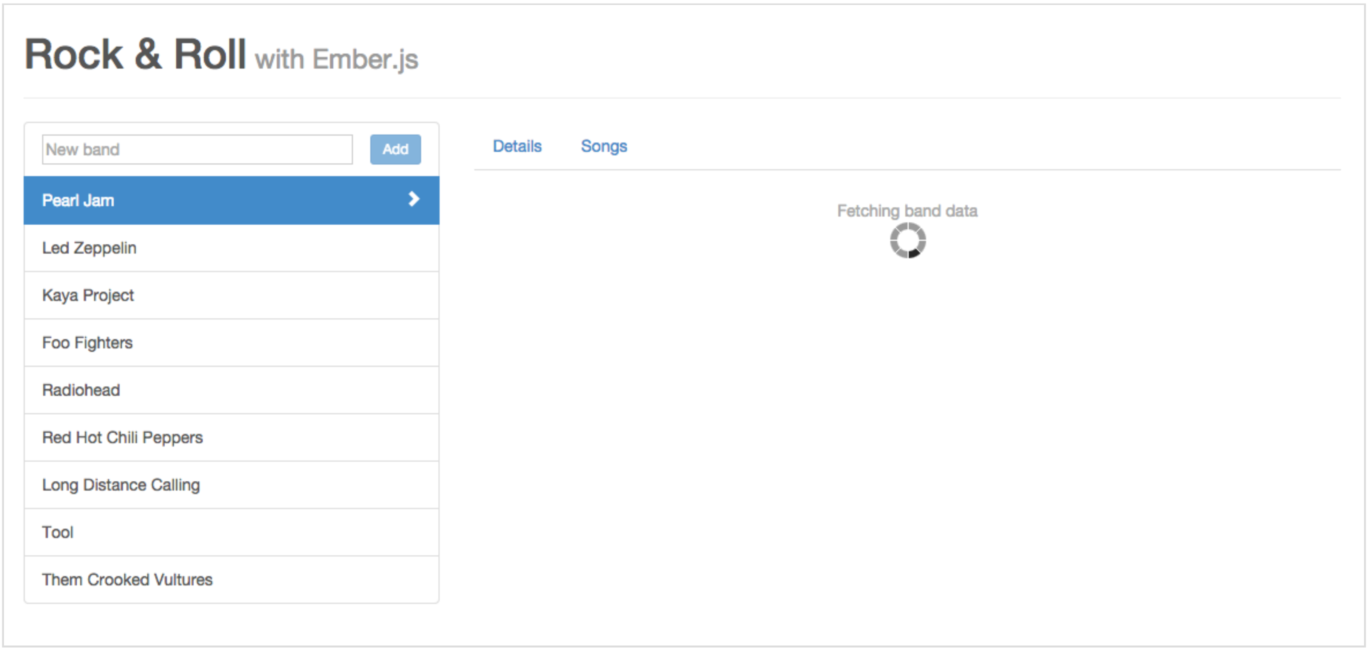
We can now use our improved `wait` function to delay the loading of songs in both `bands` and `bands.band.songs`.

```
  // app/routes/bands.js
- import { Promise as EmberPromise } from 'rsvp';
+ import wait from '../utils/wait';

- function wait(promise, delay) {
-    return new EmberPromise(function(resolve) {
-      setTimeout(function() {
-        promise.then(function(result) {
-          resolve(result);
-        });
-      }, delay);
-    });
- }
  export default Route.extend({
    model: function() {
      var bands = this.store.findAll('band');
      return wait(bands, 3 * 1000);
    },
    (...)
  });
```

```
  // app/routes/bands/band/songs.js
+ import wait from '../../../utils/wait';
  export default Route.extend({
    model: function() {
-     return this.modelFor('bands.band');
+     return wait(this.modelFor('bands.band'), 3000);
    },
    (...)
  });
```

(Moving the promise wrapping behavior into the util function already paid off. `this.modelFor` does not return a promise, so we'd have had to manually convert it to a promise if `wait` did not handle this.)

We will see our loading template instead of the songs:



The same loading route is entered from any child route of `bands.band`, so also from `bands.band.details`.

It is important to point out that our `wait` function only serves demo purposes to simulate network latency. We should thus restore the original model hooks:

```js
// app/routes/bands.js
export default Route.extend({
  model: function() {
    return this.store.findAll('band');
  },
});
```

```js
// app/routes/bands/band/songs.js
export default Route.extend({
  model: function() {
    return this.modelFor('bands.band');
  },
});
```

Under normal circumstances, network latency happens on its own and so we can rest assured we'll see our pretty spinners.

BACKSTAGE

RENDERING THE LOADING TEMPLATE IS AN OVERRIDABLE DEFAULT

Rendering the `loading` template at the level of the route is the default response to the loading event being triggered.

If you wish to do something else in response, you just have to write the action handler in the route for the `loading` action. You can, for example, render a template with a different name or trigger another action that can bubble up the route tree.

Here, I'll show a third option, throwing up an ugly alert box:

```js
// app/routes/bands/band/songs.js
import { resolve } from "rsvp";

export default Route.extend({
  model: function() {
    var promise = resolve(this.modelFor('bands.band'));
    return wait(promise, 3 * 1000);
  },

  actions: {
    loading: function() {
      window.alert('Loading the band's songs, ok?');
    },
    (...)
  }
});

```

JS

Error routes

Error routes are activated when a model hook throws an error, either because the backend returned an error response or because the application code itself ran into a problem.

Error routes work exactly the same as loading routes:

- An `error` event is fired on the route where the error was produced
- The default behavior is to render an `error` template on the level of the route that triggered the error
- All of the Ember arsenal is at our disposal to implement a custom behavior. It is just a matter of implementing an action handler for the `error` action in the route.

It only takes a template to show an error message when loading a particular band does not work as it should:

```
$ ember generate template bands/band/error
```

```hbs
<!-- app/templates/bands/band/error.hbs -->
<div class="error-pane">
  <div class="error-message">
    Something went wrong while fetching data for the band.
  </div>
</div>
```

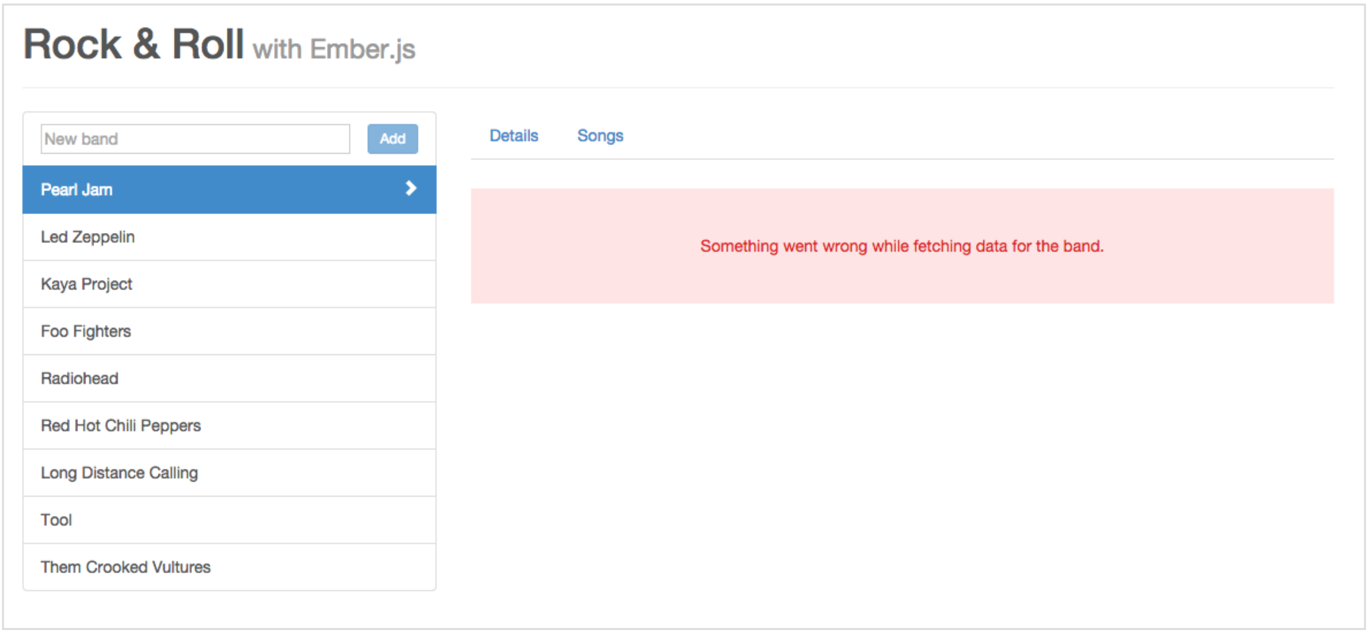
HBS

We can test the displaying of the error template by making a rejecting promise in the model hook:

```
// app/routes/bands/band/songs.js
import RSVP from "rsvp";

export default Route.extend({
  model: function() {
-   return this.modelFor('bands.band');
+   return RSVP.reject(this.modelFor('bands.band'));
  },
  (...)
});
```

JS

And indeed the error message is rendered on screen:



Something went wrong while fetching data for the band.

Let's not forget to restore the original model hook for the `songs` route:

```js
// app/routes/bands/band/songs.js
export default Route.extend({
  model: function() {
    return this.modelFor('bands.band');
  },
  (...)
});
```

Next song

We'll increase the aesthetics of our application by always displaying band names and song titles in a consistent and elegant way, no matter how the user typed them in. We'll use a template helper to achieve that.

CHAPTER 14

Helpers

Tuning

For users of our application, it's tiresome to pay attention to typing band names and song titles so that each word Starts With A Capital Letter. It would be nice if the application made sure that no matter how the names are entered, they're displayed in this readable way.

In this chapter, we'll define a template helper that can help us do that.

About helpers

Helpers are functions with various use cases. Some of them control what is shown in the template, others transform data for display or create a view in a succinct form.

We have used a great number of helpers already in our application, possibly even without being aware of it. `if`, `each`, `outlet`, `action` and `link-to` were all helpers.

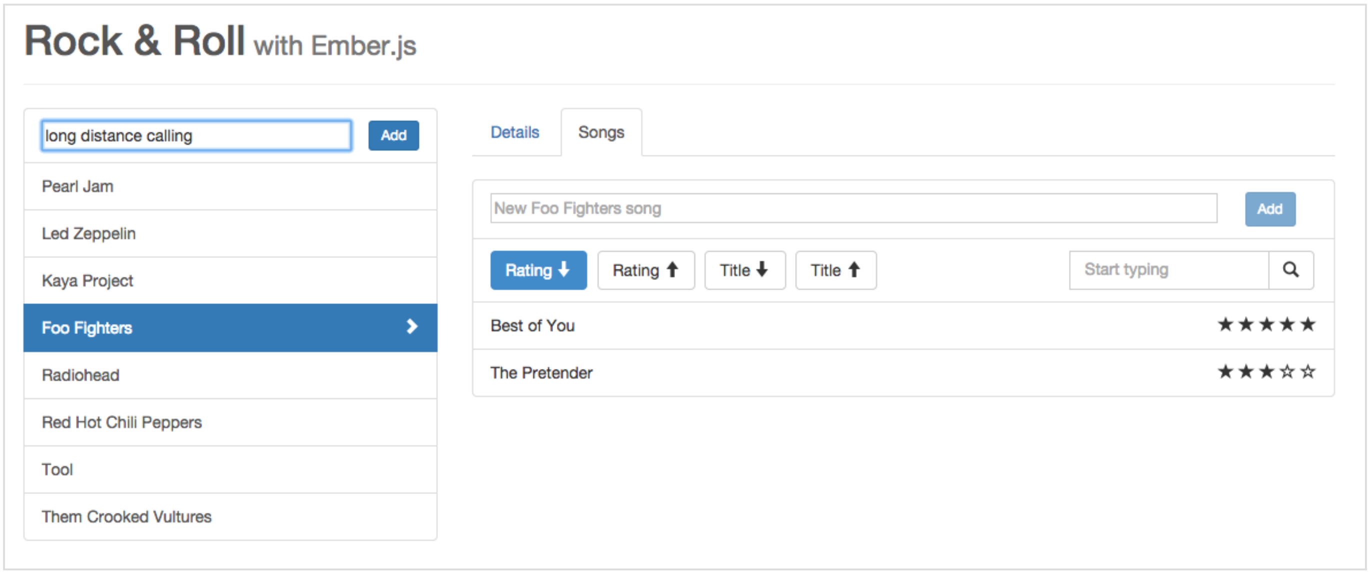
You should consider using a helper when you need to derive data for display from some source, but do not need the complex markup or event handling provided by components. The following table summarizes a few typical use cases for helpers.

