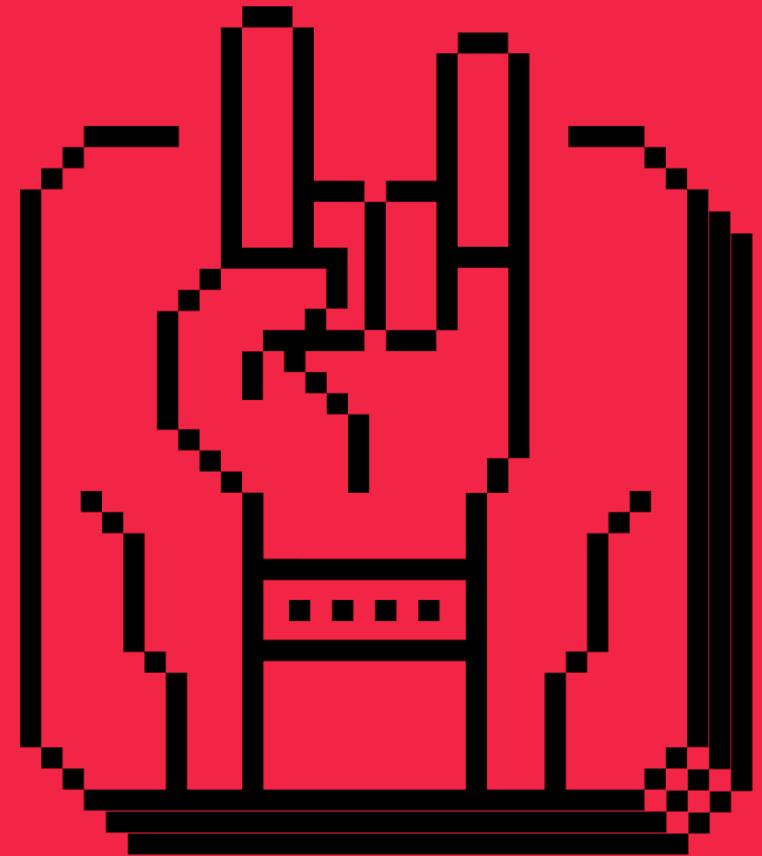


ROCK & ROLL

WITH MEMBER.JS



**BUILD AMBITIOUS APPS
WITH CONFIDENCE**

...

Written by BALINT ERDI

Set List

Preface

My love affair with Ember.js

1. Introduction to Ember.js
 2. Ember CLI
 3. Templates and data bindings
 4. Routing
 5. Nested routes
 6. Components
 7. Controllers
 8. Building a catalog
 9. Talking to a back-end
 10. Advanced routing
 11. Testing
 12. Query params
 13. Loading and error substates
 14. Helpers
 15. Deployment
- Afterword
- Appendix

Preface

The dawn of client-side frameworks

(You are reading version 20210211 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 achieved 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 a network request to the back-end 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 as you 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. Starting in the 2010s, both view-layer libraries and full-fledged frameworks have appeared that serve as single-stop, "all-parts included" toolboxes to build a complete application.

During the same period, the devices executing client-side JavaScript evolved to have much better performance. While in the 2000s, 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 tools 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 with whom I was working on a pet project 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 six months.

In February 2015, two years after my initial infatuation, I finally published the first version of the book. People liked it and kept telling me both via email and in person that it really helped them learn the framework. This kept me going and updating the book as minor versions of Ember were released on a regular schedule.

Version 2 of the book was released in October of the same year, and I've been following the steady release plan of Ember with my book updates. And roughly three years after the book saw the light of day, I published Rock and Roll with Ember.js 3.

Ember Octane was a breath of fresh air. Several old Emberisms, like the Ember.Object hierarchy and computed properties, were necessary at the time (remember, the first commit to Ember was made on April 30, 2011) but it held back the adoption of Ember by the JavaScript community. Octane brings Ember a whole lot closer to JavaScript (hello, ES6 classes!). Ember templates are quite close to html which makes it possible for designers to work independently on them, without an Ember developer needing to hold their hand.

I also feel like Octane brought invention where invention was needed, and I'm particularly excited about tracked properties, alternative component managers (like Glimmer components), and the possibility to use JavaScript classes.

Building applications with Ember, whether small or 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 first screencast went live and I sent it out to the few dozen email subscribers I had at the time, I started receiving kind emails that 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 David Toth, Scott Robinet, Joel Fuller, Kelly Selden, Pedro Cambra, Bernd Toepfer, Thore Sünert, Marten Schilstra, Gé-Jé Overschie, Gabriel Rotbart, Andy Davison, William White, Joachim Jablon, Rael Dornfest, 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, Raul Murciano, Edward Faulkner, Sam Selikoff and Miguel Camba 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, Bence Golda, Daniel Jeffery, and Isaac Lee 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, Thomas Martineau and Kelly Selden.

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 especially want to thank Godfrey Chan (aka. chancancode) for making a "rundoc"-like tool that runs commands from documentation. This new Octane book uses the same tool so that I can keep the book's text contents in sync with changes in the framework. This should come super handy when I release new versions after minor Ember updates.

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, 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 and Ember developers working on add-ons or contributing to the framework deserve huge kudos. Without you this book would obviously not exist. Thank you for your relentless (and sometimes what seems like superhuman) work.

Oh, and if you meet rwjblue (known as Robert Jackson in the real world), make sure to buy him a beer.

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 entities are tied together (for example, which template a certain route renders), Ember.js adopts **naming conventions that establish these connections**. This cuts down hugely on the amount of needed boilerplate code.

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.

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 monolithic block of code**. It leverages several **small JavaScript libraries**, like **router.js** for routing, **backburner** for managing the **run loop**, and **Glimmer** for its view layer.

The release of Glimmer.js marked the beginning of Ember moving to become **a truly modular framework**.

With the Ember.js Modules API being adopted in 2.16, and [Embroider](#) being actively worked on, the future where only the pieces of the framework used in a particular application will be shipped to the browser is not far away.

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, well-tested, 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 back-end, checking authorization to view the page, redirecting to another route, handling user actions, and many more features.

- 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 deduced by the framework. One example of this is generating the right kind of controller for a route on the fly.

4. Declarative templates and The Rule of Least Power

Ember templates are written in a declarative templating language called Handlebars. Unlike in other view libraries, you have a constrained set of helpers you can wield and, more importantly, you cannot use JavaScript directly in templates.

This conforms to the Rule of Least Power, as expressed by the W3C, which posits that to solve a given problem, one should choose the least powerful language. One of the excerpts rings especially true for view-layer languages:

"Thus, there is a tradeoff in choosing between languages that can solve a broad range of problems and languages in which programs and data are easily analyzed. Computer Science in the 1960s through 1980s spent a lot of effort making languages that were as powerful as possible. Nowadays we have to appreciate the reasons for picking not the most powerful solution but the least powerful. Expressing constraints, relationships and processing instructions in less powerful languages increases the flexibility with which information can be reused: the less powerful the language, the more you can do with the data stored in that language."

The fact that Ember.js adheres to this principle made it possible for the view engine to be completely rewritten twice already *without the templates needing to even be modified!*

With frameworks becoming optimizing compilers, this is even more the case and Ember made a very sensible choice for sticking with (an enhanced version of) Handlebars for declaring its templates.

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 team 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, six-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.

On top of that, any change to the framework now has to go through the RFC process. The RFCs are open for comments and suggestions from the community so people can voice their concerns before the suggested change makes it into the framework.

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 roadie 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 sections at the end of the chapters so that the flow of explanation is not broken by minutiae.

*"**Related hits**" sections would be "Further resources" in a book with a non-musical theme. These links help

to dig further in subtopics related to the chapter content

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

"**Appendix**" is where related, interesting concepts that are worth knowing about but don't quite fit in the flow of building the application are found.

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 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 building blocks come together to make everything work. The

roles and responsibilities of these building blocks 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 screen 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 simple flow to add a new band and then start adding songs to it.
- Add new songs
- Build our data management layer to hold band and song records and communicate with the back-end
- Hook up the application to a remote back-end API.
- Safeguard our application by writing a bunch of tests, both high- and low-level.
- Sort the list of songs by different criteria, 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.
- Show how to deploy the app onto various platforms

I encourage you to work along with 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. On a Mac, Acrobat Reader works better than the default Preview.

Code blocks

There are some **code snippets** in the book that only serve demonstrational purposes and **are not part of the application**. They can be discerned by **not having a comment header** that designates it does need to be included in the application we're building.

Here is one such code snippet:

```
1 Router.map(function() {  
2   this.route('bands');  
3   this.route('songs');  
4 });
```

js

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

```
1 // app/router.js  
2 Router.map(function() {  
3   this.route('bands');  
4   this.route('songs');  
5 });
```

js

In most instances, 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:

```
1 {{!-- app/components/star-rating.hbs --}}  
- 2 {{yield}}  
+ 3 {{#each this.stars as |star|}}  
+ 4   {{if star.full "X" "O"}}  
+ 5 {{/each}}
```

HBS

The file will be shown in its entirety which in some cases can be slightly annoying in large files – you have to page/thumb through lines that didn't change. This is due to the tool that I use to generate code diffs

(actually, the whole app) which has several advantages. I apologize if that turns out to be an inconvenience.

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 for creating and developing Ember applications. We'll then use it to create our app.

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 things. Above this, it's super fast and rebuilds and reloads your app whenever any of your development files change. 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
```

and have the respective versions printed out.

Ember CLI needs at least node version 10, so don't proceed if your installed version is 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
```

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:

```
1 $ ember --version
2 ember-cli: 3.24.0
3 node: 12.8.1
4 os: darwin x64
```

(Your node and os version strings might be different and that's fine.)

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

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` 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'd like to use Yarn, a much improved package manager for node packages, read the "Creating your application with Yarn" section in the [Appendix](#). I recommend you do this but I don't insist on it.

(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.

Go to the folder where you would like to store your application and create a new Ember CLI project:

```
1 $ ember new rarwe --no-welcome
2 installing app
3 Ember CLI v3.24.0
4
5 Creating a new Ember app in /Users/balint/code/balinterdi/emberjs/
6 rarwe-book/rarwe-octane-rundoc/dist/code/rarwe:
7   create .editorconfig
8   create .ember-cli
9   create .eslintignore
10  create .eslintrc.js
11  create .prettierignore
12  create .prettierrc.js
13  create .template-lintrc.js
14  create .travis.yml
15  create .watchmanconfig
16  create README.md
17  create app/app.js
18  create app/components/.gitkeep
19  create app/controllers/.gitkeep
20  create app/helpers/.gitkeep
21  create app/index.html
22  create app/models/.gitkeep
23  create app/router.js
24  create app/routes/.gitkeep
25  create app/styles/app.css
26  create app/templates/application.hbs
27  create config/ember-cli-update.json
28  create config/environment.js
29  create config/optional-features.json
30  create config/targets.js
31  create ember-cli-build.js
32  create .gitignore
33  create package.json
34  create public/robots.txt
35  create testem.js
36  create tests/helpers/.gitkeep
37  create tests/index.html
```

```
37  create tests/integration/.gitkeep
38  create tests/test-helper.js
39  create tests/unit/.gitkeep
40  create vendor/.gitkeep
41
42  Installing packages... This might take a couple of minutes.
43  npm: Installed dependencies
44
45  Initializing git repository.
46  Git: successfully initialized.
47
48  Successfully created project rarwe.
49  Get started by typing:
50
51  $ cd rarwe
52  $ npm start
53
54  Happy coding!
```

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](#). In a few moments, our app is ready to be developed.