Category	Template content	Output
Timestamps	{{ago message.createdAt}}	24 minutes ago
Translation	{{translate 'search'}}	Chercher
Localizations	{{localize product.price}}	$4.99
Text transformations	{{shout 'silence'}}	SILENCE!

Capitalizing each word of a name

Our current feature request is eligible for being implemented by a helper, since it's a simple text transformation.

We want to allow the user to type 'long distance calling' as a band name and still have it displayed as 'Long Distance Calling':



We begin, as usual, by using the appropriate Ember CLI generator. This time, we need a helper called `capitalize`:

```
$ ember generate helper capitalize
```

That creates a `app/helpers/capitalize.js` file that we should take a look at:

```js
// app/helpers/capitalize.js
import { helper } from '@ember/component/helper';

export function capitalize(params/*, hash*/) {
  return params;
}

export default helper(capitalize);
```

Interestingly, both a named and a default export are created in the module.

The default export is the helper we can apply in our templates. `Helper.helper` takes an ordinary function and wraps it so that the helper function establishes a binding. If the property passed to the helper changes, the helper rerenders its output. In our case, if the title of the song was edited, all instances which used the `capitalize` helper would be rerun to produce a new output for the changed input.

The module also establishes a named export, `capitalize`. This allows us to use this function in non-template contexts, like routes and controllers.

Let's add the code that implements splitting apart several words on whitespace and then capitalizing each word:

```
  // app/helpers/capitalize.js
  import { helper } from '@ember/component/helper';

- export function capitalize(params/*, hash*/) {
+ export function capitalize(input) {
+   var words = input.toString().split(/\s+/).map(function(word) {
+     return word.toLowerCase().capitalize();
+   });

-   return params;
+   return words.join(' ');
  }

  export default helper(capitalize);
```

The passed-in value (which can be the name of a band or a song title) is split apart on whitespace and then joined together by single spaces after each word has been made to start with a capital letter.

The final step is to use the new helper in the templates, to capitalize band names:

```
  <!-- app/templates/bands.hbs -->
  (...)
  {{#each model as |band|}}
    {{#link-to "bands.band" band class="list-group-item band-link"}}
-     {{band.name}}
+     {{capitalize band.name}}
      <span class="pointer glyphicon glyphicon-chevron-right"></span>
    {{/link-to}}
  {{/each}}
  (...)
```

and song titles:

```hbs
  <!-- app/templates/bands/band/songs.hbs -->
  (...)
  {{#each sortedSongs as |song|}}
    <li class="list-group-item song">
-     {{song.title}}
+     {{capitalize song.title}}
      {{star-rating item=song rating=song.rating on-click=(action "updateRating")}}
    </li>
  {{/each}}
```

Using it in templates, or elsewhere

A benefit of defining helpers as simple functions and then registering a template helper that uses that function is that we can use the function everywhere, reducing code repetition and thus maintenance costs.

We need to capitalize values in a couple of other places. First, let's introduce a more telling placeholder for the new song text field:

```hbs
  <!-- app/templates/bands/band/songs.hbs -->
  <ul class="list-group songs">
    (...)
-   {{input type="text" class="new-song" placeholder="New song" value=title}}
+   {{input type="text" class="new-song" placeholder=newSongPlaceholder value=title}}
  </ul>
```

Gaining access to the function is a matter of importing the named export:

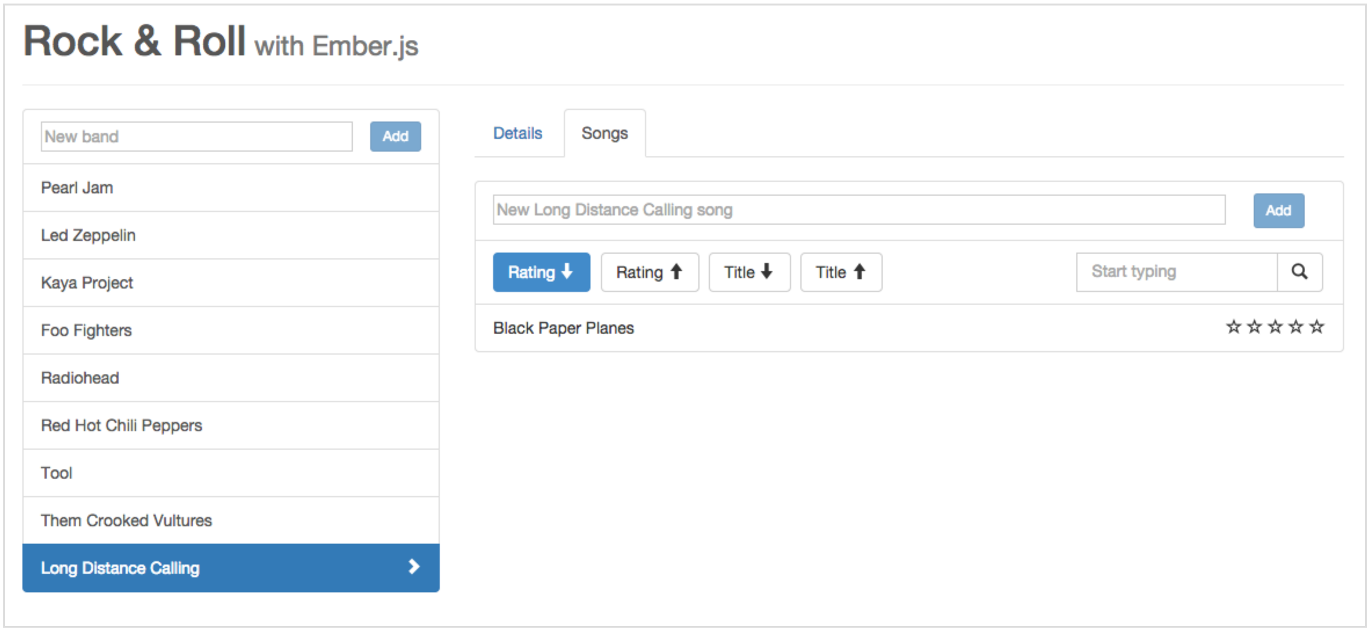
```js
  // app/controllers/bands/band/songs.js
+ import { capitalize } from 'rarwe/helpers/capitalize';

  export default Controller.extend({
    (...)
+   newSongPlaceholder: computed('model.name', function() {
+     var bandName = this.get('model.name');
+     return `New ${capitalize(bandName)} song`;
+   }),
    actions: {
      (...)
    }
  })
```

Since this is a named export, the exact name needs to be used when importing it. However, we can refer it by a different name in each module that we import it to, by using the `as` keyword in the import statement.

Let's do that for the other place we need the capitalize function, the document title for the songs page:

```js
  // app/routes/bands/band/songs.js
+ import { capitalize as capitalizeWords } from 'rarwe/helpers/capitalize';

  export default Route.extend({
    (...)
    actions: {
      didTransition: function() {
        var band = this.modelFor('bands.band');
-       document.title = `${band.get('name')} songs - Rock & Roll`;
+       var name = capitalizeWords(band.get('name'));
+       document.title = `${name} songs - Rock & Roll`;
      },
    }
    (...)
  });
```

Band names and song titles are now correctly capitalized everywhere they're displayed:



The road says

"Tests are great but they also have to be maintained. You now have a few of them that now fail because all song titles are rendered as capitalized words but some assertions still use mixed case."

He is right again, so let's modify those song titles in the assertions:

```
  // tests/acceptance/bands-test.js
  test('Sort songs in various ways', function(assert) {
    (...)
    click('button.sort-title-desc');

    andThen(function() {
      assert.equal(currentURL(), '/bands/1/songs?sort=titleDesc');
-     assertTrimmedText(assert, '.song:first', 'Spinning in Daffodils', 'The first song is the one that is the last in the alphabet');
+     assertTrimmedText(assert, '.song:first', 'Spinning In Daffodils', 'The first song is the one that is the last in the alphabet');
      (...)
    });

    click('button.sort-rating-asc');

    andThen(function() {
      assert.equal(currentURL(), '/bands/1/songs?sort=ratingAsc');
      (...)
-     assertTrimmedText(assert, '.song:last', 'Spinning in Daffodils', 'The last song is the highest ranked, last in the alphabet');
+     assertTrimmedText(assert, '.song:last', 'Spinning In Daffodils', 'The last song is the highest ranked, last in the alphabet');
    });
  });
```

JS

Next song

Next we'll take a look at how to add clear feedback to our application through the use of animations.

CHAPTER 15

Animations

Tuning

Rich client-side applications can mean different things for different people, but most would agree that rich user interactions are a common feature. However, rich should not mean confusing. Widgets should be a pleasure to use, and their use should be evident. Components should give clear feedback on how the the user has changed the application's state, or whether it has changed at all.

Animations, if applied judiciously, can play a significant role in feedback. Nuanced transitions signal to the user what has changed. Toward this end, we will add a couple of simple animated transitions in this chapter.

For a long time, it was very hard to integrate animations in Ember applications. That changed when Edward Faulkner released liquid-fire, which has since become an Ember add-on and the de facto standard animation tool for Ember.

Setting up liquid-fire

Since `liquid-fire` is a standard Ember CLI add-on, it is very simple to add it to our application:

```
ember install liquid-fire
```

That will not do anything to our application, though, as animations have to be defined and implemented first. Let's restart the server and get some grounding.

liquid-fire architecture

"Transition", meaning changing from one state to another, is the key concept in liquid-fire. That change can be between two routes, two values or two collections of objects. Transitions can also be animated and, in this chapter, they will be so I will use the terms "transitions" and "animations" interchangeably. There are three components of an animation in liquid-fire: the transition map, the transition functions and the template helpers.

Transition map

The transition map is similar to the routing map and lives under `app/transitions.js`. Just as the routing map defines the routes of an Ember application, the transition map defines the transitions. In other words, it defines *what* should be animated.

Transition functions

The transition functions implement the actual animation from the old state to the new state. Each animation function resides in its own module and exports a single function. An example would look like this:

```js
import { animate, stop, Promise } from 'liquid-fire';

export default function crossFade(opts) {
  opts = opts || {};
  stop(this.oldElement);
  return Promise.all([
    animate(this.oldElement, { opacity: 0 }, opts),
    animate(this.newElement, { opacity: [ (opts.maxOpacity || 1), 0 ]
}, opts)
  ]);
}
```

If the above content was stored in `app/transitions/cross-fade.js`, it would give us an animation name of `cross-fade` that we could use in the transition map (we'll see how in this chapter). The good news is that a handful of animations (including cross-fade, actually) are part of the library so we don't need to write any code for implementing animations while we familiarize ourselves with liquid-fire.

Transition functions define *how* the transitions matched by the transition map should be animated.

Template helpers

The third and final piece of the puzzle is template helpers. Ember uses Handlebars as its templating language and adds helpers to it to enable certain features. One example is the `link-to` helper for moving between routes, something that the pure Handlebars language does not support.

However, these Ember helpers don't know anything about animations so "transition-aware" counterparts need to be defined. These have the liquid- prefix followed by the name of the Ember helper so we have `liquid-if`, `liquid-with`, `liquid-outlet`, and so on. Since these helpers observe values passed to them, they can pass the observed change to the transition map. The transition map can then decide whether there is a matching rule and then invoke the corresponding animation.

The helpers thus enable any animation to happen at all.

Putting it all together

The liquid-fire helpers in the templates observe values they have bindings on. The observed values can be passed explicitly, like an `isEditing` property passed to a `liquid-if`, or be implicit, like router transitions for `liquid-outlets`.

However that happens, the helpers pass the observed state change to the transition map. The map then tries to find a matching transition for the change that satisfies all constraints.

If it finds such a transition, it invokes the animation function defined for the transition to carry out the actual animation.

This architecture is splendidly declarative, decoupled and thus highly flexible. Now that we understand how it works *in theory*, let's get down and add a couple of animations, to see it function *in practice*.

Animate moving between routes

For our first animation, let going between the band details and band songs tabs be animated with a slide-in, slide-out effect.

We first need to enable this by turning the stock Ember helper that is responsible for re-rendering part of the page when a router transition happens to an "animation-aware" liquid-fire helper. That helper is `outlet` which needs to be converted into `liquid-outlet`. Right, but which one?