Taking a look at a new project

```
$ cd rarwe
```

Let's take a look at the created files:

```
1  rarwe
2  └── app
3  |   ├── components
4  |   ├── controllers
5  |   ├── helpers
6  |   ├── models
7  |   ├── routes
8  |   ├── styles
9  |   |   └── app.css
10 |   └── templates
11 |       └── application.hbs
12 |   └── app.js
13 |   └── index.html
14 |   └── router.js
15 └── config
16 |   ├── ember-cli-update.json
17 |   ├── environment.js
18 |   ├── optional-features.json
19 |   └── targets.js
20 └── public
21 |   └── robots.txt
22 └── tests
23 |   ├── helpers
24 |   ├── integration
25 |   ├── unit
26 |   |   └── index.html
27 |   |   └── test-helper.js
28 └── vendor
29 └── .editorconfig
30 └── .ember-cli
31 └── .eslintcache
32 └── .eslintignore
33 └── .eslintrc.js
34 └── .gitignore
35 └── .prettierignore
36 └── .prettierrc.js
37 └── .template-lintrc.js
```

```
38 └── .travis.yml  
39 └── .watchmanconfig  
40 └── README.md  
41 └── ember-cli-build.js  
42 └── package-lock.json  
43 └── package.json  
44 └── testem.js  
45  
46 15 directories, 28 files
```

Starting and stopping the development server

Ember CLI comes with a lot of different commands for a variety of development tasks, such as the `ember new` command that we saw earlier. It also comes with a *development server*, which we can launch with the `ember server` command:

```
1 $ ember server  
2  
3 Build successful (14369ms) - Serving on http://localhost:4200/
```

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.

The `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.

The `config/optional-features.json` file has a list of features that are not essential to the functioning of your app and can be switched on or off here. One example flag is `jquery-integration`.

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 for your project.

The `node_modules` folder is where those 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 see a `yarn.lock` file (instead of `package-lock.json`) 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 what 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 the `Glimmer.js view engine`, which sits on top of the `Handlebars templating language`, 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 by typing `ember s` at the command line (which is short for `ember server`):

```
1 $ cd rarwe  
2 $ ember s
```

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.

We can now start getting acquainted with templates.

The first template

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

```
1 {{!!-- app/templates/application.hbs --}}
2 {{page-title "Rarwe"}}
3
- 4 <h2 id="title">Welcome to Ember</h2>
- 5
- 6 {{outlet}}
+ 7 <ul>
+ 8 {{#each
+ 9   array
+ 10  (hash title="Black Dog" band="Led Zeppelin" rating=3)
+ 11  (hash title="Yellow Ledbetter" band="Pearl Jam" rating=4)
+ 12  (hash title="The Pretender" band="Foo Fighters" rating=2)
+ 13  ) as |song|
+ 14  }}
+ 15  <li>
+ 16    {{song.title}} by {{song.band}} ({{song.rating}})
+ 17  </li>
+ 18 {{/each}}
+ 19 </ul>
```

HBS

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

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

Ember's templating engine is called [Glimmer](#) and is compatible with a subset of the [Handlebars](#) templating language. Anything enclosed in mustaches (`{{ ... }}`) is a dynamic expression. The first such expression is a helper, `{{#each}}`. It loops over the collection passed as the first argument and make each item accessible inside the block with the name given after the `as` keyword. (Technically, the removed `{{outlet}}` was the first such dynamic expression. It will be explained in [the Routing chapter](#)).

The `array` helper takes a varying number of arguments and creates an array that contains them.

The `hash` helper creates a POJO (Plain Old JavaScript Object) in template-land.

Putting these together, the `song` block variable will loop through each object created by the `hash` helper and render a list item for each of them.

We should see our quite rudimentary list of songs rendered on screen:

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

There's also a `page-title` helper call, which was put into the template when Ember CLI generated the project. It sets the document title that is shown in the browser tab. Let's make it more meaningful for our app:

```
1  {{!-- app/templates/application.hbs --}}
- 2  {{page-title "Rarwe"}}
+ 3  {{page-title "Rock & Roll with Octane"}}
4
5  <ul>
6    {{#each
7      (array
8        (hash title="Black Dog" band="Led Zeppelin" rating=3)
9        (hash title="Yellow Ledbetter" band="Pearl Jam" rating=4)
10       (hash title="The Pretender" band="Foo Fighters" rating=2)
11     ) as |song|
12   )}
13   <li>
14     {{song.title}} by {{song.band}} ({{song.rating}})
15   </li>
16   {{/each}}
17 </ul>
```

HBS

That looks better:



Next, let's spiff the list up now by adding some style.

Adding assets to the build

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

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 Tailwind assets in our application.

The first way is to use an Ember add-on suited for that library. The second is to install the npm package and then do some additional plumbing to integrate it into our app. While using an Ember add-on is the simplest way to go, it has a few drawbacks. One of them is that the Ember add-on is always tied to a specific version of the npm package and we're at the mercy of the maintainer if we want to upgrade that version. We have to wait for a new version to be released that upgrades the npm package version.

Since using npm packages directly "just works" with `ember-auto-import`, we'll install `tailwindcss` and then do some configuration.

```
$ npm install -D tailwindcss
```

One of the suggested ways to install Tailwind is as a PostCSS plugin, so we need to add PostCSS, too. This time, we'll use the Ember add-on as it does a lot of low-level plumbing that we don't want to replicate:

```
$ ember install ember-cli-postcss
```

We have to tell PostCSS to use Tailwind as a plugin. We do this in the `ember-cli-build.js` file where most build-related things live:

```
1 // ember-cli-build.js
2 'use strict';
3
4 const EmberApp = require('ember-cli/lib/broccoli/ember-app');
5
6 module.exports = function (defaults) {
7   let app = new EmberApp(defaults, {
8     // Add options here
+ 9     postcssOptions: {
+ 10       compile: {
+ 11         plugins: [{ module: require('tailwindcss') }],
+ 12       },
+ 13     },
14   });
15
16   // Use `app.import` to add additional libraries to the generated
17   // output files.
18   //
19   // If you need to use different assets in different
20   // environments, specify an object as the first parameter. That
21   // object's keys should be the environment name and the values
22   // should be the asset to use in that environment.
23   //
24   // If the library that you are including contains AMD or ES6
25   // modules that you would like to import into your application
26   // please specify an object with the list of modules as keys
27   // along with the exports of each module as its value.
28
29   return app.toTree();
30 }
```

The final step to enable Tailwind in our app is to use a handful of Tailwind directives in our main stylesheet:

```
1  /* app/styles/app.css */
+ 2 @tailwind base;
+ 3 @tailwind components;
+ 4 @tailwind utilities;
```

Because we've added new packages and modified a configuration file, we have to restart the Ember server:

We can now add some utility CSS classes to our list to make it look decent.

While we're at it, we'll also start building a simple layout (which we'll extend as we go). We'll add a header and a container element for the whole app.

```

1 {{!!-- app/templates/application.hbs --}}
2 {{pageTitle "Rock & Roll with Octane"}}
3
- 4 <ul>
- 5   {{#each
- 6     (array
- 7       (hash title="Black Dog" band="Led Zeppelin" rating=3)
- 8       (hash title="Yellow Ledbetter" band="Pearl Jam" rating=4)
- 9       (hash title="The Pretender" band="Foo Fighters" rating=2)
- 10    ) as |song|
- 11  }}
- 12  <li>
- 13    {{song.title}} by {{song.band}} ({{song.rating}})
- 14  </li>
- 15  {{/each}}
- 16 </ul>
+ 17 <div class="bg-blue-900 text-gray-100 p-8 h-screen">
+ 18   <div class="mx-auto mb-4">
+ 19     <div class="h-12 flex items-center mb-4 border-b-2">
+ 20       <a href="#">
+ 21         <h1 class="font-bold text-2xl">
+ 22           Rock & Roll <span class="font-normal">with Octane</span>
+ 23         </h1>
+ 24       </a>
+ 25     </div>
+ 26   <ul>
+ 27     {{#each
+ 28       (array
+ 29         (hash title="Black Dog" band="Led Zeppelin" rating=3)
+ 30         (hash title="Yellow Ledbetter" band="Pearl Jam" rating=4)
+ 31         (hash title="The Pretender" band="Foo Fighters" rating=2)
+ 32       ) as |song|
+ 33     }}
+ 34     <li class="leading-loose">
+ 35       {{song.title}} by {{song.band}} <span class="float-
+ 36         right">{{song.rating}}</span>

```

HBS

```
+ 37 {{ /each }}  
+ 38 </ul>  
+ 39 </div>  
+ 40 </div>
```

Tailwind is a utility-first CSS framework which means DOM elements can end up having lots of classes. If you don't feel like copying all of them, you're welcome to skip any that you don't feel are necessary (like "button polish" classes we'll use later).

HANDLEBARS' FAIL-SOFTNESS

BACKSTAGE

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:

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

HBS

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.

Now our app looks slightly more decent than the "black Times New Roman text on white background" style back-end developers tend to favor (no offense meant, I was a back-end developer for most of my professional life :P)

Rock & Roll with Octane

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

Next song

Notice we didn't write any JavaScript to have a working template. We defined the data the template works on in the template itself. In real life, however, this happens very rarely as data is loaded and possibly transformed before handing it over to the view layer for display. The most characteristic way in Ember to load data for templates is using routes which also has the very important feature of giving our app a structure.

So we'll look at how to do that in the next chapter, by building the loading and showing of bands and songs on different pages. Routing 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.

Related hits

- [TailwindCSS site](#)

CHAPTER 4

Routing

Tuning

As mentioned in the introduction chapter, URLs are crucial to Ember applications. They are the serialized form of the application's state, and deserialization (creating model objects from the URL) is handled by the routing mechanism. As I hope will be revealed in the next couple of chapters, Ember gives a lot of power to routes, and wielding that power is one of the most important things to learn to be an efficient Ember developer. 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 back-end (even if that back-end 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.