The routes we are interested in are `bands.band.details` and `bands.band.songs`. We saw in the Nested routes chapter, that the content rendered for these routes needs to reside in the template of their parent route, `bands.band`. So let's open up that template and make the change:

```
  // app/templates/bands/band.hbs
  <ul class="nav nav-tabs">
    <li>{{link-to "Details" "bands.band.details" model}}</li>
    <li>{{link-to "Songs"   "bands.band.songs"   model}}</li>
  </ul>
  <div class="band-info">
-   {{outlet}}
+   {{liquid-outlet}}
  </div>
```

The animation is now enabled in the view layer but there should be a transition that matches the change of navigating between these two routes. We don't have a transition map yet, so let's create the `app/transitions.js` file and paste in the following:

```js
// app/transitions.js
export default function() {
  this.transition(
    this.fromRoute('bands.band.songs'),
    this.toRoute('bands.band.details'),
    this.use('toRight'),
    this.reverse('toLeft')
  );
}
```

Each call to `this.transition` defines a rule. Its arguments define constraints and declare which animation(s) should be used if all the constraints are matched. In the above example `this.fromRoute` and `this.toRoute` constrain the transition to be activated only if the route changes from `bands.band.songs` to `bands.band.details` and specify the `toRight` animation to be used. The last argument, `this.reverse` is a nifty shorthand that says that if the transition happens in the other direction, from `bands.band.details` to `bands.band.songs`, the `toLeft` animation should be employed.

As the third and final step, we have to implement the `toRight` and `toLeft` animations. Only, we don't, as these animations are included in the "starter kit" that comes with liquid-fire (see the full list of which animations are provided by the addon here).

If we click (or tap) the navigation tabs, we can see that our animation really works.

Animate a value change

As our second animation, let's add some visual feedback to the user after a band's description has been updated.

We currently handle the save operation first in the controller, then let it bubble and catch it in the route (by returning true from the action handler in the controller):

```js
// app/controllers/bands/band/details.js
export default Controller.extend({
  isEditing: false,

  actions: {
    edit: function() {
      this.set('isEditing', true);
    },
    save: function() {
      this.set('isEditing', false);
      return true;
    }
  }
});
```

JS

```js
// app/routes/bands/band/details.js
export default Route.extend({
  actions: {
    save: function() {
      var controller = this.get('controller'),
          band = controller.get('model');

      return band.save();
    },
    (...)
  }
});
```

JS

From this, we can see that animating the `isEditing` property on the controller is what we need. First, we need to enable the animation in the view layer by using the liquid helper counterpart of the Ember helper. Here is the template:

```
<!-- app/templates/bands/band/details.hbs -->
<div class="panel panel-default band-panel">
  <div class="panel-body">
    (...)
    {{#if isEditing}}
      <div class="form-group">
        {{textarea class="form-control" value=model.description}}
      </div>
    {{else}}
      <p>{{model.description}}</p>
    {{/if}}
  </div>
</div>
```

The `if` helper needs to be replaced by `liquid-if`. In addition, we'll add a CSS class to the element that `liquid-if` creates in order to use it in the transition map in the next step:

```
  <!-- app/templates/bands/band/details.hbs -->
  <div class="panel panel-default band-panel">
    <div class="panel-body">
      (...)
-     {{if isEditing}}
+     {{#liquid-if isEditing class="band-description"}}
        <div class="form-group">
          {{textarea class="form-control" value=model.description}}
        </div>
      {{else}}
        <p>{{model.description}}</p>
-     {{/if}}
+     {{/liquid-if}}
    </div>
  </div>
```

We can now define the constraints for the animation by adding a call to `this.transition` in the transition map:

```js
// app/transitions.js
export default function() {
  this.transition(
    this.fromRoute('bands.band.songs'),
    this.toRoute('bands.band.details'),
    this.use('toRight'),
    this.reverse('toLeft')
  );
+  this.transition(
+    this.hasClass('band-description'),
+    this.toValue(false),
+    this.use('fade', { duration: 500 })
+  );
}
```

We use two new constraints, `hasClass` and `toValue`. `hasClass` matches elements that have that CSS class (that's why we added `band-description` in the previous step). `toValue` matches if the new value of the observed property is equal to the parameter passed in. Since we want our animation only to be triggered when the user saves the band's description (in other words, when `isEditing` becomes false), we pass in `false`.

Another stock animation, `fade`, is used. Options can be passed in as a second parameter and we leverage that to make the fade take 500 milliseconds.

Just like in the previous case, as `fade` is part of liquid-fire, the animation function does not need to be defined, we can see our animation right away.

Adding a custom animation

When the rating of a song is updated by clicking on one of the stars beside it, the position of that song might change and so it can be a little unclear that the rating has been successfully updated since the song might be shifted up- or downwards in the list.

As the third and final animation, we'll make that clearer by bumping the stars of the song slightly when its rating is changed. By now we know that as a first step, a liquid-fire helper needs to be used in the appropriate template, which is, in this case, the template of the star-rating component:

```
<!-- app/templates/components/star-rating.hbs -->
{{#each stars as |star|}}
  <a href="#" {{action "updateRating" star.rating}}
     class="star-rating glyphicon {{if star.full 'glyphicon-star' 'glyphicon-star-empty'}}">
  </a>
{{/each}}
```

Our first thought would be to replace `#each` with `#liquid-each` and there is actually an open issue in the liquid-fire repository to add a liquid-each helper, but it does not yet exist. We could achieve the same effect, though, with wrapping the `#each` in a `liquid-bind`, thusly:

```
  <!-- app/templates/components/star-rating.hbs -->
-   {{#each stars as |star|}}
+ {{#liquid-bind stars as |liquidStars|}}
+   {{#each liquidStars as |star|}}
      <a href="#" {{action "updateRating" star.rating}}
         class="star-rating glyphicon {{if star.full 'glyphicon-star' 'glyphicon-star-empty'}}">
      </a>
    {{/each}}
+ {{/liquid-bind}}
```

We wrapped the `#each` in a `liquid-bind` and looped through the variable yielded back by it, `liquidStars`, in the `#each`. Whenever `stars` changes, the `liquid-bind` will observe that change and pass it to the transition map. Because `stars` changes whenever the song's rating changes, this fits our needs.

Let's now add a transition rule in the transition map to match that animation:

```
  // app/transitions.js
  export default function(){
    (...)
+   this.transition(
+     this.inHelper('liquid-bind'),
+     this.use('slight-scale')
+   );
  }
```

We haven't used `this.inHelper` yet. It matches a transition that was observed by the helper that we pass to it, in this case, `liquid-bind`. As we currently have only one instance of that helper, this will work, though it might be prudent to add a css class and then use `this.hasClass(...)`, just as we did for the band's description textarea in the previous section.

We ask for the `slight-scale` animation to be used but this animation is not a built-in one, so we have to implement it. First, we create a stub for the transition:

```
$ ember generate transition slight-scale
```

This has generated a file called `slight-scale.js` in the `app/transitions` folder. Let's open it and replace its contents with the following code:

```
// app/transitions/slight-scale.js
import { animate } from "liquid-fire";

export default function scale(opts={}) {
  var transition = this;
  return animate(transition.oldElement, {scale: [0.9, 1]}, opts).then(function() {
    return animate(transition.newElement, {scale: [1, 0.9]}, opts);
  });
}
```

`animate` is a very useful helper to define custom animations with Velocity.js, an animation library. It returns a promise which gives a simple API to compose animations. However, using `animate` (and, for that matter, Velocity.js) is optional. Animations can be implemented using any technique or library as long as they return a promise from the animation function.

`transition.oldElement` and `transition.newElement` are the elements to transition from and transition to, respectively. We thus first scale the set of stars back a bit (going from its original size to 0.9 of it), and once that has finished, doing the reverse. That will look like the stars are bumped slightly.

As the songs are still sorted right after a song's rating changes, the song will move and animate at the same time, which is a bit confusing. So, to see our animation in full glory, let's first disable the sorting by making the `sortedSongs` property on the controller just an alias to the `matchingSongs` one:

```diff
  // app/controllers/bands/band/songs.js
- import { sort } from "@ember/object/computed";
+ import { sort, alias } from "@ember/object/computed";

  export default Controller.extend({
    (...)
-   sortedSongs: sort('matchingSongs', 'sortProperties'),
+   sortedSongs: alias('matchingSongs'),
    (...)
  });
```

We can now observe how a visual clue is given when a song's rating is updated.

Debugging transitions

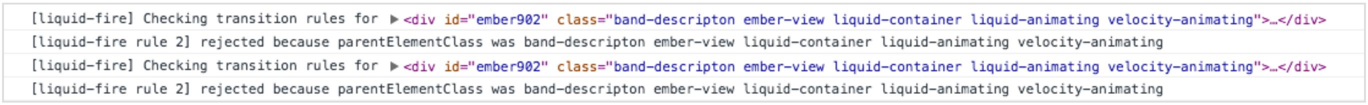
Finding out why a certain animation is not triggered can be daunting at times. Fortunately, liquid-fire provides a great helper called `this.debug` that you have to insert into the transition rule you're trying to fix.

Let's assume that I mistyped the CSS class for the band's description in the template as `band-descripton` and was pulling my hair out over why the textarea does not fade back as the transition rule posits.

To find out why, we first have to add `this.debug` to the transition rule:

```js
// app/transitions.js
export default function(){
  (...)
  this.transition(
    this.hasClass('band-description'),
    this.toValue(false),
    this.use('fade', { duration: 500 }),
+   this.debug()
  );
  (...)
}
```

And then let's take the user action that should trigger the animation, in this case, clicking a rating star. No animation is seen but the transition rule we added the debug top prints debug information to the console so we see why it did not match:



```
[liquid-fire] Checking transition rules for ▶<div id="ember902" class="band-descripton ember-view liquid-container liquid-animating velocity-animating">…</div>
[liquid-fire rule 2] rejected because parentElementClass was band-descripton ember-view liquid-container liquid-animating velocity-animating
[liquid-fire] Checking transition rules for ▶<div id="ember902" class="band-descripton ember-view liquid-container liquid-animating velocity-animating">…</div>
[liquid-fire rule 2] rejected because parentElementClass was band-descripton ember-view liquid-container liquid-animating velocity-animating
```

The `debug` helper makes it a whole lot easier to debug animation issues in our apps.

The road says

"You know what? Your rating stars now animate perfectly but you killed the sorting of the songs which is a lot worse, what do you think?"

The road is probably right, so let's go in and restore the sorting of the songs:

```
  // app/controllers/bands/band/songs.js
- import { sort, alias } from "@ember/object/computed";
+ import { sort } from "@ember/object/computed";

  export default Controller.extend({
    (...)
-   sortedSongs: alias('matchingSongs'),
+   sortedSongs: computed.sort('matchingSongs', 'sortProperties'),
    (...)
  });
```

We can also remove everything related to making the stars bump, which means we can restore the component's template:

```
<!-- app/templates/components/star-rating.hbs -->
{{#each stars as |star|}}
  <a href="#" {{action "updateRating" star.rating}}
     class="star-rating glyphicon {{if star.full 'glyphicon-star' 'glyphicon-star-empty'}}">
  </a>
{{/each}}
```

, and remove the transition rule from the transition map:

```js
  // app/transitions.js
  export default function(){
    this.transition(
      this.fromRoute('bands.band.songs'),
      this.toRoute('bands.band.details'),
      this.use('toRight'),
      this.reverse('toLeft')
    );
    this.transition(
      this.hasClass('band-description'),
      this.toValue(false),
      this.use('fade', { duration: 500 }),
    );
-   this.transition(
-     this.inHelper('liquid-bind'),
-     this.use('slight-scale')
-   );
  }
```

And the custom animation that we implemented:

```
$ rm app/transitions/slight-scale.js
```

Next song

Although we no longer have the "animation markup" (the `liquid-bind` helper) in the template of the star-rating component, the fact that we had to add it in there is disquieting. After all, we should be able to use the rating stars with- or without animations in the same project.

The component's template should be independent of other libraries in the application, and we violated this reasonable assumption by including a liquid-fire helper in the template.

We'll solve this concern by "escaping forward". We'll extract the `star-rating` widget into an Ember addon and provide it with a block form invocation so that it can be freely customized.

CHAPTER 16

Making an Ember addon

Tuning

We saw that in order to make the stars bump a little when the song's rating is updated, we had to edit the component's template to add the liquid-fire markup. Doing this would have prevented us to use a non-animating version of the star-rating component since the component is always shipped with its template.

To make our component truly flexible (and thus reusable) we should provide both a block and a non-block form. The block form will allow consuming applications to customize its markup (for example, to add animations) while the non-block form will provide a sane default that should apply to many usage scenarios out-of-the-box.

To take it a step further and foster its reusability, we'll make the component into an Ember CLI addon and then import it into our app.

What is an Ember addon?