Routes set up data

The first step is to define the routes that 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:

```
1 // app/router.js
2 import EmberRouter from '@ember/routing/router';
3 import config from 'rarwe/config/environment';
4
5 export default class Router extends EmberRouter {
6   location = config.locationType;
7   rootURL = config.rootURL;
8 }
9
10 Router.map(function () {});
```

First, we import the configuration from `rarwe/config/environment` which is the module created from the `config/environment.js` module via a relative import.

If you peek inside the configuration you'll find that `locationType` is set to `auto` which is then assigned to the router's `location` property via the import. 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.

`hash` relies on the `hashchange` event in the browser and uses the `#` in the URL.

`none` does not affect the URL in any way. Though this may not seem to serve much purpose, it comes in 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.

One of the many things Ember CLI gives us to build Ember apps is blueprints. Blueprints provide an easy way to create –and modify– files that belong together. That again relieves the developer of having to manually create those files and fill them out with boilerplate code.

Blueprints can be run with the `generate` command, or `g`, for short. The `route` blueprint, for example, creates the route file, the related template and unit test and updates the route definitions.

So let's go ahead and run the blueprint for the two routes we were about to create before I embarked on that detour.

```
1 $ ember g route bands
2 installing route
3   create app/routes/bands.js
4   create app/templates/bands.hbs
5 updating router
6   add route bands
7 installing route-test
8   create tests/unit/routes/bands-test.js
9
10 $ ember g route songs
11 installing route
12   create app/routes/songs.js
13   create app/templates/songs.hbs
14 updating router
15   add route songs
16 installing route-test
17   create tests/unit/routes/songs-test.js
```

First, the routing configuration in the `router.js` file is amended with the new routes, adding a `this.route` line for each. Second, a corresponding, empty route module is created for each route.

The configuration in `router.js` only *specifies* the routes of the application. The behavior for these routes needs to be fleshed out in their corresponding route objects, each one an extension of Ember's `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`.

Let's start by removing the list of songs we have in the application template – we'll move that list to its dedicated page we've just created.

```

1  {{!!-- app/templates/application.hbs --}}
2  {{pageTitle "Rock & Roll with Octane"}}
3
4  <div class="bg-blue-900 text-gray-100 p-8 h-screen">
5    <div class="mx-auto mb-4">
6      <div class="h-12 flex items-center mb-4 border-b-2">
7        <a href="#">
8          <h1 class="font-bold text-2xl">
9            Rock & Roll <span class="font-normal">with Octane</span>
10         </h1>
11       </a>
12     </div>
13     <ul>
14       {{#each
15         (array
16           (hash title="Black Dog" band="Led Zeppelin" rating=3)
17           (hash title="Yellow Ledbetter" band="Pearl Jam" rating=4)
18           (hash title="The Pretender" band="Foo Fighters" rating=2)
19         ) as |song|
20       )}}
21       <li class="leading-loose">
22         {{song.title}} by {{song.band}} <span class="float-
23           right">{{song.rating}}</span>
24       </li>
25     {{/each}}
26   </ul>
27   {{outlet}}
28 </div>

```

HBS

All that's left in the application template is some markup for the application container, the site header, and the `{{outlet}}` (see more about outlets below). The application template will thus serve as a layout for the two other templates.

Let's start with our songs page. The `songs` template currently only holds an `{{outlet}}` so let's move the list of songs into it:

```

1  {{!!-- app/templates/songs.hbs --}}
2  {{pageTitle "Songs"}}
- 3 {{outlet}}
+ 4 
+ 5 <ul>
+ 6   {{#each
+ 7     (array
+ 8       (hash title="Black Dog" band="Led Zeppelin" rating=3)
+ 9       (hash title="Yellow Ledbetter" band="Pearl Jam" rating=4)
+ 10      (hash title="The Pretender" band="Foo Fighters" rating=2)
+ 11    ) as |song|
+ 12  }}
+ 13    <li class="leading-loose">
+ 14      {{song.title}} by {{song.band}} <span class="float-
right">{{song.rating}}</span>
+ 15    </li>
+ 16  {{/each}}
+ 17 </ul>

```

HBS

Our application recompiles successfully but we only see the header:

Rock & Roll with Octane

The router always enters the `application` route first (rendered at the root, `/`) 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 child 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 outlet defined by their parent template. 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. However, if you now manually add `/songs` to the URL, you'll see the list of songs we moved to `songs.hbs`:

Rock & Roll with Octane	
Black Dog by Led Zeppelin	3
Yellow Ledbetter by Pearl Jam	4
The Pretender by Foo Fighters	2

The application template provided the site header and specified an outlet. The child route, `songs`, rendered the list of songs (in `songs.hbs`) into this outlet.

Moving around with LinkTo

We could vastly improve user experience by providing links to the `bands` and `songs` pages instead of making users type into the URL bar.

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. Ember has a built-in way to go to routes of the application: the `LinkTo` component. Let's add a simple navigation header just below the site header:

```

1  {{!!-- app/templates/application.hbs --}}
2  {{pageTitle "Rock & Roll with Octane"}}
3
4  <div class="bg-blue-900 text-gray-100 p-8 h-screen">
5    <div class="mx-auto mb-4">
6      <div class="h-12 flex items-center mb-4 border-b-2">
7        <a href="#">
8          <h1 class="font-bold text-2xl">
9            Rock & Roll <span class="font-normal">with Octane</span>
10         </h1>
11       </a>
12     </div>
13     <nav class="h-8 flex items-center mb-8" role="navigation">
14       <LinkTo class="hover:text-gray-500" @route="bands">Bands</LinkTo>
15       <LinkTo class="hover:text-gray-500 ml-4" @route="songs">Songs</LinkTo>
16     </nav>
17     {{outlet}}
18   </div>
19 </div>

```

HBS

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.

The songs page should show the list of songs we'd added earlier. However, clicking on the Bands link takes us to an empty page. That's because we haven't put anything in the bands template (aka. `bands.hbs`). Let's do that now – in a novel way.

Using the model property

Our `songs` template has "inlined" data: the list of songs is defined in the template itself. We'll move one step closer to how things are in actual, real-world apps and define what's called the model of the template.

The underlying concept is that each template has a main set of data 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.

Here's how it works: you define a `model` method (commonly known as the model hook) on your route and Ember makes the value you return from this method available to the template as `@model`.

Let's see if it works for us. We start by defining the `model` hook in the `bands` route and return an array of bands:

```
1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3
4 export default class BandsRoute extends Route {
+ 5   model() {
+ 6     return [
+ 7       { name: 'Led Zeppelin' },
+ 8       { name: 'Pearl Jam' },
+ 9       { name: 'Foo Fighters' },
+10     ];
+11   }
12 }
```

In the template belonging to the route, `bands.hbs` we can access the array returned from the model hook as `@model`.

We can thus go through the bands and show the name for each:

```
1  {{!!-- app/templates/bands.hbs --}}
2  {{pageTitle "Bands"}}
3  {{outlet}}
4
5  <ul>
6    {{#each @model as |band|}}
7      <li>{{band.name}}</li>
8    {{/each}}
9  </ul>
```

HBS

Interacting with the app with the Ember Inspector

One of the great tools for inspecting (and debugging) our app is the Ember Inspector extension.

You can download it either [for Chrome](#), or [for Firefox](#).

Once you have installed the extension, go to the Bands page and open Developer Tools. You'll have an Ember tab right beside the default tabs. Click it and select Routes from the left panel. Next, click Route in the Objects column in the bands row and then the \$E next to the currentModel property in the right sidebar (you will see the \$E when you hover over currentModel).

(I recorded [a short screencast](#) about how to do that.)

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

Let's try to change "Pearl Jam" to "PJ":

```
$E[1].name = "PJ";
```

JS

The instruction succeeds but the displayed name doesn't change because Ember's rendering layer is not aware of the change and thus doesn't re-render the `{ {band.name} }` template snippet in the bands template.

There are two ways to fix this. The old way is to use a special setter that notifies the rendering layer of the change:

```
Ember.set($E[1], 'name', 'PJ')
```

js

The above works just fine but let's see the Octane way – using a tracked property.

Tracked properties

```
1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
+ 3 import { tracked } from '@glimmer/tracking';
+ 4
+ 5 class Band {
+ 6   @tracked name;
+ 7
+ 8   constructor(name) {
+ 9     this.name = name;
+10 }
+11 }
12
13 export default class BandsRoute extends Route {
+14   model() {
+15     return [
+16       new Band('Led Zeppelin'),
+17       new Band('Pearl Jam'),
+18       new Band('Foo Fighters'),
+19     ];
+20   }
21 }
```

If we now try to update the band name using simple JavaScript property setting (`$E[1].name = "PJ"`), it will work and we'll see the name update in the list of bands. The takeaway is that if a property is used in the template, it should be marked as tracked. Ember's tracking mechanism can then take care of keeping the property's value in sync with the one rendered in the template.

There is a simple rule for tracked properties: if you need to react to the change of the property (most often, by rendering the new value in a template), it should be marked as tracked.

Routing at the core of Ember

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. This 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.

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 their elegance is one of the things that sealed my fate with Ember for good. I hope to benignly infect you with my enthusiasm.

Related hits

- [How Autotracking Works by Chris Garrett](#)

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.*

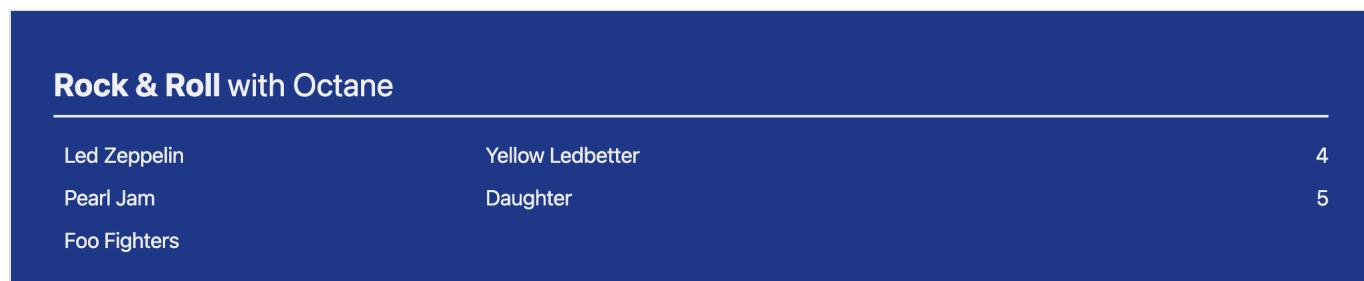
In API design, a has-many association (a band has many songs) is very often represented in the URL with the children in the association after the identified parent resource. In our case, that would be `/bands/:id/songs`.

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.

Defining the nested routes

Here is a mockup of what we want to build:



Our routes currently look like this:

```

1 // app/router.js
2 import EmberRouter from '@ember/routing/router';
3 import config from 'rarwe/config/environment';
4
5 export default class Router extends EmberRouter {
6   location = config.locationType;
7   rootURL = config.rootURL;
8 }
9
10 Router.map(function () {
11   this.route('bands');
12   this.route('songs');
13 });

```

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. Let's delete the `<nav>` tag and add some container markup so that the application template looks like this:

```

1  {{!!-- app/templates/application.hbs --}}
2  {{pageTitle "Rock & Roll with Octane"}}
3
4  <div class="bg-blue-900 text-gray-100 p-8 h-screen">
5    <div class="mx-auto mb-4">
6      <div class="h-12 flex items-center mb-4 border-b-2">
7        <a href="#">
8          <h1 class="font-bold text-2xl">
9            Rock & Roll <span class="font-normal">with Octane</span>
10         </h1>
11       </a>
12     </div>
13     <nav class="h-8 flex items-center mb-8" role="navigation">
14       <LinkTo class="hover:text-gray-500" @route="bands">Bands</LinkTo>
15       <LinkTo class="hover:text-gray-500 ml-4" @route="songs">Songs</LinkTo>
16     </nav>
17     {{outlet}}
18   <div class="flex flex-wrap -mx-2">
19     {{outlet}}
20   </div>
21 </div>
22 </div>

```

HBS

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:

```

1 // app/router.js
2 import EmberRouter from '@ember/routing/router';
3 import config from 'rarwe/config/environment';
4
5 export default class Router extends EmberRouter {
6   location = config.locationType;
7   rootURL = config.rootURL;
8 }
9
10 Router.map(function () {
- 11   this.route('bands');
- 12   this.route('songs');
+ 13   this.route('bands', function () {
+ 14     this.route('songs');
+ 15   });
16 });

```

(If, during building this chapter, you find yourself looking at perplexing console errors after making changes, it's possible the app still loads the /songs path which no longer has a matching route.)

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 is 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 this understanding, let's see if we can now create the types 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. The middle segment of the path, the id of the band, needs to be dynamic. In Ember, it has to be prefixed by a `:`.

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

```
1 $ ember g route bands/band --path=':id'  
2 installing route  
3   create app/routes/bands/band.js  
4   create app/templates/bands/band.hbs  
5 updating router  
6   add route bands/band  
7 installing route-test  
8   create tests/unit/routes/bands/band-test.js
```

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 the following:

```
1 // app/router.js
2 import EmberRouter from '@ember/routing/router';
3 import config from 'rarwe/config/environment';
4
5 export default class Router extends EmberRouter {
6   location = config.locationType;
7   rootURL = config.rootURL;
8 }
9
10 Router.map(function () {
11   this.route('bands', function () {
12     this.route('songs');
13
14     this.route('band', {
15       path: ':id'
16     });
17   });
18 }) ;
```

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

```

1 // app/router.js
2 import EmberRouter from '@ember/routing/router';
3 import config from 'rarwe/config/environment';
4
5 export default class Router extends EmberRouter {
6   location = config.locationType;
7   rootURL = config.rootURL;
8 }
9
10 Router.map(function () {
11   this.route('bands', function () {
- 12     this.route('songs');
- 13   }
- 14   this.route('band', {
- 15     path: ':id'
+ 16     this.route('band', { path: ':id' }, function () {
+ 17       this.route('songs');
18     });
19   });
20 });

```

By defining a dynamic segment (one that starts with a :) as `path`, 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 id that goes in the middle segment is also mapped, by virtue of the `band` segment being dynamic.

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.

And then our app will handle URLs for albums just like for songs:

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

You might wonder why we don't have a `path` option for the other routes, like `bands`. The `path` option tells Ember's router what URL path to generate for that route. By default, `path` is the route's name so `this.route('bands')` is the same as `this.route('bands', { path: 'bands' })`, only shorter.

Should you want a different, non-dynamic name, you can just pass it. I mean, pass it:

`this.route('bands', { path: 'les-bandes' })`. If you did this, the app would handle the following routes:

- `/`
- `/les-bandes`
- `/les-bandes/pearl-jam`
- `/les-bandes/led-zeppelin`
- `/les-bandes/pearl-jam/songs`
- `/les-bandes/led-zeppelin/songs`

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 its 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` calls) 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 is 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 relationship between bands and songs. We create a `Song` class and a few instances and assign each song to its appropriate band:

```

1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3 import { tracked } from '@glimmer/tracking';
4
5 class Band {
6   @tracked name;
7
8   constructor(name) {
9     constructor({ id, name, songs }) {
10       this.id = id;
11       this.name = name;
12       this.songs = songs;
13     }
14   }
15
16 class Song {
17   constructor({ title, rating, band }) {
18     this.title = title;
19     this.rating = rating ?? 0;
20     this.band = band;
21   }
22 }
23
24 export default class BandsRoute extends Route {
25   model() {
26     return [
27       new Band('Led Zeppelin'),
28       new Band('Pearl Jam'),
29       new Band('Foo Fighters'),
30     ];
31     let blackDog = new Song({
32       title: 'Black Dog',
33       band: 'Led Zeppelin',
34       rating: 3,
35     });
36
37     let yellowLedbetter = new Song({

```

```

+ 38     title: 'Yellow Ledbetter',
+ 39     band: 'Pearl Jam',
+ 40     rating: 4,
+ 41   });
+ 42
+ 43   let pretender = new Song({
+ 44     title: 'The Pretender',
+ 45     band: 'Foo Fighters',
+ 46     rating: 2,
+ 47   });
+ 48
+ 49   let daughter = new Song({
+ 50     title: 'Daughter',
+ 51     band: 'Pearl Jam',
+ 52     rating: 5,
+ 53   });
+ 54
+ 55   let ledZeppelin = new Band({
+ 56     id: 'led-zeppelin',
+ 57     name: 'Led Zeppelin',
+ 58     songs: [blackDog],
+ 59   });
+ 60
+ 61   let pearlJam = new Band({
+ 62     id: 'pearl-jam',
+ 63     name: 'Pearl Jam',
+ 64     songs: [yellowLedbetter, daughter],
+ 65   });
+ 66
+ 67   let fooFighters = new Band({
+ 68     id: 'foo-fighters',
+ 69     name: 'Foo Fighters',
+ 70     songs: [pretender],
+ 71   });
+ 72
+ 73   return [ledZeppelin, pearlJam, fooFighters];
74 }
75 }
```

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 returns all the bands.

In the next step, the `pearl-jam` URL segment is looked up. The dynamic `:id` path in the `band` route is found, and must be matched. Since `:id` 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 "semantic" id (for example, `pearl-jam`):

```
1 // app/routes/bands/band.js
2 import Route from '@ember/routing/route';
3
4 export default class BandsBandRoute extends Route {
+ 5   model(params) {
+ 6     let bands = this.modelFor('bands');
+ 7     return bands.find((band) => band.id === params.id);
+ 8   }
9 }
```

`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 id that was extracted from the URL.

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 add a new file and manually write the following into it:

```
1 // app/routes/bands/band/songs.js
2 import Route from '@ember/routing/route';
3
4 export default class BandsBandSongsRoute extends Route {
5   model() {
6     let band = this.modelFor('bands.band');
7     return band.songs;
8   }
9 }
```

We use `modelFor` again, this time to retrieve the band that was returned one level higher, and then return the list of songs belonging to that band.

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 following table shows the analysis of a route transition when booting up the app on `/bands/pearl-jam/songs`:

Route name	Matched URL segment	Matching route path	Params	Route hook
application	/	N/A	{}	application#model
bands	bands	bands	{}	bands#model
band	pearl-jam	:id	{ id: 'pearl-jam' }	band#model
songs	songs	songs	{}	songs#model

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 whether by clicking on a `LinkTo` or calling a transition method on the router service, the router will make the transition directly, not descending through the parent routes on its way.

This also means that 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 are the route definitions again:

```

1 // app/router.js
2 import EmberRouter from '@ember/routing/router';
3 import config from 'rarwe/config/environment';
4
5 export default class Router extends EmberRouter {
6   location = config.locationType;
7   rootURL = config.rootURL;
8 }
9
10 Router.map(function () {
11   this.route('bands', function () {
12     this.route('band', { path: ':id' }, function () {
13       this.route('songs');
14     });
15   });
16 });

```

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 templates for `application`, `bands`, `bands.band` and `bands.band.songs`.

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 screenshot 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 to create our band-songs master-detail view.

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 this change:

```

1  {{!!-- app/templates/bands.hbs --}}
2  {{pageTitle "Bands"}}
3
- 4 <ul>
- 5   {{#each @model as |band|}}
- 6     <li>{{band.name}}</li>
- 7   {{/each}}
- 8 </ul>
+ 9 <div class="w-1/3 px-2">
+ 10  <ul class="pl-2 pr-8">
+ 11    {{#each @model as |band|}}
+ 12      <li class="mb-2">
+ 13        <LinkTo @route="bands.band.songs" @model={{band.id}}>
+ 14          {{band.name}}
+ 15        </LinkTo>
+ 16      </li>
+ 17    {{/each}}
+ 18  </ul>
+ 19 </div>
+ 20 <div class="w-2/3 px-2">
+ 21  {{outlet}}
+ 22 </div>

```

HBS

Here, `LinkTo` also has a `@model` argument which will be used to construct the url segments. The `bands.band.songs` route has a dynamic segment, `:id`, so we are passing it in. If the route had more than one dynamic segment, we'd have to pass in an array, like this: `<LinkTo @route="very.very.deeply.nested.route" @model={{array value1 value2 ...}}>`.

The most important part is the `{{outlet}}` in the "main content" section. If we didn't have that, content rendered by child routes would be missing from the page (a somewhat common and quite perplexing error to track down).

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`:

```
1 $ ember g template bands/index  
2 installing template  
3     create app/templates/bands/index.hbs
```

...and then place the hint in it:

```
1 {{!-- app/templates/bands/index.hbs --}}  
+ 2 <p class="text-center">Select a band.</p>
```

HBS

The parent template `bands` creates a list of the bands in the left column and has an `{ { outlet } }` 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:

Rock & Roll with Octane

Led Zeppelin
Pearl Jam
Foo Fighters

Select a band.