An addon in Ember is like a Javascript library or a Ruby gem. It bundles code that implements a certain task (like setting the document title or displaying a flash message), and allows it to be distributed so that it can be used in any Ember application. Ember CLI is equipped with the `ember addon` command to create them. Before you make a new addon, change to a directory where you want to host it and then do:

```
$ ember addon ember-cli-star-rating
$ cd ember-cli-star-rating
```

That creates a file structure quite similar to an app's with a few notable differences:

```
ember-cli-star-rating
├── LICENSE.md
├── README.md
├── addon
├── app
├── config
│   ├── ember-try.js
│   └── environment.js
├── ember-cli-build.js
├── index.js
├── node\_modules
│   ├── (...)
├── package.json
├── testem.json
├── tests
│   ├── dummy
│   ├── helpers
│   ├── index.html
│   ├── integration
│   ├── test-helper.js
│   └── unit
└── vendor
```

The main difference is the presence of the `addon` folder. This is where most of the code of the addon lives and is equivalent of the `app` folder for applications.

Since Ember addons are valid npm packages, they need to have a `package.json` and an `index.js` file in the top-level folder. Because the `package.json` file contains the addon's configuration, let's peek into that first:

```
// package.json
{
  "name": "ember-cli-star-rating",
  "version": "0.0.0",
  "description": "The default blueprint for ember-cli addons.",
  (...)
  "keywords": [
    "ember-addon"
  ],
  (...)
  "repository": "",
  (...)
  "ember-addon": {
    "configPath": "tests/dummy/config"
  }
}
```

The name relates to the npm package. The `ember-addon` keyword is paramount as this is how Ember CLI detects Ember addons in a project's dependency. The `ember-addon` hash is for the configuration of the addon itself (as opposed to that of the npm package). `configPath` shows the path for the configuration file of the Ember app that Ember CLI will use when commands are issued within the addon (like `ember serve`).

The `repository` field is where the package is hosted so let's fill it out with its home on Github:

```
// package.json
{
  "name": "ember-cli-star-rating",
  (...)
- "repository": "",
+ "repository": "https://github.com/balinterdi/ember-cli-star-rating",
}
```

According to npm conventions, the `index.js` file is the entry point for the npm package. It now only has the module's name:

```js
// index.js
/* eslint-env node */
'use strict';

module.exports = {
  name: 'ember-cli-star-rating'
};
```

Developing the addon

We can now go ahead and create the `star-rating` component for our addon:

```
$ ember generate component star-rating
installing component
  create addon/components/star-rating.js
  create addon/templates/components/star-rating.hbs
installing component-test
  create tests/integration/components/star-rating-test.js
installing component-addon
  create app/components/star-rating.js
```

Notice that Ember CLI detected that we're inside an addon and thus created the component's files inside the `addon` folder. Let's take a look at the component stub:

```js
// addon/components/star-rating.js
import Component from '@ember/component';
import layout from '../templates/components/star-rating';

export default Component.extend({
  layout
});
```

The content is very similar to that of a component created inside an Ember app with one exception. In addons, the component's template needs to be explicitly imported and assigned to the layout property of the component whereas in apps, the resolver sets this up by virtue of having the template placed on the correct path.

The generated content for the component is exactly the same as if we were in an application:

```
<!-- addon/templates/components/star-rating.hbs -->
{{yield}}
```

We can copy both the component file and its template from our application:

```js
// addon/components/star-rating.js
import Component from '@ember/component';
import { computed } from '@ember/object';
import layout from '../templates/components/star-rating';

export default Component.extend({
  layout: layout,
  tagName: 'div',
  classNames: ['rating-panel'],

  rating:     0,
  maxRating:  5,
  item:       null,
  "on-click": '',

  stars: computed('rating', 'maxRating', function() {
    var fullStars = this.starRange(1, this.get('rating'), 'full');
    var emptyStars = this.starRange(this.get('rating') + 1,
this.get('maxRating'), 'empty');
    return fullStars.concat(emptyStars);
  }),

  starRange: function(start, end, type) {
    var starsData = [];
    for (var i = start; i <= end; i++) {
      starsData.push({ rating: i, full: type === 'full' });
    }
    return starsData;
  },

  actions: {
    setRating: function(newRating) {
      this.get('on-click')({
        item: this.get('item'),
        rating: newRating
      });
    }
```

JS

```
  }
});
```

```
<!-- addon/templates/components/star-rating.hbs -->
{{#each stars as |star|}}
  <a href="#" {{action "setRating" star.rating}}
     class="star-rating glyphicon {{if star.full 'glyphicon-star'
'glyphicon-star-empty'}}">
  </a>
{{/each}}
```

In order for consuming applications to be able to use the star-rating widget, we need to establish a bridge between the addon and the app. This happens in the `app` folder of the addon and has already been accomplished by the `ember generate component star-rating` command:

```js
// app/components/star-rating.js
export { default } from 'ember-cli-star-rating/components/star-rating';
```

We are now ready to integrate the addon into our application.

Integrating our addon into our app

When we are developing our addon locally, we have to link to the local folder of the addon from the host application. The `npm link` command serves that purpose and has two phases. First, let's go to the addon directory and issue the following command:

```
$ npm link
```

(If you use yarn, type `yarn link`).

That will create a global symlink for the package name (in our case, `ember-cli-star-rating`) and thus prepares that package to be used from host applications. We can now go to our application and complete the linking process:

```
$ npm link ember-cli-star-rating
```

(If you use yarn, type `yarn link ember-cli-star-rating`).

If all went well, we should see the created link in the node_modules directory:

```
$ ls node_modules
lrwxr-xr-x   1 (...) ember-cli-star-rating@ -> /usr/local/lib/
node_modules/ember-cli-star-rating
```

The linking is done but we also need to specify the dependency in `package.json`. Any valid package version will do, I used the asterisk:

```
// package.json
{
  "name": "rarwe",
  (...)
  "devDependencies": {
    (...)
    "ember-cli-star-rating": "*"
  }
}
```

JS

Before relaunching the server, let's delete all files related to the component so that they don't conflict with the files in the addon:

```
$ rm app/components/star-rating.js
$ rm app/templates/components/star-rating.hbs
$ rm tests/integration/components/star-rating-test.js
```

When you run `ember serve`, you might still get an error message like the following:

```
Addon templates were detected, but there are no template compilers registered
for `ember-cli-star-rating`. Please make sure your template precompiler
(commonly `ember-cli-htmlbars`) is listed in `dependencies`
(NOT `devDependencies`) in `ember-cli-star-rating`'s `package.json`.
```

Let's do as we're told and move the `ember-cli-htmlbars` line from `devDependencies` to `dependencies`, in the addon's `package.json`:

```
  // package.json
  {
    "name": "ember-cli-star-rating",
    "devDependencies": {
      (...)
-     "ember-cli-htmlbars": "^2.0.1",
    },
    "keywords": [
      "ember-addon"
    ],
    "dependencies": {
+     "ember-cli-htmlbars": "^2.0.1",
      (...)
    },
  }
```

Victory is ours! When we run the `ember serve` command again, not only does the app boots up correctly, our rating stars work as before. However, they are now served from the addon!

Adding a block form

We have now extracted the `star-rating` component into its own addon and we have the exact same functionality in the app as before. We can now take a step further and allow the component's layout to be dictated by the consuming application which would make it possible to have animating stars or the rating stars to be used with different markup (it currently uses the `glyphicon` classes specific to Bootstrap). In other words, this means using the block form of the component, something we don't yet have.

The current, non-block, form uses the following invocation pattern:

```
{{star-rating item=song rating=song.rating on-click=(action "updateRating")}}
```

We would like to enable a block form so that the "caller" can decide how to draw the stars. However, we'd also like to keep the non-block form working. Here is how to amend the template to cover both cases:

```
  <!-- addon/templates/components/star-rating.hbs -->
+ {{#if hasBlock}}
+   {{yield stars}}
+ {{else}}
    {{#each stars as |star|}}
      <a href="#" {{action "setRating" star.rating}}
         class="star-rating glyphicon {{if star.full 'glyphicon-star' 'glyphicon-star-empty'}}">
      </a>
    {{/each}}
+ {{/if}}
```

`hasBlock` will be true if the block form is used. `yield` then gives back the `stars` property of the component to the caller so that it can do as it pleases.

Customizing the star-rating component's look and behavior

We could now re-enable the bumping of stars when the rating is updated without changing the template of the `star-rating` component, by using its block form:

```
  <!-- app/templates/bands/band/songs.hbs -->
  <ul class="list-group songs">
    {{#each sortedSongs as |song|}}
      <li class="list-group-item song">
        {{capitalize song.title}}
-       {{star-rating item=song rating=song.rating on-click=(action "updateRating")}}
+       {{#star-rating item=song rating=song.rating as |stars|}}
+         {{#liquid-bind stars as |liquidStars|}}
+           {{#each liquidStars as |star|}}
+              <a href="#" {{action "updateRating" song star.rating}}
+                 class="star-rating glyphicon {{if star.full 'glyphicon-star' 'glyphicon-star-empty'}}">
+              </a>
+           {{/each}}
+         {{/liquid-bind}}
+       {{/star-rating}}
      </li>
    {{/each}}
  </ul>
```

The `updateRating` action handler would also have to change to take positional parameters:

```
  // app/controllers/bands/band/songs.js
  export default Controller.extend({
    (...)
    actions: {
      (...)
-     updateRating: function(params) {
-       var song = params.item,
-           rating = params.rating;
+     updateRating: function(song, rating) {
        if (song.get('rating') === rating) {
          rating = null;
        }
        song.set('rating', rating);
        return song.save();

      }
    }
  });
```

What if one wants to use the Font Awesome icons, instead of the glyphicons provided by Bootstrap? No problem there:

```
<!-- app/templates/bands/band/songs.hbs -->
<ul class="list-group songs">
  {{#each sortedSongs as |song|}}
    <li class="list-group-item song">
      {{capitalize song.title}}
      {{#star-rating item=song rating=song.rating as |stars|}}
        {{#each stars as |star|}}
          <a href="#" {{action "updateRating" song star.rating}}
             class="star-rating glyphicon {{if star.full 'fa-star'
'fa-star-o'}}">
          </a>
        {{/each}}
      {{/star-rating}}
    </li>
  {{/each}}
</ul>
```

For that to work, the Font Awesome CSS asset needs to be loaded. The simplest way to do that is to hot-link it from a CDN:

```html
<!-- app/index.html -->
<!DOCTYPE html>
<html>
  <head>
    (...)
    <link rel="stylesheet" href="assets/vendor.css">
    <link rel="stylesheet" href="assets/rarwe.css">
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
font-awesome/4.4.0/css/font-awesome.min.css">
  </head>
  (...)
</html>
```

Now that the component can also return the stars without adding any markup, the possibilities are endless.

Publishing the addon

Safe in the knowledge that our component does as intended and can be used for several use cases, we can now publish it and let others share the enjoyment. However, that will not work for you in the case of `ember-cli-star-rating`, as I had already published the package.

However, if (when) you write your own Ember addon, publishing it will be a matter of publishing the npm package (remember, Ember addons are standard npm packages):

```
$ npm version 0.0.1 # replace with correct version
$ npm publish
$ git push origin master
$ git push --tags
```

Using a published addon from the app

Now that our addon is published, we can stop using its local version and install the published one. First, let's revert the linking from the consuming application:

```
$ npm unlink ember-cli-star-rating
```

(If you use yarn, type `yarn link`).

And then remove the `*` dependency in `package.json`:

```
  // package.json
  {
    "name": "rarwe",
    (...)
    "devDependencies": {
      (...)
-     "ember-cli-star-rating": "*"
    }
  }
```

We now have a clean slate and can install the addon via ember (and ultimately, npm):

```
$ ember install ember-cli-star-rating
```

Which also saves it as a dependency of our project:

```
  // package.json
  {
    "name": "rarwe",
    (...)
    "devDependencies": {
      (...)
+     "ember-cli-star-rating": "0.5.0",
      (...)
    }
  }
```

After a server restart, everything should work as before.