We also want to show the songs of a band 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:

```
1  {{!!-- app/templates/bands/band.hbs --} }
2  {{pageTitle "Band" }}
3  {{outlet}}
```

HBS

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:

```
1  $ ember g template bands/band/songs
2  installing template
3  create app/templates/bands/band/songs.hbs
```

It should have the following content:

```
1  {{!!-- app/templates/bands/band/songs.hbs --} }
2  {{#if @model.length}}
3    <ul>
4      {{#each @model as |song|}}
5        <li class="mb-2">
6          {{song.title}}
7          <span class="float-right">{{song.rating}}</span>
8        </li>
9      {{/each}}
10    </ul>
11  {{else}}
12    <p class="text-center">
13      The band has no songs yet.
14    </p>
15  {{/if}}
```

HBS

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

Rock & Roll with Octane

Led Zeppelin	Yellow Ledbetter	4
Pearl Jam	Daughter	5
Foo Fighters		

Handlebars has a nifty extension of the `#each` helper. It has an `else` branch whose contents gets rendered if the collection passed to it falsey, or empty: `{{{#each ...}}}...{{{else}}}...{{/each}}}`. We didn't use it above because we needed different wrapping tags for the two cases but it's a nice tool to have in your toolbox.

Improving page titles

Whenever we generate a route, Ember inserts a `{{{page-title}}}` into the templates based on the name of the route. In some cases, this works out quite well, like in the case of the `bands` route we generated in the previous chapter. In other instances, though, we have to manually adjust those default titles.

If you go to any songs page, you'll find that the page title for them is `Band | Bands | Rock & Roll with Octane`. That's because nested routes also imply inheriting the page title of the parent route - and then prepending to it. Since we have `{{{page-title "Rock & Roll with Octane"}}}` in our application template and the application route is the parent of all routes, it will be the end of all page titles.

The `bands` route then adds its page title, `Bands` and the `bands.band` route adds `Band`, giving us the full page title of `Band | Bands | Rock & Roll with Octane` which just looks funny.

To improve our page titles, let's first remove the one in `bands.band`:

```
1  {{{!-- app/templates/bands/band.hbs --}}}  
- 2  {{{page-title "Band"}}}  
3  {{{outlet}}}
```

HBS

For our song pages, we'd like the page title to be Songs | Rock & Roll with Octane. If we simply define {{page-title "Songs"}} in the songs template, it's going to prepend it to Bands | Rock & Roll with Octane, becoming Songs | Bands | Rock & Roll with Octane. Instead, we should overwrite the previous title. Luckily, there's a replace option that does exactly that:

```
1  {{!!-- app/templates/bands/band/songs.hbs --}}
+ 2  {{page-title "Songs | Rock & Roll with Octane" replace=true}}
+ 3
4  {{#if @model.length}}
5    <ul>
6      {{#each @model as |song|}}
7        <li class="mb-2">
8          {{song.title}}
9          <span class="float-right">{{song.rating}}</span>
10       </li>
11     {{/each}}
12   </ul>
13 {{else}}
14   <p class="text-center">
15     The band has no songs yet.
16   </p>
17 {{/if}}
```

HBS

If we want a different separator than the default pipe, we can use the separator option. You can check out the [ember-page-title add-on](#) for more options.

We'll further improve our page titles in the [Controllers chapter](#).

{ {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 roadie 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?

The roadie is right. A simple way to delete all of these is by using the inverse of the generator which created them, the “destructor”. Instead of generating, we destroy:

```
1 $ ember destroy route songs
2   uninstalling route
3     remove app/templates/songs.hbs
4   updating router
5     remove route songs
6   uninstalling route-test
```

Next song

Taking our cue from the star rating widget in iTunes, we'll display –and update– song ratings with stars. To implement this feature, we'll get acquainted with the concept of components.

CHAPTER 6

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

Components come in many flavors. In some cases, they are designed to be reusable not just within the same application, but potentially between different projects, too. Other times, you create components for better developer ergonomics (more readable code, better encapsulation of state, etc.) and reusability is not a concern.

Components can only access arguments passed into them when invoked or properties they define. They know nothing about the context they are rendered in and can only communicate with the "outer world" by calling functions passed into to them – or communicating with singleton objects, like services. The resulting encapsulation fosters debuggability.

Component specification

Let's sketch out a spec for our star-rating component, keeping in mind that we want it to 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 (via the `maxRating` argument), and how many of these should be rendered as a full star (via the `rating` argument). Since clicking on one of these stars should trigger an action, that action must be customizable, too (via the `onUpdate` argument). With the specification nailed down, let's get to work.

Implementing the component

Defining the component's properties

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

```
1 $ ember g component star-rating --with-component-class
2 installing component
3   create app/components/star-rating.js
4   create app/components/star-rating.hbs
5 installing component-test
6   create tests/integration/components/star-rating-test.js
```

By default, the command only creates the component's template, an `.hbs` file so we also passed in the `--with-component-class` flag to have a `.js` file created. They are both added in the `app/components` folder of the app. Alternatively, you can run `ember g component-class star-rating` to create just the JavaScript file.

Components extend the `Component` class defined in the `@glimmer/component` module. In other words,

we created a Glimmer component which is a modern, light-weight alternative to Ember components (extending `@ember/component`) and is the default component type in Octane projects. In the class definition (the JavaScript file), the component should define its properties, methods, and actions.

Let's think a bit about what we need to display in the template. Taking in the `rating` and `maxRating` arguments, we'll need to show full stars up to `rating` and then display empty stars for the rest, `maxRating - rating` (save off-by-one errors :)). A handy way to do that is by defining a property (an ES6, to be precise):

```
1 // app/components/star-rating.js
2 import Component from '@glimmer/component';
3
4 export default class StarRatingComponent extends Component {
+ 5   get maxRating() {
+ 6     return this.args.maxRating ?? 5;
+ 7   }
+ 8
+ 9   get stars() {
+10     let stars = [];
+11     for (let i = 1; i <= this.maxRating; i++) {
+12       stars.push({
+13         rating: i,
+14         full: i <= this.args.rating,
+15       });
+16     }
+17     return stars;
+18   }
19 }
```

The arguments passed into a component are accessible on `this.args`. So `this.args.rating` refers to the value that was passed in as `rating`. The `maxRating` getter is a good example of providing a default value for the component: the passed in `maxRating` is used when provided, and we use a default value of 5 when it's not. Setting it up this way, we can use `this.maxRating` in the JavaScript file of the component, or the template.

In the `stars` getter, each array item (representing a star) has a `rating` (going from 1 to `maxRating`) and a

`full` property to decide whether the star should be drawn full or empty.

In Ember templates, results of function calls cannot be rendered (if you'd like to learn more about why that is so, read the "On the utility of explicit dependency definitions" section in the [Appendix](#)). That's why we needed to define a property, `stars`.

The `ember g component` command also created a component test file. Since we'll dedicate a separate chapter to testing, let's remove it for now:

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

Rendering the component

We now have the necessary supporting code for rendering our star-rating widget. The next step is to lay out in the template what should be rendered. Just to see that it works, let's output "ASCII stars" as a first step: "X"s for full stars, and "O"s for empty ones:

```
1  {{!-- app/components/star-rating.hbs --}}
- 2  {{yield}}
+ 3  {{#each this.stars as |star|}}
+ 4    {{if star.full "X" "O"}}
+ 5  {{/each}}
```

HBS

The above use of `if` is called "in-line if". It's the equivalent of the JavaScript ternary operator, `?.` It takes an expression as its first parameter: if it's true it returns its second parameter, if false, it returns the third one.

For our glorious component to be rendered on screen, we still have to invoke it. In the songs template, we switch the simple display of `song.rating` to calling the component with that same rating:

```

1  {{!!-- app/templates/bands/band/songs.hbs --}}
2  {{pageTitle "Songs | Rock & Roll with Octane" replace=true}}
3
4  {{#if @model.length}}
5    <ul>
6      {{#each @model as |song|}}
7        <li class="mb-2">
8          {{song.title}}
9          <span class="float-right">{{song.rating}}</span>
+ 10         <span class="float-right">
+ 11           <StarRating @rating={{song.rating}} />
+ 12         </span>
13       </li>
14     {{/each}}
15   </ul>
16 {{else}}
17   <p class="text-center">
18     The band has no songs yet.
19   </p>
20 {{/if}}

```

HBS

The way to call components in Ember is by typing the (camelized) name of the component in angle brackets. Component arguments need to be prefixed by `@` and html attributes (like `class`) need to be passed without a prefix (`class="mt-4"`). Component invocations need to start with a capital letter, to distinguish them from html tags.

Historically, there has been another component invocation method – using curly braces. It had been the only way to call components until Ember 3.4 and I bet you'll still encounter them in lots of Ember apps. I encourage you to learn more about it in the [Appendix](#).

The rough-cut version of the component is now rendered on screen:

Rock & Roll with Octane

Led Zeppelin

Yellow Ledbetter

XXXXO

Pearl Jam

Daughter

XXXXX

Foo Fighters

As aesthetics shouldn't be neglected even for a low-budget project like ours, let's improve displaying ratings and use actual stars instead of crosses and circles. An easy way to do that is by using Font Awesome fonts. Luckily there's an Ember-specific package we can use, so let's install it:

```
$ ember i @fortawesome/ember-fontawesome
```

That only gives us the plumbing, we also need to install the font sets we want to use:

```
$ npm install -D @fortawesome/free-regular-svg-icons @fortawesome/free-solid-svg-icons
```

We've added new packages so let's not forget to restart the server.

The `@fortawesome/ember-fontawesome` package provides us with an `FaIcon` component. It takes an `@icon` argument to know which icon to render and a `@prefix` argument for the icon style. We want the "star" icon, and the "fas" (solid) or "far" (regular) style, depending on whether the star should be full or empty:

```
1  {{!-- app/components/star-rating.hbs --}}
- 2  {{yield}}
+ 3  {{#each this.stars as |star|}}
+ 4    <FaIcon
+ 5      @icon="star"
+ 6      @prefix={{if star.full "fas" "far" }}
+ 7    />
+ 8  {{/each}}
```

HBS

With this change, song ratings are now displayed with stars:

Rock & Roll with Octane

Led Zeppelin

Yellow Ledbetter



Pearl Jam

Daughter



Foo Fighters

THE COMPONENT'S BLOCK FORM

BACKSTAGE

Components can be invoked in two forms: block and non-block form. We saw the `<StarRating>` component rendered in its non-block form:

```
<StarRating @rating={{song.rating}} />
```

HBS

The block form is also similar to HTML elements:

```
1 <StarRating @rating={{song.rating}}>
2   {{!-- Some content here --}}
3 </StarRating>
```

HBS

The block is the content between the opening and closing tag of the component name. It's not required in the block form but otherwise the component invocation looks plain funny. Plus, we could just use the non-block form of the component.

The component can return the passed block via the `{{yield}}` helper invoked from the component's template, optionally passing back block variables to the invoking context.

Theory is nice but since we have an example at hand, let me show you how the block form of the `star-rating` component could be implemented. You don't have to code along as I'm doing this as we don't need a block form of our component. However, knowing the block form is very useful as you'll encounter it all the time in projects.

Instead of rendering the stars in the component which assumes a few things about how it's used (that they use Font Awesome icons, for example) let's use the block form to allow the caller to customize what markup it renders. That means we should return (`yield` in Ember's component lingo) the `stars` property because that contains all the knowledge for customization:

```

+ 1 {{#if (has-block)}}
+ 2   {{yield this.stars}}
+ 3 {{else}}
+ 4   {{#each this.stars as |star|}}
+ 5     <FaIcon
+ 6       @icon="star"
+ 7       @prefix={{if star.full "fas" "far"}}
+ 8     />
+ 9   {{/each}}
+ 10 {{/if}}

```

HBS

`(has-block)` is a built-in template helper that is true if a block was passed to the component (i.e. if it's used in its block form).

The block form makes it possible to use the `star-rating` add-on with other icon libraries, like Glyphicon:

```

1 <StarRating @rating={{song.rating}} as |stars|>
2   {{#each stars as |star|}}
3     <span
4       class="glyphicon {{if star.isFull 'glyphicon-star'
'glyphicon-star-empty'}}">
5     </span>
6   {{/each}}
7 </StarRating>

```

HBS

In our app, the non-block form suffices, but most of the time (and almost certainly if you author an add-on) you'll need a block form, too.

Speaking to the outside world

We now have a component that shows a song's rating via stars. 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'll need to handle the click and update the song's rating to the star's value that was clicked. We don't need to update the stars' appearance as the `stars` getter in the component takes care of that. Because we use `this.args.rating` inside the function, any time `this.args.rating` changes, the getter will recompute and the new `stars` will be rendered in the template. In Ember, getters are auto-tracked so we don't need to mark it as such.

The simplest way to set a value in the template is to use the [ember-set-helper add-on](#), so we'll need to install it (and then restart the Ember server process):

```
$ ember install ember-set-helper
```

The API of our `StarRating` component should be extended by an argument: a function that gets called when a star is clicked. Let's name it `onUpdate` and use the `set` helper in defining the function:

```

1  {{!!-- app/templates/bands/band/songs.hbs --}}
2  {{pageTitle "Songs | Rock & Roll with Octane" replace=true}}
3
4  {{#if @model.length}}
5    <ul>
6      {{#each @model as |song|}}
7        <li class="mb-2">
8          {{song.title}}
9          <span class="float-right">
- 10         <StarRating @rating={{song.rating}} />
+ 11         <StarRating
+ 12           @rating={{song.rating}}
+ 13           @onUpdate={{set song "rating"}}
+ 14         />
15       </span>
16     </li>
17   {{/each}}
18 </ul>
19 {{else}}
20   <p class="text-center">
21     The band has no songs yet.
22   </p>
23 {{/if}}

```

HBS

The first argument to `set` is the object we want to update, the second one is the name of the property to update. If we don't provide the third argument, the property will be set to the value the function is called with. So `{{set song "rating"}}` creates a setter function that sets `song.rating` to whatever value the function is called with.

We now have to call the `@onUpdate` function from the component whenever one of the stars is clicked.

```

1  {{!-- app/components/star-rating.hbs --}}
- 2  {{yield}}
+ 3  {{#each this.stars as |star|}}
+ 4    <button type="button" {{on "click" (fn @onUpdate star.rating)}}>
+ 5      <FaIcon
+ 6        @icon="star"
+ 7        @prefix={{if star.full "fas" "far"}}
+ 8      />
+ 9    </button>
+ 10   {{/each}}

```

HBS

There are lots of novelties here, so let's see them one by one.

`on` adds an event listener to the element it is invoked on. It takes the event type as its first argument and the handler function as the second one.

Next, the `fn` helper can take an arbitrary number of arguments at call time, which will be prepended to the list of arguments the resulting function is called with (if you're familiar with functional programming terms, that's called partial application). The event handler set up by `on` will then be called with these arguments first, and the DOM event as the last parameter.

If this was JavaScript, `fn` could be implemented as below:

```

1  function fn(originalFn, ...prependedArgs) {
2    return function(...args) {
3      return originalFn(...prependedArgs, ...args);
4    }
5  }

```

We know from before that `@onUpdate` is a setter for `song.rating` which means that clicking the button will call the setter with `song.rating`. Putting these together, the song's rating will be set to the rating value (1-5) the user clicked on.

As the song's rating is updated, the change of the `@rating` argument 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.

BACKSTAGE

DATA DOWN ACTIONS UP (DDAU)

In the recently finished `StarRating` component, we passed `@rating` into the component by calling `<StarRating @rating={{song.rating}} ... />` instead of mutating it in place. We also passed in an `@onUpdate` function that is called by the component and used to update the song's rating *in the caller's context*. So we're passing data (in this case, the `@rating` argument) down, while using a handler function (in Ember terms, an action) to communicate upwards.

That uni-directional flow has been dubbed "Data Down Actions Up" (or DDAU, for short) and is one of the best practices that becomes second nature to Ember developers after a while. It establishes a data flow that is easy to debug (at least definitely easier than if we had two-way data bindings and we changed passed-in values willy-nilly) and gives flexibility to the component that leverages it.

Component invocations as function calls

In many ways, invoking components in Ember.js is just like calling functions (in, say, JavaScript). The result is a chunk of HTML that is inserted into the template where the invocation was made from. The passed in key-value assignments are like named arguments with which a function is called.

There are some differences, like the binding that is established between a passed in value and the property in the component, but you can bridge that gap by imagining that the function is called each time any of its arguments changes.

I've found that this analogy helps comprehension and dispels most of the magic with components.

Expressions in templates

At this point, we have several slightly differing dynamic expressions in our templates, so let's stop for a moment to gain some clarity on them.

Let's first take a look at the `songs.hbs` template, and go from top to bottom:

```
1 {{!!-- app/templates/bands/band/songs.hbs --} }
2 {{pageTitle "Songs | Rock & Roll with Octane" replace=true}}
3
4 {{#if @model.length}}
5   <ul>
6     {{#each @model as |song|}}
7       <li class="mb-2">
8         {{song.title}}
9         <span class="float-right">
10          <StarRating
11            @rating={{song.rating}}
12            @onUpdate={{set song "rating"}}
13          />
14        </span>
15      </li>
16    {{/each}}
17  </ul>
18 {{else}}
19   <p class="text-center">
20     The band has no songs yet.
21   </p>
22 {{/if}}
```

HBS

`#if` and `#each` are basic Ember helpers. We'll see helpers in more detail (and add one of our own making) in the [Helpers chapter](#).

`@model` denotes an argument that was passed into this context and is immutable (this is also known as a *named argument*). In this case, it's the `model` returned from the corresponding route and the "passing in"

happens invisibly, behind the scenes.

The `#each` helper provides a so-called *block variable* between pipes: `| song |`. The variable only lives within the block – so it's similar to `let` declarations in JavaScript.

`<StarRating>` is a **component invocation**. Component arguments need to be prefixed with `@` upon calling: `@rating` and `@onUpdate` are the arguments we're passing in.

The `set` in `{ {set song "rating" } }` is again a helper, implemented by the `ember-set-helper` add-on.

Since a few concepts don't appear in this file, let's now open the component's template, `star-rating.hbs`:

```
1 {{!-- app/components/star-rating.hbs --}}
2 {{#each this.stars as |star|}}
3   <button type="button" {{on "click" (fn @onUpdate star.rating)}}>
4     <FaIcon
5       @icon="star"
6       @prefix={{if star.full "fas" "far"}}
7     />
8   </button>
9 {{/each}}
```

HBS

`this.stars` is a *property* defined on the context of the template. Because that's a component template, the context (the `this`) is the backing class of the component.

`on` is a *modifier*. You don't need to worry about them too much for now. The interesting thing about them is they need to be added in "element space": within a tag, not between them.

Finally, `fn` and `if` are again helpers, just like `set` in the previous template.

Clarity in templates

It's also worth mentioning that Ember allowed developers to be a lot sloppier in earlier versions of Ember. Coming across `{ {stars} }` in a template could have meant a property, a component argument, a block

variable or even a helper call with no arguments! Needless to say, this meant more cognitive load and switching between files for developers.

On the road to Octane, this has seen an enormous improvement to the point where one can tell right away where a certain expression comes from at a glance.

Expression	What is it?	Definition found in
<code>{{stars}}</code>	block variable	the nearest block
<code>{{this.stars}}</code>	property	the template's context
<code>{{@stars}}</code>	argument	the call site

So developers need to be more precise and use the right prefix, and we actually have a case at hand we need to fix.

WHAT ABOUT MUT?

You might have heard about the `mut` helper, built into the framework, which provides a handy way to provide a setter function. Its purpose is very similar to the one of `ember-set-helper` so it's legit to ask why we add an extra dependency. `mut` is indeed handy but it has a few drawbacks which are very well described in pzuraq's article, [On `{mut}` and 2-Way-Binding](#).

I stopped using `mut` because its developer ergonomics are not ideal and because it serves both as a setter and a getter which can be quite confusing in certain scenarios. That said, it really is a handy shorthand but now that we have `ember-set-helper`, I prefer to use `set`.

BACKSTAGE

Next song

At this point, the user can freely navigate between bands and see and update song ratings but there is no way to create new bands or songs. We'll fix that in the next chapter through the cunning use of controllers.

Related hits

- ember-set-helper
- On {{mut}} and 2-Way-Binding

CHAPTER 7

Controllers

Tuning

At this point, we can navigate through the app and see a list of bands and their songs, but we cannot add new ones, so our next task will be defining actions that implement these tasks.

Creating songs

We'll put a call-to-action button below the list of songs which prompts the user to create a new song. If clicked, we should show an input field for the song title and an "Add" button to save the song. Let's start by adding a nice, purple button:

```

1  {{!!-- app/templates/bands/band/songs.hbs --}}
2  {{pageTitle "Songs | Rock & Roll with Octane" replace=true}}
3
4  {{#if @model.length}}
5    <ul>
6      {{#each @model as |song|}}
7        <li class="mb-2">
8          {{song.title}}
9          <span class="float-right">
10         <StarRating
11             @rating={{song.rating}}
12             @onUpdate={{set song "rating"}}
13             />
14         </span>
15       </li>
16     {{/each}}
17   </ul>
18 {{else}}
19   <p class="text-center">
20     The band has no songs yet.
21   </p>
22 {{/if}}
+ 23 <div class="flex justify-center mt-2">
+ 24   <button
+ 25     type="button"
+ 26     class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-lg
+           hover:text-white hover:bg-purple-500 focus:outline-none"
+ 27   >
+ 28   Add song
+ 29   </button>
+ 30 </div>

```

HBS

That's indeed a nice button but it's not useful unless it triggers an action, so let's add one by using the `on` modifier as we did for the `star-rating` component. When clicked, we want the button to be replaced by a text field for the new song title and a pair of buttons to save the song or cancel the process.