The road says

"Dude, you ...". Yeah, I know, I need to clean up after myself. I left the `songs` template using Font Awesome icons. The simplest thing to do is to revert the rating stars to their original way, using them in their non-block form:

```
  <!-- app/templates/bands/band/songs.hbs -->
  <ul class="list-group songs">
    {{#each sortedSongs as |song|}}
      <li class="list-group-item song">
        {{capitalize song.title}}
-       {{#star-rating item=song rating=song.rating as |stars|}}
-         {{#each stars as |star|}}
-           <span class="star-rating fa
-                 {{if star.full 'fa-star' 'fa-star-o'}}"
-                 {{action "updateRating" song star.rating}}>
-           </span>
-         {{/each}}
-       {{/star-rating}}
+       {{star-rating item=song rating=song.rating on-click=(action "updateRating")}}
      </li>
    {{/each}}
  </ul>
```

Also, the `updateRating` action needs to be reverted to its previous form, taking a `params` object:

```
  // app/controllers/bands/band/songs.js
  export default Controller.extend({
    (...)
    actions: {
      (...)
-     updateRating: function(song, rating) {
+     updateRating: function(params) {
+       var song = params.item,
+           rating = params.rating;
        if (song.get('rating') === rating) {
          rating = null;
        }
        song.set('rating', rating);
        return song.save();
      }
    }
  });
```

Furthermore, let's remove the css style rule for the component:

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

Let's also not forget to remove the Font Awesome CSS from `app/index.html`:

```html
<!-- app/index.html -->
<!DOCTYPE html>
<html>
  <head>
    (...)
    <link rel="stylesheet" href="assets/vendor.css">
    <link rel="stylesheet" href="assets/rarwe.css">
-   <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
font-awesome/4.4.0/css/font-awesome.min.css">
  </head>
  (...)
</html>
```
HTML

Next song

We have made our app slicker by moving out our star-rating component into an addon, making it more versatile in the process. Let's make it slicker (and, perhaps more importantly, more readable) still by writing modern JavaScript, using ES2015 features.

CHAPTER 17

ES2015 - Writing modern JavaScript

Intro

ES2015, formerly known as ES6, is the next version of EcmaScript (colloquially known as JavaScript). It is a set of features, some of which have already been implemented in most browsers, but not all. That means that if we want these advanced language constructs without worrying about cross-browser compatibility, we need to use a transpiler, a language compiler where the source and target languages are the same (in our case, JavaScript).

Babel is the most popular transpiler for ES6 (and actually, some ES7) features and fortunately for us, Ember CLI uses Babel to process the source code of our Ember applications. Consequently, we can indulge in the pleasure of using the next generation of JavaScript, without looking at browser compatibility tables or resorting to polyfills, and that's exactly what we'll do in this chapter.

We'll look at certain ES2015 features that make sense for us to use in our app, and show how our code can be rewritten.

Modernizing the app

Modules

One ES2015 feature we already use in our applications is modules. We imported the dependent modules via `import ... from <module-name>` and `import { ... } from <module-name>` and used `export default ...` and `export ...` to establish what other modules can use from the modules of our app. I go into some more detail about this in the Templates and Data bindings chapter, so flip back there if you want a refresher.

Template strings

Another feature already present in our app is template strings which we saw used in Templates and Data bindings (template strings replaced `Ember.String.fmt`). Template strings will extrapolate a string by replacing the dynamic parts, enclosed in `${...}`, with their values:

```
>> var artist = { name: 'Eddie', homeTown: 'Seattle' };
>> var s = `${artist.name} lives in ${artist.homeTown}`;
// s = "Eddie lives in Seattle"
```

Short-hand method definitions

Short hand method definitions are about dropping the `function` keyword to make defining functions more concise. The transformation is very straightforward, so let me give you an example. In the `bands.band` route, we currently see this:

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

export default Route.extend({
  model: function(params) {
    return this.store.findRecord('band', params.id);
  },
  (...)
});
```

, which we can replace by the following:

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

export default Route.extend({
- model: function(params) {
+ model(params) {
    return this.store.findRecord('band', params.id);
  }
  (...)
});
```

All we did was to remove the `function` keyword and the `:`.

Replacing var (let & const)

`var` was the only way to define variables in JavaScript versions prior to ES2015. `let` and `const` are the new way to do this, and their common trait is that they are block-scoped vs. function-scoped. A block can be a function, the content of an `if` or a `for` loop, and many others.

`let` will throw a "duplicate declaration" error if you attempt to redeclare the same variable in the same block.

```
function() {
  let x = 0;
  (...)
  let x = 1; // => Line 4: Duplicate declaration: "x"
}
```

`const` is even stricter and will raise if you attempt to reassign another value to it. Let's rewrite the `updateRating` method in the `songs` controller, trying to use `const` for rating:

```js
  // app/controllers/bands/band/songs.js
  import Controller from '@ember/controller';

  export default Controller.extend({
    (...)
    actions: {
      updateRating(params) {
-       var song = params.item;
-       var rating = params.rating;
+       let song = params.item;
+       const rating = params.rating;

        if (song.get('rating') === rating) {
          rating = null;
        }
        song.set('rating', rating);
        return song.save();
      }
    }
  });
```

The error that we get is as follows:

```
File: rarwe/controllers/bands/band/songs.js
rarwe/controllers/bands/band/songs.js: Line 56: "rating" is read-only
SyntaxError: rarwe/controllers/bands/band/songs.js: Line 56: "rating" is read-only
  54 |
  55 |       if (song.get('rating') === rating) {
> 56 |         rating = null;
     |         ^
  57 |       }
```

Turning the `const` into a `let` makes this error go away:

```
  // app/controllers/bands/band/songs.js
  export default Controller.extend({
    (...)
    actions: {
      updateRating(params) {
        let song = params.item;
-       const rating = params.rating;
+       let rating = params.rating;

        if (song.get('rating') === rating) {
          rating = null;
        }
        song.set('rating', rating);
        return song.save();
      }
    }
  });
```

It is important to note that `const` just guards against assigning another value to the variable, it does not prevent the value from being mutated (in other words, it does not make it immutable):

```
const artists = [];
artists.push('Eddie'); // => No errors, artists is now ["Eddie"]
```

Almost all variables turn out to be easily convertible to `const`. In the case of this app, the above is the only `let` I needed to use in the `app` directory (and one instance in the `tests` folder).

Arrow functions

Arrow functions work around the need to save the outer context in a separate variable and introduce a more concise function definition at the same time. Let's see this in practice for the `matchingSongs` property (assuming we already converted the `searchTerm` variable to use `const` instead of `var`):

```js
// app/controllers/bands/band/songs.js
export default Controller.extend({
  (...)
  matchingSongs: computed('model.songs.@each.title', 'searchTerm',
function() {
    let searchTerm = this.get('searchTerm').toLowerCase();
    return this.get('model.songs').filter(function(song) {
      return song.get('title').toLowerCase().indexOf(searchTerm) !==
-1;
    });
  }),
  (...)
});
```

JS

The callback function of the `filter` could be converted to the arrow function form and thus the `let searchTerm` declaration can be moved inside the function, closer to where it's needed. That is possible because the arrow function retains the context (the `this`) of where it's used:

```js
// app/controllers/bands/band/songs.js
export default Controller.extend({
  (...)
  matchingSongs: computed('model.songs.@each.title', 'searchTerm',
function() {
    return this.get('model.songs').filter((song) => {
      let searchTerm = this.get('searchTerm').toLowerCase();
      return song.get('title').toLowerCase().indexOf(searchTerm) !==
-1;
    });
  }),
  (...)
});
```

JS

Be mindful of where you use the arrow function, though. Some functions explicitly set the context in their callback function and in these cases, you do not want to clobber it with the arrow function. One example would be to convert the computed property definition at the top-level:

```js
// app/controllers/bands/band/songs.js
export default Controller.extend({
  (...)
  matchingSongs: computed('model.songs.@each.title', 'searchTerm', ()
=> {
    return this.get('model.songs').filter((song) => {
      let searchTerm = this.get('searchTerm').toLowerCase();
      return song.get('title').toLowerCase().indexOf(searchTerm) !==
-1;
    });
  }),
  (...)
});
```

The result would be that the `this` in `this.get('model.songs')` would be undefined as it would save and use the `this` from the outer scope, the `this` when the class object is processed.

Destructuring

Destructuring can extract certain parts of non-scalar variables, arrays and objects, and assign them to new variables in a concise fashion. Let's see some examples with arrays:

```
>> x = [1, 2, 3, 4];
>> [first, second] = x // first = 1, second = 2
>> [head, ...rest] = x // head = 1, rest = [2, 3, 4]
>> [, , third, forth] = x // third = 3, forth = 4
```

Destructuring is possibly even more powerful with objects, as you can extract parts of a nested structure:

```
>> const eddie = {
  name: 'Eddie Vedder', address: {
    street: '24, Maple Street',
    zip: 98101,
    city: 'Seattle',
    country: 'US'
  },
  age: 50
};
>> let { name, address: { city }, age } = eddie;
// name = "Eddie Vedder", city = "Seattle", age: 50
```

Or even rename variables you assign the extracted parts to:

```
>> let { name, address: { city: homeTown }, age } = eddie;
// name = "Eddie Vedder", homeTown = "Seattle", age: 50
```

Let's see how to make that work for the `updateRating` action handler:

```
  // app/controllers/bands/band/songs.js
  export default Controller.extend({
    (...)
    actions: {
      updateRating(params) {
-       let song = params.item;
-       let rating = params.rating;
+       let { item: song, rating } = params;

        if (song.get('rating') === rating) {
          rating = null;
        }
        song.set('rating', rating);
        return song.save();
      }
    }
  });
```

JS

So `params.item` will be assigned to the `song` variable, while `params.rating` will be assigned to `rating`. Neat, isn't it?

Other useful ES2015 features

There are lots of other ES2015 features that can make JavaScript code more concise, robust, easier to read (or write), or all of these. One of my favorites (if I may) is rest arguments in function definitions:

```
>> function bandBio(name, ...members) {
  return `The members of ${name} are ${members.join(', ')}`;
};
>> bandBio('Them Crooked Vultures', 'Josh Homme', 'John Paul Jones',
'Dave
Grohl', 'Alain Johannes');
// "The members of Them Crooked Vultures are Josh Homme, John Paul
Jones,
Dave Grohl, Alain Johannes"
```

If you are interested to learn more about ES2015 features, I recommend you to browse the Learn ES2015 page on the babel.js website.

BACKSTAGE

EMBER-WATSON

`ember-watson` is a great Ember addon that can save you a ton of time by making it do the mechanical code transformations instead of you doing it one by one. Let's install it into our codebase by running the following:

$ npm install ember-watson@latest --save-dev

It has a handful of commands, among which the most relevant to us now is `methodify`, which applies the short-hand method definition conversion (see above) to our functions. Let's run it:

$ ember watson:methodify

Now all the methods in our codebase use the ES2015 method definition.

Computed property macros

We'd already talked about computed property macros (CPMs, for short) in the Sorting and searching with query params chapter. Their gist is that they cut down on the amount of code you would write to define computed properties for common use cases, like filtering, mapping, sorting or defining the emptiness of an array. Even more importantly, they make the code less prone to errors since they save you from mistyping the name of dependent keys.

Since in this chapter we are all about conciseness and robustness, let's see where we could employ CPMs.

The first such instance is the boolean flag that decides whether the button to create a new band is disabled:

```js
// app/controllers/bands.js
import Controller from '@ember/controller';
import { isEmpty } from '@ember/utils';
import { computed } from '@ember/object';

export default Controller.extend({
  name: '',

  isAddButtonDisabled: computed('name', function() {
    return isEmpty(this.get('name'));
  })
});
```

The `@ember/object/computed` module contains all the macros available.

Taking a look we see there is an `empty` macro which returns true if and only the named property is null, undefined, empty string, empty array or empty function. That suits us perfectly and we can simplify our code likewise:

```
  // app/controllers/bands.js
  import Controller from '@ember/controller';
- import { isEmpty } from '@ember/utils';
- import { computed } from '@ember/object';
+ import { empty } from '@ember/object/computed';

  export default Controller.extend({
    name: '',
-   isAddButtonDisabled: computed('name', function() {
-     return isEmpty(this.get('name'));
-   }),
+   isAddButtonDisabled: empty('name'),
  });
```

The disabled property of the button also exists for songs, so popping open the `songs` controller, we can make the same change there:

```
  // app/controllers/bands/band/songs.js
- import { isEmpty } from '@ember/utils';
- import { sort } from '@ember/object/computed';
+ import { sort, empty } from '@ember/object/computed';

  export default Controller.extend({
    (...)
-   isAddButtonDisabled: computed('title', function() {
-     return isEmpty(this.get('title'));
-   }),
+   isAddButtonDisabled: empty('title'),
  });
```

We can do some more simplification for `canCreateSong` in the same controller. Remember, this is the property that determines whether to show the song creation input for the user:

```js
// app/controllers/bands/band/songs.js
export default Controller.extend({
  (...)
  isAddButtonDisabled: empty('title'),

  canCreateSong: computed('songCreationStarted', 'model.songs.[]', function() {
    return this.get('songCreationStarted') || this.get('model.songs.length');
  }),
});
```

`canCreateSong` is the result of an OR operation of two properties. To make the code more explicit, let's first define a boolean property for `model.songs.length` and then OR the two properties together:

```js
  // app/controllers/bands/band/songs.js
- import { sort, empty } from '@ember/object/computed';
+ import { sort, empty, bool, or } from '@ember/object/computed';

  export default Controller.extend({
    (...)
    isAddButtonDisabled: computed.empty('title'),

-   canCreateSong: computed('songCreationStarted', 'model.songs.[]', function() {
-     return this.get('songCreationStarted') || this.get('model.songs.length');
-   }),
+   hasSongs: bool('model.songs.length'),
+   canCreateSong: or('songCreationStarted', 'hasSongs'),
  });
```

This is now extremely clear, it almost reads like English text.