Let's call the variable that determines whether to show the "Add song" button or the text field and buttons

`showAddSong`. We make the following modifications in the template:

```

1  {{!!-- app/templates/bands/band/songs.hbs --}}
2  {{pageTitle "Songs | Rock & Roll with Octane" replace=true}}
3
4  {{#if @model.length}}
5    <ul>
6      {{#each @model as |song|}}
7        <li class="mb-2">
8          {{song.title}}
9          <span class="float-right">
10         <StarRating
11             @rating={{song.rating}}
12             @onUpdate={{set song "rating"}}
13             />
14         </span>
15       </li>
16     {{/each}}
17   </ul>
18 {{else}}
19   <p class="text-center">
20     The band has no songs yet.
21   </p>
22 {{/if}}
+ 23 {{#if this.showAddSong}}
+ 24   <div class="flex justify-center mt-2">
+ 25     <button
+ 26       type="button"
+ 27       class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-
lg hover:text-white hover:bg-purple-500 focus:outline-none"
+ 28       {{on "click" (set this "showAddSong" false)}}
+ 29     >
+ 30     Add song
+ 31   </button>
+ 32 </div>
+ 33 {{else}}
+ 34   <div class="mt-6 flex">
+ 35     <input
+ 36       type="text"

```

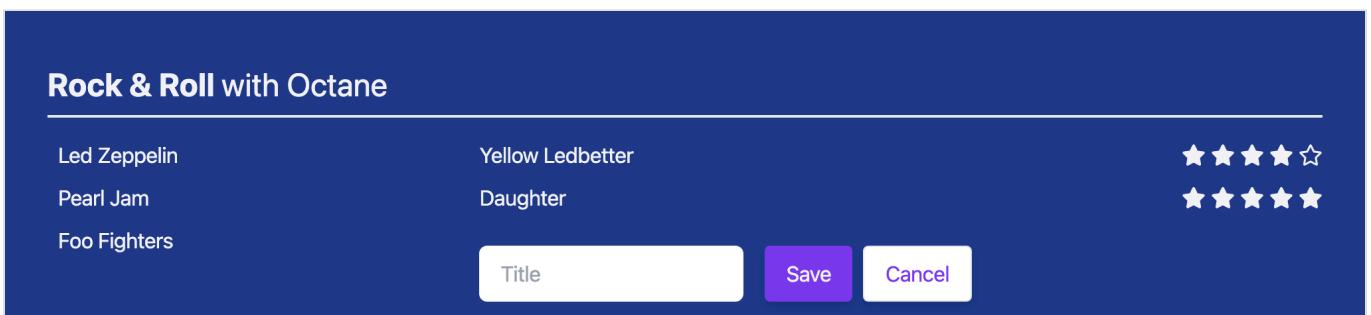
```

+ 37   class="text-gray-800 bg-white rounded-md py-2 px-4"
+ 38   placeholder="Title"
+ 39   value={{this.title}}
+ 40   />
+ 41   <button
+ 42     type="button"
+ 43     class="ml-4 px-4 py-2 rounded bg-purple-600 shadow-md
      hover:shadow-lg hover:text-white"
+ 44   >
+ 45     Save
+ 46   </button>
+ 47   <button
+ 48     type="button"
+ 49     class="ml-2 px-4 p-2 rounded bg-white border border-bg-
      purple-600 shadow-md text-purple-600 hover:shadow-lg"
+ 50   >
+ 51     Cancel
+ 52   </button>
+ 53 </div>
+ 54 {{/if}}

```

A simple action is defined by the `set` helper: when the button is clicked, `this.showAddSong` is flipped to false which de-renders the Add song button and shows the text field with a Save and Cancel button beside it.

However, when the page is rendered initially, we see the text field and buttons instead of the Add song button:



The reason is that `this.showAddSong` does not have an initial value (in other words, it's `undefined`), so

the `else` branch is rendered. We should set an initial value of `true` but the question is: where does the `showAddSong` property live? In other words, what is the context (the `this`) of the `songs` template?

As you might have guessed from the chapter title, the answer is: it lives on the controller. Just as the context of a `component` template is the backing JavaScript class (the `component.js` file), the context of a `route-driven` template is the corresponding controller.

WHY NOT A COMPONENT?

BACKSTAGE

You could argue that route-driven templates are templates, too. Is there really a need for a separate class, controllers, to back route-driven templates? They could just be components and we could thus create a component that matches the name of the template (in our case, `bands/band/songs`), just as we do in the case of "real" components.

Part of the reason is that –believe it or not– controllers pre-date components so they'd need to be deprecated and then removed from the framework.

One of the earliest RFCs aimed to introduce the concept of "[routable components](#)". However, this turned out to be quite tricky and the RFC was eventually closed (almost 3 years later). There is another, open RFC to provide an alternative, "[An alternative to Controllers](#)" that you can track.

I expect controllers to be around for some time still and they can be quite useful, so it's good to know about how they work and when you should reach for them. They are also different from components in a few ways so their replacement is not trivial. The best way to think about them is that they are top-level components for a specific route – with a few peculiarities we'll learn about in this chapter.

We established we need a `songs` controller to be able to define a default value for `showAddSong`, so let's create one:

```
1 $ ember g controller bands/band/songs
2 installing controller
3   create app/controllers/bands/band/songs.js
4 installing controller-test
5   create tests/unit/controllers/bands/band/songs-test.js
```

The name and location of controllers follow the same pattern we saw in the [Nesting routes chapter](#). The controller for the `bands.band.songs` route has the same name and its file is placed in `app/controllers/bands/band/songs.js`.

We can now define the default value of `showAddSong` as `true`:

```
1 // app/controllers/bands/band/songs.js
2 import Controller from '@ember/controller';
+ 3 import { tracked } from '@glimmer/tracking';
4
5 export default class BandsBandSongsController extends Controller {
+ 6   @tracked showAddSong = true;
7 }
```

As the property is used in the template, we defined it as `tracked`. That small change is enough for the "Add song" button to be shown by default.

We can now make saving a song with the given title work. In order to do that, we'll need to add interactivity to the buttons and the input.

```

1  {{!-- app/templates/bands/band/songs.hbs --}}
2  {{pageTitle "Songs | Rock & Roll with Octane" replace=true}}
3
4  {{#if @model.length}}
5    <ul>
6      {{#each @model as |song|}}
7        <li class="mb-2">
8          {{song.title}}
9          <span class="float-right">
10            <StarRating
11              @rating={{song.rating}}
12              @onUpdate={{set song "rating"}}
13            />
14          </span>
15        </li>
16      {{/each}}
17    </ul>
18  {{else}}
19    <p class="text-center">
20      The band has no songs yet.
21    </p>
22  {{/if}}
23  {{#if this.showAddSong}}
24    <div class="flex justify-center mt-2">
25      <button
26        type="button"
27        class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-lg
28        hover:text-white hover:bg-purple-500 focus:outline-none"
29        {{on "click" (set this "showAddSong" false)}}
30      >
31        Add song
32      </button>
33    </div>
34  {{else}}
35    <div class="mt-6 flex">
36      <input

```

HBS

```

37         class="text-gray-800 bg-white rounded-md py-2 px-4"
38         placeholder="Title"
39         value={{this.title}}
+ 40         {{on "input" this.updateTitle}}
41     />
42     <button
43         type="button"
44         class="ml-4 px-4 py-2 rounded bg-purple-600 shadow-md
        hover:shadow-lg hover:text-white"
+ 45         {{on "click" this.saveSong}}
46     >
47     Save
48     </button>
49     <button
50         type="button"
51         class="ml-2 px-4 p-2 rounded bg-white border border-bg-
        purple-600 shadow-md text-purple-600 hover:shadow-lg"
+ 52         {{on "click" this.cancel}}
53     >
54     Cancel
55     </button>
56   </div>
57 {{/if}}

```

In the Components chapter we got away with defining the event handler for DOM events inline, by using the `set` helper. Here, however, we'll need to define the actions ourselves.

Currently, since we haven't written those actions, we get the following error:

Assertion failed: You must pass a function as the second argument to the `on` modifier.

The error is due to the fact that `this.updateTitle`, `this.saveSong` and `this.cancel` are all undefined so we now have to define them on the template's context. Remember what that context is? Yep, the controller.

Actions

Let's open the controller file and define these new event handlers:

```
1 // app/controllers/bands/band/songs.js
2 import Controller from '@ember/controller';
3 import { tracked } from '@glimmer/tracking';
+ 4 import { action } from '@ember/object';
+ 5 import { Song } from 'rarwe/routes/bands';
6
7 export default class BandsBandSongsController extends Controller {
8   @tracked showAddSong = true;
+ 9   @tracked title = '';
+ 10
+ 11   @action
+ 12   updateTitle(event) {
+ 13     this.title = event.target.value;
+ 14   }
+ 15
+ 16   @action
+ 17   saveSong() {
+ 18     let song = new Song({ title: this.title, band: this.band });
+ 19     this.band.songs = [...this.band.songs, song];
+ 20     this.title = '';
+ 21     this.showAddSong = true;
+ 22   }
+ 23
+ 24   @action
+ 25   cancel() {
+ 26     this.title = '';
+ 27     this.showAddSong = true;
+ 28   }
29 }
```

It's the first time we come across the `action` decorator. All it does is that it correctly binds the context of the

template to the backing class – it makes sure that when you use `this.saveSong` in the template, `this` will refer to the controller instance. This makes it also useful (and widely used) for components where you need to do the same thing. You only need to use `@action` if the function you attach it to gets called from the template.

The above code assumes that 1) we can import the `Song` class from the route it's defined in, and 2) that `this.band` is set to the band we're creating the song for. Neither of these holds, so let's fix them.