BACKSTAGE

EMBER-CPM

There are a handful of computed property macros shipped with the framework , but we might come across situations where we need a macro for something that is not in there. The `ember-cpm` addon caters for this case by providing a few macros that cover common needs (like `sumBy`, `firstPresent` or `join`).

It can be installed as an Ember addon:

```
> ember install ember-cpm
```

Next song

We now really have an app that sings. Errors are harder to make and, in some cases, already crop up at compile time and the code is more readable and shorter. It still only lives on our machine, though. For a serious, production-ready application, this is inadmissible, so we'll take some time to learn about how to deploy it in the next chapter.

CHAPTER 18

Deployment

Tuning

We would like others to be able to use our app on the web which means we need to deploy it. In this chapter we're going to see three ways (and three different platforms) to achieve that. These three platforms are Surge, Heroku and AWS.

Surge

One of the simplest way to put our Ember app live is by using surge, a platform for static sites.

To facilitate using surge, let's install the `ember-cli-surge` addon:

```
$ ember install ember-cli-surge
```

(If you don't yet have the surge npm package installed, install it with `npm install -g surge` - or `yarn global add surge`).

Once the addon is installed, log in to surge from your project's directory:

```
$ surge login
```

The addon install placed a CNAME file in the root of our app that contains the domain our app will be available on surge.

By default, the domain is .surge.sh, so in this case, `rarwe.surge.sh`. I've set up the `app.rockandrollwithemberjs.com` domain to point to surge, as described in the docs.

I've then modified the CNAME file:

```
// CNAME
- rarwe.surge.sh
+ app.rockandrollwithemberjs.com
```

One thing we have to watch out for before we expect our app to work in production is the ajax requests our app sends. In development, they are fired to the same host (in other words, no host is specified) and we use a proxy, provided by Ember CLI, to tunnel those requests to the API.

When we deploy the app to production, no such proxy will be available and we thus need to define the right host. In Ember Data, the architecture piece that specifies where requests are sent is called the adapter. Defining a host for the top-level, application adapter will ensure that all requests sent through ED will use that setting.

So let's generate the application adapter:

```
$ ember generate adapter application
```

The `host` setting of the adapter needs to be set to `http://json-api.rockandrollwithemberjs.com`, but only in production as we want to keep using the `--proxy` option in development. Consequently, the API host setting needs to be configured:

```js
// config/environment.js

module.exports = function(environment) {
  var ENV = {
    modulePrefix: 'rarwe',
    environment: environment,
    rootURL: '/',
    (...)
  },

  if (environment === 'production') {
+   ENV.apiHost = 'http://json-api.rockandrollwithemberjs.com';
  }

  return ENV;
}
```

The `apiHost` setting can now be used to configure the application adapter:

```js
// app/adapters/application.js
import DS from 'ember-data';
import ENV from 'rarwe/config/environment';

export default DS.JSONAPIAdapter.extend({
  host: ENV.apiHost
});
```

(`ENV.apiHost` will be `undefined` in all other environments and thus the proxy option will keep working in the development environment.)

We can now go ahead and deploy our application:

```
$ ember surge --environment production

Built project successfully. Stored in "dist".

    Login to surge.sh!

    Surge - surge.sh

              email: balint.erdi@gmail.com
              token: *****************
       project path: dist
               size: 14 files, 1.2 MB
             domain: app.rockandrollwithemberjs.com

             upload: [====================] 100%, eta: 0.0s
   propagate on CDN: [====================] 100%
               plan: Free
              users: balint.erdi@gmail.com
         IP Address: 45.55.110.124

    Success! Project is published and running at
app.rockandrollwithemberjs.com
```

Indeed, the Rock and Roll app is now functional at app.rockandrollwithemberjs.com.

Heroku

Deploying to Heroku is another option that is very easy to start with.

You have to install the Heroku toolbelt so that you have the `heroku` command and you need to have a Heroku account but no further setup is needed on your machine.

Deploying Ember apps to Heroku works by way of specifying a Heroku "buildpack", a collection of scripts that gets run upon each deployment. We need to create a new Heroku application and define the buildpack that will build our app during the deployment process:

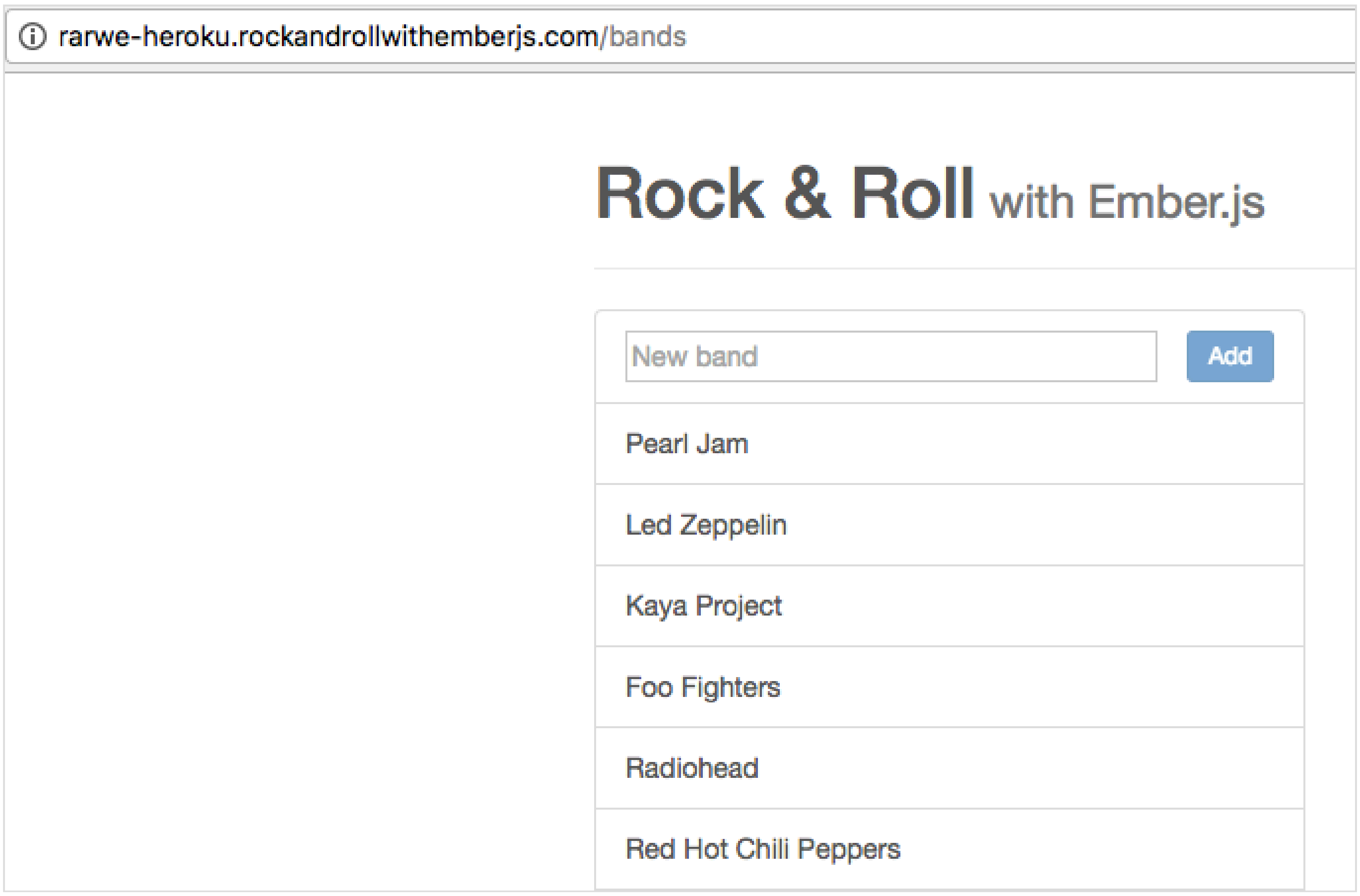
```
$ heroku create
Creating app... done, ? stark-island-60913
https://stark-island-60913.herokuapp.com/ | https://git.heroku.com/
stark-island-60913.git
$ heroku buildpacks:set https://codon-buildpacks.s3.amazonaws.com/
buildpacks/heroku/emberjs.tgz
Buildpack set. Next release on stark-island-60913 will use
https://codon-buildpacks.s3.amazonaws.com/buildpacks/heroku/
emberjs.tgz.
Run git push heroku master to create a new release using this
buildpack.
```

Deploying to the Heroku platform is a matter of pushing to the master branch of the remote called heroku:

```
$ git push heroku master
Counting objects: 484, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (272/272), done.
Writing objects: 100% (484/484), 55.86 KiB | 0 bytes/s, done.
Total 484 (delta 224), reused 332 (delta 153)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: > emberjs app detected
remote: > Setting NPM_CONFIG_PRODUCTION to false to install ember-cli toolchain
remote: > Fetching buildpack heroku/nodejs
remote: > Node.js detected
remote:
remote: > Creating runtime environment
remote:
remote:        NPM_CONFIG_LOGLEVEL=error
remote:        NPM_CONFIG_PRODUCTION=false
remote:        NODE_VERBOSE=false
remote:        NODE_ENV=production
(...)
remote: > Caching ember assets
remote: > Writing static.json
remote: > Fetching buildpack heroku/static
remote: > Static HTML detected
remote:   % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
remote:                                  Dload  Upload   Total   Spent    Left  Speed
remote: 100  838k  100  838k    0     0  14.5M      0 --:--:-- --:--:-- --:--:-- 14.6M
remote: > Installed directory to /app/bin
remote: > Discovering process types
remote:        Procfile declares types     -> (none)
remote:        Default types for buildpack -> web
remote:
```

```
remote: > Compressing...
remote:         Done: 48M
remote: > Launching...
remote:         Released v3
remote:         https://stark-island-60913.herokuapp.com/ deployed to
Heroku
remote:
remote: Verifying deploy... done.
```

Our app is now live on Heroku:



rarwe-heroku.rockandrollwithemberjs.com/bands

Rock & Roll with Ember.js

Pearl Jam
Led Zeppelin
Kaya Project
Foo Fighters
Radiohead
Red Hot Chili Peppers

As you can see both the installation of node, npm and nginx and the building of the app happens during deployment, on the Heroku platform. Consequently, the whole process can take several minutes, somewhat less on subsequent pushes as the installed npm packages are cached for later deployments. Pushing to Heroku is still a simple, though definitely not the fastest, way to expose your app to the greater public.

ember-cli-deploy

ember-cli-deploy provides a framework for deploying Ember apps just as liquid-fire sets does for animations or torii does for authentication and authorization.

In the light of this, it is hardly surprising that ember-cli-deploy is the most complex, but also the most flexible, of the deployment solutions in this chapter. Using a couple of abstractions and favoring extensibility, it enables Ember apps to be deployed to all kinds of platforms & hosting solutions.

It splits the deployment process into two parts: deploying the container of the application (in other words, the index.html page), and deploying the assets (scripts, stylesheets, images, fonts, etc.). The basic insight is that even though the index page refers to the assets, these two pieces can be deployed independently, even on different platforms.

The architectural blocks that implement these deployment steps are called "index adapters" and "asset adapters", respectively. The framework nature of this project makes it so that implementing the deployment of assets (or the main page) to a certain platform means implementing an asset adapter (or index adapter). There are a handful of adapters already written by the community.

I'll go through the setting up of deploying both the index page and the assets to S3, as this is a relatively simple and popular choice.

Deploying our app to S3

Let's start by installing the base package, `ember-cli-deploy`:

```
$ ember install ember-cli-deploy
  installing ember-cli-deploy
  create config/deploy.js
ember-cli-deploy needs plugins to actually do the deployment work.
See http://ember-cli-deploy.github.io/ember-cli-deploy/docs/v0.5.x/
quick-start/
to learn how to install plugins and see what plugins are available.
Installed addon package.
```

The installer reminds us that ember-cli-deploy is just the wireframe and we need to install plugins for the whole deployment process to be functional.

We'll need to install the following plugins:

```
$ ember install ember-cli-deploy-build
$ ember install ember-cli-deploy-revision-data
$ ember install ember-cli-deploy-display-revisions
$ ember install ember-cli-deploy-s3
$ ember install ember-cli-deploy-s3-index
```

Now, let's look inside the generated configuration file:

```js
// config/deploy.js
module.exports = function(deployTarget) {
  var ENV = {
    build: {}
  };

  if (deployTarget === 'development') {
    ENV.build.environment = 'development';
  }

  if (deployTarget === 'staging') {
    ENV.build.environment = 'production';
  }

  if (deployTarget === 'production') {
    ENV.build.environment = 'production';
  }

  return ENV;
};
```

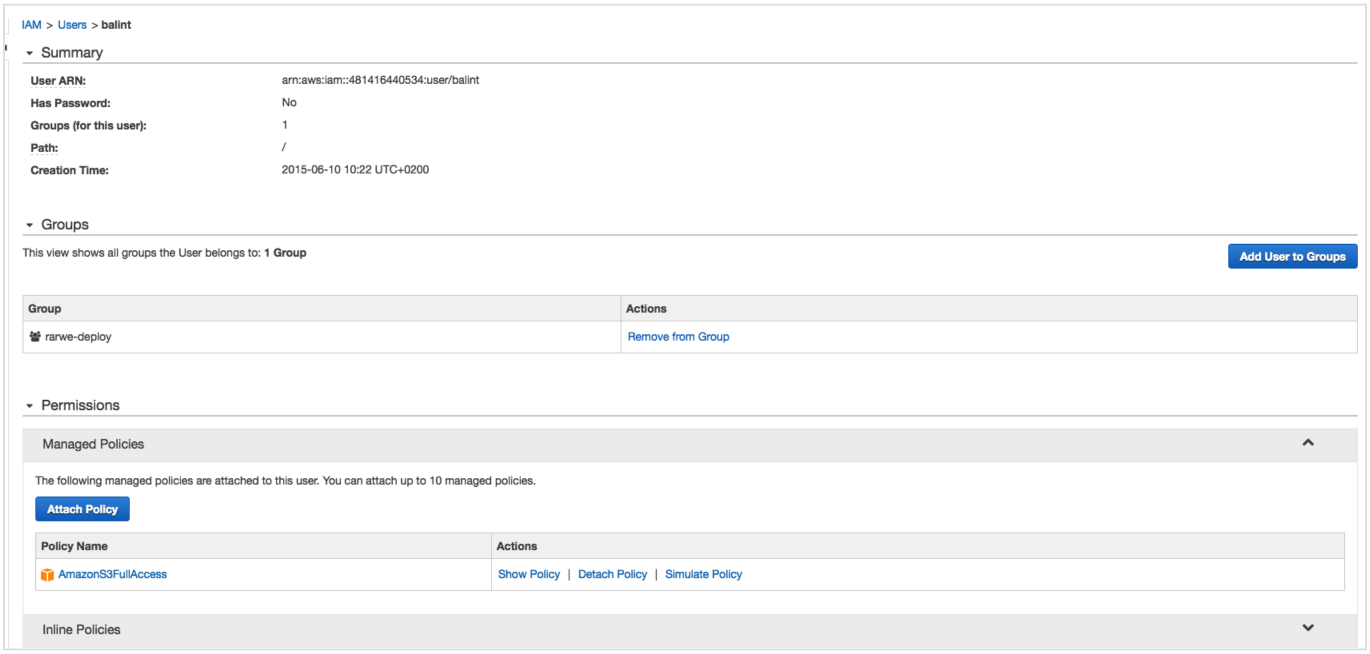
JS

As we can get away with some cowboy coding (and deployment) on this project, we'll push directly to production and not care about the `development` and `staging` environments:

```js
  // config/deploy.js
  module.exports = function(deployTarget) {
    var ENV = {
      build: {}
    };

    if (deployTarget === 'development') {
      ENV.build.environment = 'development';
    }

    if (deployTarget === 'staging') {
      ENV.build.environment = 'production';
    }

    if (deployTarget === 'production') {
      ENV.build.environment = 'production';
+     ENV.s3 = {
+       accessKeyId: process.env['AWS_PERSONAL_IAM_BALINT_ACCESS_KEY_ID'],
+       secretAccessKey: process.env['AWS_PERSONAL_IAM_BALINT_SECRET_ACCESS_KEY'],
+       bucket: 'rarwe-production',
+       region: 'us-east-1'
+     };
+     ENV['s3-index'] = {
+       accessKeyId: process.env['AWS_PERSONAL_IAM_BALINT_ACCESS_KEY_ID'],
+       secretAccessKey: process.env['AWS_PERSONAL_IAM_BALINT_SECRET_ACCESS_KEY'],
+       bucket: 'rarwe-production',
+       region: 'us-east-1'
+     };
    }
    return ENV;
  };
```

JS

In my AWS console, I created a new bucket (`rarwe-production`) and made it publicly viewable for everyone so that the assets can be publicly served. To be able to deploy the files, I created an IAM user role and gave the AmazonS3FullAccess to that role. The access key id and secret key for that role are set as environment variables and used in the config above.



IAM > Users > balint

Summary

User ARN:	arn:aws:iam::481416440534:user/balint
Has Password:	No
Groups (for this user):	1
Path:	/
Creation Time:	2015-06-10 10:22 UTC+0200

Groups

This view shows all groups the User belongs to: **1 Group**

Group	Actions
rarwe-deploy	Remove from Group

Permissions

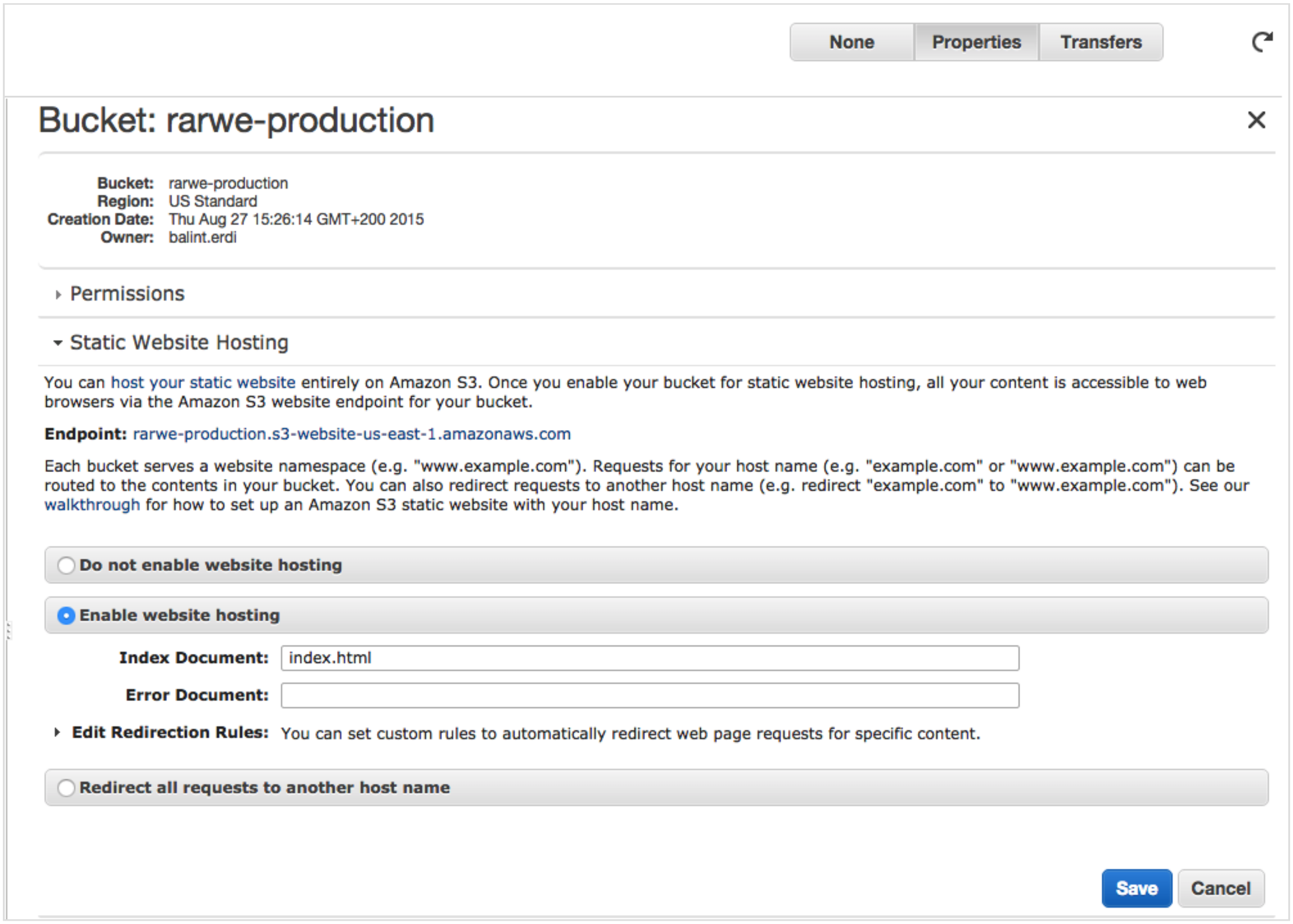
Managed Policies

The following managed policies are attached to this user. You can attach up to 10 managed policies.

Policy Name	Actions
AmazonS3FullAccess	Show Policy | Detach Policy | Simulate Policy

Inline Policies

On top of that, in order for S3 to serve the index.html file for the root of our application, static web hosting needs to be enabled for the bucket:



Bucket: rarwe-production

Bucket:	rarwe-production
Region:	US Standard
Creation Date:	Thu Aug 27 15:26:14 GMT+200 2015
Owner:	balint.erdi

Permissions

Static Website Hosting

You can host your static website entirely on Amazon S3. Once you enable your bucket for static website hosting, all your content is accessible to web browsers via the Amazon S3 website endpoint for your bucket.

Endpoint: rarwe-production.s3-website-us-east-1.amazonaws.com

Each bucket serves a website namespace (e.g. "www.example.com"). Requests for your host name (e.g. "example.com" or "www.example.com") can be routed to the contents in your bucket. You can also redirect requests to another host name (e.g. redirect "example.com" to "www.example.com"). See our walkthrough for how to set up an Amazon S3 static website with your host name.

○ Do not enable website hosting

● Enable website hosting

Index Document: index.html

Error Document:

Edit Redirection Rules: You can set custom rules to automatically redirect web page requests for specific content.

○ Redirect all requests to another host name

We now stand ready to deploy the app in production:

```
$ ember deploy --environment production

Deploying [====] 100% [plugin: s3-index -> fetchRevisions]
ember deploy production  20.48s user 1.73s system 77% cpu 28.484 total
```

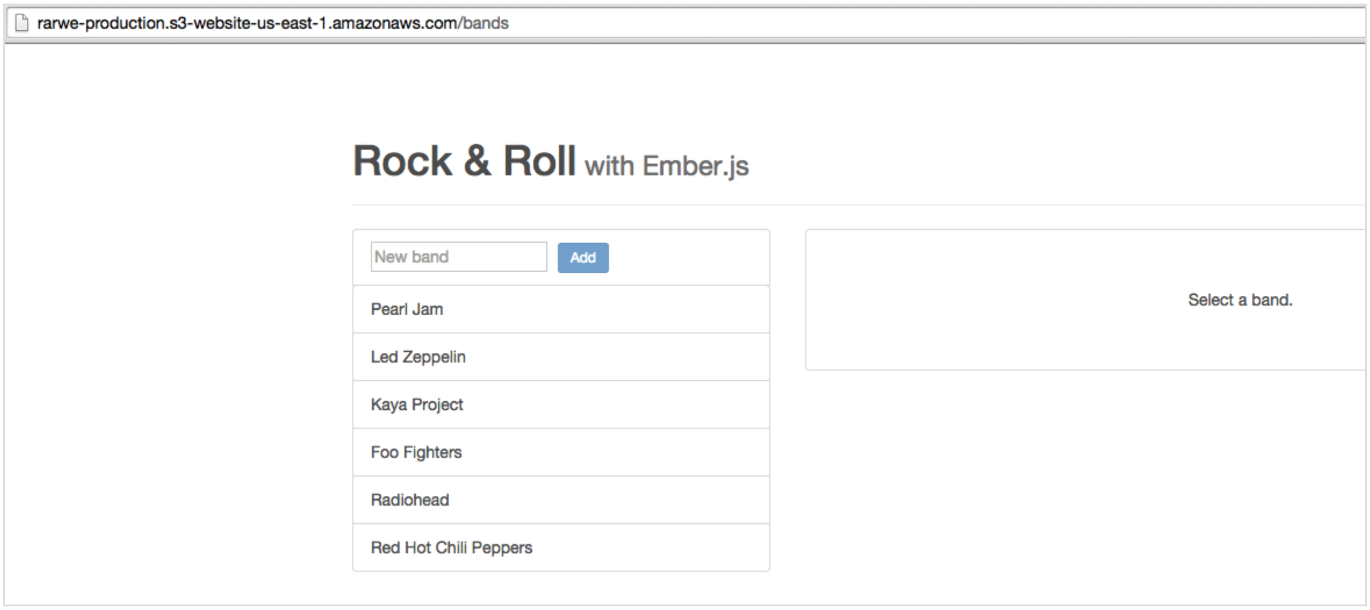
We are not done yet, as we've only deployed a new version but haven't activated it. The `deploy:list` command lists the deployed versions:

```
$ ember deploy:list production
-   timestamp           | revision
- =================================
-   2016/09/16 11:10:10 | 90619f13fa8a91a251ca8ef8376e87b8
```

Armed with the revision number, we can use `deploy:activate` to make the new version the active one:

```
$ ember deploy:activate production --revision
90619f13fa8a91a251ca8ef8376e87b8
- ✔  index.html:90619f13fa8a91a251ca8ef8376e87b8 => index.html
```

With that, our app is running on Amazon's infrastucture:



Next song

That concludes the first set of features for the Rock & Roll application. The CEO has just informed me that our seed funding has run out, so I have to stop development immediately.

We might be out of funds, but hopefully fueled by enthusiasm, so let me conclude the book with a brief message that you'll find in the Afterword.

Afterword

Your journey with Ember.js

Our journey, for now, ends here. We have come a long way, and I hope that you now possess the necessary knowledge to approach the development of Ember applications with confidence.

Equally importantly, I hope you have caught the enthusiasm for Ember I felt when I was learning the ropes, and that I still feel today.

Whatever you end up using Ember for, or even if your journey takes you elsewhere, I would love to hear from you. Please also feel free to get in touch with any questions or comments related to the book.

Encore

Creating your application with Yarn

Ember CLI was made "yarn-aware" from version 2.13 and has full support for the blazing-fast package manager. You can use yarn from the get-go by passing an extra flag when you create your application.

So go to the folder where you would like to store your 'Rock & Roll with Ember.js' application and create a new Ember CLI project passing the `--yarn` option:

```
$ ember new rarwe --no-welcome --yarn
```

A `yarn.lock` file will be created in your Ember project and from this point on, all add-on installs will leverage yarn.

If you started your project using npm but want to switch to yarn, just do

```
$ yarn install
```

, which creates the lockfile. Ember CLI will detect you want to use yarn subsequently, and will comply.

You can read more about yarn at https://yarnpkg.com/.

If you came here from the Ember CLI chapter, you can now go back and continue with the "Taking a look at a new project" section.

ES2015 modules

Ember CLI uses Babel which allows us to use the module definition syntax of the next version of JavaScript, ECMAScript 2015 (ES2015, for short). These modules are transpiled to ES5 so they can be run in browsers of today that do not yet fully support ES2015.

Each ES2015 module can export one or several things that other modules can import. This thing (or things) is what the module shares with the outside world. Not being able to access things that are internal to the module is a good thing, since it provides encapsulation and increases robustness.

The simplest module exports one thing only by defining a so-called 'default export'. This is achieved by prefixing the thing-to-be-exported by the words `export default`:

```js
// lib/fibonacci.js

export default function fibonacci(n) {
}
```

When another module needs what the fibonacci module exports, it accesses it via an import statement:

```js
// tests/fibonacci-test.js

import fib from './fibonacci';

console.log("Did I get this right? " + (fib(3) === fib(1) + fib(2)) ?
"YES!" :
"Nope");
(...)
```

Note that the name of the imported function in the exporting module does not matter. What matters is the name I choose to import it with. It was given the name `fibonacci` in its exporting module but I imported it as `fib` by writing `import fib from './fibonacci'` which makes it available as `fib`. I could have also written `import f from './fibonacci'` and then use it as `f` in the `tests/fibonacc-test.js` file.

You can also export several things by defining 'named exports'. Since we will only use default exports in this book, I won't go into detail here. If you are interested in learning about them, I recommend Axel Rauschmayer's excellent summary on ES2015 modules.

Class and instance properties

Above, I defined `title: '', rating: 0, band: ''` at the class level of `Song`. You may wonder what happens exactly when properties are defined at the class level, so let me explain it by way of an example:

```js
import EmberObject from '@ember/object';

var Person = EmberObject.extend({
  name: 'Noname',

  intro: function() {
    console.log(`Hey, I am ${this.name}. Nice to meet you`);
  }
});

var noname = Person.create();
noname.intro(); // "Hey, I am Noname. Nice to meet you"

var amy = Person.create({ name: "Amy" });
amy.intro(); // "Hey, I am Amy. Nice to meet you"

var step = Person.create({
  name: 'Step, the Artist',
  intro: function() {
    console.log(`Hey, I am ${this.name}. I live in Berlin`);
  }
});
step.intro(); // "Hey, I am Step, the Artist. I live in Berlin"
```

A string in back-ticks is a template string, included in the next version of JavaScript, ES2015. As Ember CLI transpiles ES2015 features to the current JavaScript version, we can use them in our Ember CLI apps.

Template strings interpolate the variable bindings in the string:

```js
a = 'Alice';
b = 'Bob';
console.log(`${a} and ${b}, sitting in a tree`); // => "Alice and Bob, sitting in a tree";
```

Back to the example. As you can see, properties that are not provided to `create` will get their values from the class object (from their prototype). They thus serve as default values for all instances of the class. To give an example, if we create a song without a rating, it will have the rating defined in `Song`:

```js
Song.create({ title: 'Sirens' }).get('rating') // =>  0
```

We should keep in mind that JavaScript does not yet have built-in support for classes but for the purpose of wrapping our head around how accessing properties on Ember objects work, it is a useful concept.

EmberObject accessors

In Ember, you have to get the value of any property of an `EmberObject` (that includes routes, controllers, components, etc.), since they all descend from `EmberObject` via `Ember.get` and set it via `Ember.set` (most often seen as `obj.get('property')` and `obj.set('property', 'value')`).

The reason has to do with performance. Since Ember knows exactly when a certain property was changed (via `Ember.set`) and knows which computed properties depend on that changed property (since the developer has to define the dependent keys for each computed property) it also knows which properties need to be marked as dirty and recomputed the next time they are accessed via `Ember.get`.

This would not be the case if Ember used standard JavaScript property accessors, `obj.property` and `obj.property = 'value'`, as it would have to go through all properties of all objects to see which ones have changed since last time.

This results in quicker updates and gives Ember a performance edge.

Mixins

Mixins are a way to extract functionality (like sorting) that can then be shared across various classes.

They add properties and methods to classes that include them. They cannot be instantiated but can be included ("mixed in") to other classes to add their functionality. In some programming languages (for example, Ruby) they might be called modules.

In Ember, mixins are used very heavily. They provide a clean way to compose the feature set of a class from various components by customizing the mixin's properties and methods, and potentially adding some more. The below example demonstrates how this can be done:

```js
import EmberObject from '@ember/object';
import Mixin from "@ember/object/mixin";
import { isBlank } from "@ember/utils";

var CanSpeak = Mixin.create({
  fillingWord: '',

  speak: function(sentences) {
    var fillingWord = this.get('fillingWord');
    if (isBlank(fillingWord)) {
      console.log(sentences.join(' '));
    } else {
      console.log(sentences.join(' ' + fillingWord.capitalize() + ',
'));
    }
  }
});

var Person = EmberObject.extend(CanSpeak, {
  name: 'Noname',
  fillingWord: 'uhm',

  intro: function() {
    return this.speak([`Hey, I am ${this.name}.`, 'nice to meet
you.']);
  }
});

var Robot = EmberObject.extend(CanSpeak, {
});

var nobody = Person.create();
nobody.intro(); // => "Hey, I am Noname. Uhm, nice to meet you."

var amy = Person.create({ name: 'Amy', fillingWord: 'ah' });
amy.intro(); // => "Hey, I am Amy. Ah, nice to meet you."

```
JS

```
var robot = Robot.create();
robot.speak(['Good morning.', 'What can I help you with today?']); //
=> "Good morning. What can I help you with today?"
```

The `extend` method takes a variable number of arguments for mixins, and a final argument that contains the properties of the composed class itself. Note that the properties defined in the mixin (`fillingWord`) can be either overridden at the class-level in classes that use the mixin (see `fillingWord: 'uhm'` in `Person`) or in the individual object instances. (For a discussion of the differences, see the "Class and instance properties" above). Technically the instance - class - mixin(s) form a chain for property lookup.

Concatenated properties

As opposed to "normal" properties, concatenated properties add the passed in values to the ones defined in the class instead of overwriting them:

```
import StarRatingComponent from './components/star-rating';

var stars = StarRatingComponent.create({ classNames:
['star-rating-widget'] });
stars.get('classNames'); // => ["ember-view", "rating-panel",
"star-rating-widget"]
```

JS

The `ember-view` class comes from Ember's `Component` class which our component extends. The `rating-panel` class is defined in our "class" body while `star-rating-widget` was passed in at construction time. This demonstrates how at each level the values for `classNames` were added to the existing values instead of overwriting them.

Class names being a concatenated property implies that only CSS classes inherent to the component should be defined inside the component class, since they will be present everywhere we use the component in templates. Other classes can be passed in when using the component as necessary.

On the utility of explicit dependency definitions

By not allowing function calls to render their output in templates, and explicitly defining what the property depends on, this provides a serious advantage.

To see this, let's assume Ember did allow function calls in templates, and we would have `{{#each stars() as |star|}}` in the template. That part of the template would need to be rerendered whenever calling `stars()` would produce a different result than what is currently rendered. How do we know whether it would be different? We can't know that, and so we would have to resort to calling `stars()` and render the result in the DOM. If `stars()` is slow, we incur a performance hit, perhaps unnecessarily if the result is unchanged.

Contrast that with turning it into a property and clearly defining its dependencies. There is no guessing involved. It suffices to observe the dependent keys and rerender when any of them changes, yielding a performance boost.

The default component template

When we generate a component with Ember CLI, we can see that it has nothing but a `{{yield}}` inside.

We saw in the Nested routes chapter that the `link-to` helper could be used in two forms: block and non-block. Components have this same feature. When used in their block form, `yield` defines the slot where the calling context can render its content.

To give an example, a form component that focuses the first text input when rendered would just have a `{{yield}}` as its template and could be used something like this:

```hbs
{{#focusing-form onSubmit="register"}}
  <label for="email">Email</label>
  {{input type=text value=user.email}}
  (...)
{{/focusing-form}}
```

You can think of the possibility to yield as the possibility to customize the component's markup. Anything you put in the component's template will always be rendered for the component so you should yield to account for the differences in markup.

Fake synchronous tests

The acceptance test helpers (`visit`, `click`, `fillIn`, etc.) look like they were written for a platform where there is no concurrency. Each helper is on its own line, like `visit('/bands')`, `click('.band-link:first')`, and so on. At the same time, the browser environment is all about asynchronous operations. Clicking the first band link has to wait until all asynchronous operations triggered by the previous line, `visit('/bands')` have completed. Otherwise, the link might not even be there to click on.

The async construct that Ember favors is promises so it should come as no surprise that these test helpers return promises that the next helper has to wait on to be resolved. In fact, we could have written the first test in a form that hints at the use of promises:

```
test('Create a new band', function(assert) {
 (...)
  visit('/bands')
  .fillIn('.new-band', 'Long Distance Calling')
  .click('.new-band-button')
  .then(function() {
    assert.equal(find('.band-link').length, 2, 'All band links are rendered');
    assert.equal(find('.band-link:last').text().trim(), 'Long Distance Calling', 'Created band appears at end of the list');
    assert.equal(find('.nav a.active:contains("Songs")').length, 1, 'The Songs tab is active');
  });
});
```

JS

All of the async test helpers that Ember provides return a promise (by calling a function called `wait`) that is going to be resolved if the router finished its transition, all async operations have completed, and no run loop is running. In brief, if everything has settled and everything is rendered on screen.

Furthermore, they can each be written on their own line, making them look like standard synchronous code, because they are bound to a single promise instance, `Test.lastPromise`. Subsequent operations are chained together so that each runs only after the previous has finished. We just don't see it being set up, because it's done by code in the ember-testing package.