Exporting the `Song` class is as easy as playing an E minor chord:

```

1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3 import { tracked } from '@glimmer/tracking';
4
5 class Band {
6   @tracked name;
7
8   constructor({ id, name, songs }) {
9     this.id = id;
10    this.name = name;
11    this.songs = songs;
12  }
13}
14
- 15 class Song {
+ 16 export class Song {
17   constructor({ title, rating, band }) {
18     this.title = title;
19     this.rating = rating ?? 0;
20     this.band = band;
21   }
22 }
23
24 export default class BandsRoute extends Route {
25   model() {
26     let blackDog = new Song({
27       title: 'Black Dog',
28       band: 'Led Zeppelin',
29       rating: 3,
30     });
31
32     let yellowLedbetter = new Song({
33       title: 'Yellow Ledbetter',
34       band: 'Pearl Jam',
35       rating: 4,
36     });
37

```

```

38     let pretender = new Song({
39         title: 'The Pretender',
40         band: 'Foo Fighters',
41         rating: 2,
42     });
43
44     let daughter = new Song({
45         title: 'Daughter',
46         band: 'Pearl Jam',
47         rating: 5,
48     });
49
50     let ledZeppelin = new Band({
51         id: 'led-zeppelin',
52         name: 'Led Zeppelin',
53         songs: [blackDog],
54     });
55
56     let pearlJam = new Band({
57         id: 'pearl-jam',
58         name: 'Pearl Jam',
59         songs: [yellowLedbetter, daughter],
60     });
61
62     let fooFighters = new Band({
63         id: 'foo-fighters',
64         name: 'Foo Fighters',
65         songs: [pretender],
66     });
67
68     return [ledZeppelin, pearlJam, fooFighters];
69 }
70 }
```

Setting the `band` property on the controller needs new knowledge, though. Ember's routing layer provides a hook that comes handy when you want to set properties on the controller: `setupController`. As we can access the `band` in the route (actually, we already do), we can leverage the `setupController` method to set it:

```

1 // app/routes/bands/band/songs.js
2 import Route from '@ember/routing/route';
3
4 export default class BandsBandSongsRoute extends Route {
5   model() {
6     let band = this.modelFor('bands.band');
7     return band.songs;
8   }
+ 9
+ 10   setupController(controller) {
+ 11     super.setupController(...arguments);
+ 12     controller.set('band', this.modelFor('bands.band'));
+ 13   }
14 }
```

The method gets passed the controller itself as the first parameter and the resolved model as the second one (which we don't need here). We shouldn't forget to call `super` as it's the default implementation that takes care of setting the `model` property on the controller.

The new song is now created and added to the related band's `songs` array correctly and yet we don't see the new song appear in the list. How come?

Rendering the list of songs reactively

In the `songs` template (`songs.hbs`) we render each song in `@model`. However, `@model` is set only once, when we enter the corresponding route. When we add the new song to the band, `band.songs` changes but `{{#each @model as |song|}}` is not re-run as `@model` itself didn't change. One way to fix this would be to reset `@model` to the new value but as the `@` prefix indicates, it's an immutable reference from what was returned from the `model` hook of the route.

A cleaner solution is to iterate over an expression that does update whenever we change the songs of the band. Since we know we also need to get hold of the band we create the songs for, it's easiest to change the route so that it returns the band object and loop through `@model.songs` in the template:

```
1 // app/routes/bands/band/songs.js
2 import Route from '@ember/routing/route';
3
4 export default class BandsBandSongsRoute extends Route {
5   model() {
6     let band = this.modelFor('bands.band');
7     return band.songs;
8   }
9
10  setupController(controller) {
11    super.setupController(...arguments);
12    controller.set('band', this.modelFor('bands.band'));
13  }
14}
```

Yes, you see that right, we removed everything – the default implementation of the model hook is to return the model of the parent route. Since we need the band as the model, that suits us perfectly. Now, to adjust the controller and template:

```

1 // app/controllers/bands/band/songs.js
2 import Controller from '@ember/controller';
3 import { tracked } from '@glimmer/tracking';
4 import { action } from '@ember/object';
5 import { Song } from 'rarwe/routes/bands';
6
7 export default class BandsBandSongsController extends Controller {
8   @tracked showAddSong = true;
9   @tracked title = '';
10
11   @action
12   updateTitle(event) {
13     this.title = event.target.value;
14   }
15
16   @action
17   saveSong() {
- 18     let song = new Song({ title: this.title, band: this.band });
- 19     this.band.songs = [...this.band.songs, song];
+ 20     let song = new Song({ title: this.title, band: this.model });
+ 21     this.model.songs = [...this.model.songs, song];
18     this.title = '';
19     this.showAddSong = true;
20   }
21
22   @action
23   cancel() {
24     this.title = '';
25     this.showAddSong = true;
26   }
27 }
```

```

1  {{!!-- app/templates/bands/band/songs.hbs --}}
2  {{pageTitle "Songs | Rock & Roll with Octane" replace=true}}
3
- 4 {{#if @model.length}}
+ 5 {{#if @model.songs.length}}
6    <ul>
- 7      {{#each @model as |song|}}
+ 8      {{#each @model.songs as |song|}}
9        <li class="mb-2">
10          {{song.title}}
11          <span class="float-right">
12            <StarRating
13              @rating={{song.rating}}
14              @onUpdate={{set song "rating"}}
15            />
16          </span>
17        </li>
18      {{/each}}
19    </ul>
20  {{else}}
21    <p class="text-center">
22      The band has no songs yet.
23    </p>
24  {{/if}}
25  {{#if this.showAddSong}}
26    <div class="flex justify-center mt-2">
27      <button
28        type="button"
29        class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-lg
hover:text-white hover:bg-purple-500 focus:outline-none"
30        {{on "click" (set this "showAddSong" false)}}
31      >
32        Add song
33      </button>
34    </div>
35  {{else}}
36    <div class="mt-6 flex">

```

```

37   <input
38     type="text"
39     class="text-gray-800 bg-white rounded-md py-2 px-4"
40     placeholder="Title"
41     value={{this.title}}
42     {{on "input" this.updateTitle}}
43   />
44   <button
45     type="button"
46     class="ml-4 px-4 py-2 rounded bg-purple-600 shadow-md
47       hover:shadow-lg hover:text-white"
48     {{on "click" this.saveSong}}
49   >
50     Save
51   </button>
52   <button
53     type="button"
54     class="ml-2 px-4 p-2 rounded bg-white border border-bg-
55       purple-600 shadow-md text-purple-600 hover:shadow-lg"
56     {{on "click" this.cancel}}
57   >
58     Cancel
59   </button>
59 {{/if}}

```

When we now create a new song, it doesn't appear in the list. The song is added to `this.model.songs` where the model is a Band object. However, the `songs` property is not tracked so Ember's templating engine doesn't re-render the `@model.songs` chunk in the template.

We can fix this by marking the `songs` property as tracked:

```
1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3 import { tracked } from '@glimmer/tracking';
4
5 class Band {
6   @tracked name;
+ 7   @tracked songs;
8
9   constructor({ id, name, songs }) {
10     this.id = id;
11     this.name = name;
12     this.songs = songs;
13   }
14 }
15
16 export class Song {
17   constructor({ title, rating, band }) {
18     this.title = title;
19     this.rating = rating ?? 0;
20     this.band = band;
21   }
22 }
23
24 export default class BandsRoute extends Route {
25   model() {
26     let blackDog = new Song({
27       title: 'Black Dog',
28       band: 'Led Zeppelin',
29       rating: 3,
30     });
31
32     let yellowLedbetter = new Song({
33       title: 'Yellow Ledbetter',
34       band: 'Pearl Jam',
35       rating: 4,
36     });
37   }
}
```

```

38     let pretender = new Song({
39         title: 'The Pretender',
40         band: 'Foo Fighters',
41         rating: 2,
42     });
43
44     let daughter = new Song({
45         title: 'Daughter',
46         band: 'Pearl Jam',
47         rating: 5,
48     });
49
50     let ledZeppelin = new Band({
51         id: 'led-zeppelin',
52         name: 'Led Zeppelin',
53         songs: [blackDog],
54     });
55
56     let pearlJam = new Band({
57         id: 'pearl-jam',
58         name: 'Pearl Jam',
59         songs: [yellowLedbetter, daughter],
60     });
61
62     let fooFighters = new Band({
63         id: 'foo-fighters',
64         name: 'Foo Fighters',
65         songs: [pretender],
66     });
67
68     return [ledZeppelin, pearlJam, fooFighters];
69 }
70 }
```

We can now create songs that appear in the list instantly.

This is a good time to improve the page title of our songs pages. Currently it's always just "Songs" and we can now include the name of the band so that the user knows by glancing at the browser tab which band the

page is for:

```

1  {{!!-- app/templates/bands/band/songs.hbs --}}
- 2  {{pageTitle "Songs | Rock & Roll with Octane" replace=true}}
+ 3  {{pageTitle @model.name " songs | Rock & Roll with Octane"
   replace=true}}
4
5  {{#if @model.songs.length}}
6    <ul>
7      {{#each @model.songs as |song|}}
8        <li class="mb-2">
9          {{song.title}}
10         <span class="float-right">
11           <StarRating
12             @rating={{song.rating}}
13             @onUpdate={{set song "rating"}}
14           />
15         </span>
16       </li>
17     {{/each}}
18   </ul>
19 {{else}}
20   <p class="text-center">
21     The band has no songs yet.
22   </p>
23 {{/if}}
24 {{#if this.showAddSong}}
25   <div class="flex justify-center mt-2">
26     <button
27       type="button"
28       class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-lg
29 hover:text-white hover:bg-purple-500 focus:outline-none"
30       {{on "click" (set this "showAddSong" false)}}
31     >
32       Add song
33     </button>
34   </div>
35 {{else}}
36   <div class="mt-6 flex">

```

```

36   <input
37     type="text"
38     class="text-gray-800 bg-white rounded-md py-2 px-4"
39     placeholder="Title"
40     value={{this.title}}
41     {{on "input" this.updateTitle}}
42   />
43   <button
44     type="button"
45     class="ml-4 px-4 py-2 rounded bg-purple-600 shadow-md
46       hover:shadow-lg hover:text-white"
47     {{on "click" this.saveSong}}
48   >
49     Save
50   </button>
51   <button
52     type="button"
53     class="ml-2 px-4 p-2 rounded bg-white border border-bg-
54       purple-600 shadow-md text-purple-600 hover:shadow-lg"
55     {{on "click" this.cancel}}
56   >
57     Cancel
58   </button>
59 </div>
60 {{/if}}

```

Beware of singletons

If we play around with the app, we notice something strange. If we start typing a new song title, but then decide to switch to a new band, the text field stays on the screen, with the unfinished song title as its value. That seems weird and is caused by the fact that controllers are singletons in Ember.

What does that mean? It means that each controller class only has one single instance in the application. That instance is created when we enter the corresponding route and is not destroyed when we move away from it.

That explains the above phenomenon: `this.showAddSong` and `this.title` are properties on the controller so they retain their value even when moving to a new page. If that's not how we want it to be, Ember has us covered: it provides a specific hook to update controller properties when moving between routes. It's called `resetController` and it's called on the route every time a route transition is initiated.

We can use this method to reset the properties:

```
1 // app/routes/bands/band/songs.js
2 import Route from '@ember/routing/route';
3
4 export default class BandsBandSongsRoute extends Route {
+ 5   resetController(controller) {
+ 6     controller.title = '';
+ 7     controller.showAddSong = true;
+ 8   }
9 }
```

Creating bands

Let's take a slightly different approach to create bands and have bands be created on a separate page, with a very simple form. That means we'll need to create a new route to show that form on, and a button to take the user there.

```
1 $ ember g route bands/new
2 installing route
3   create app/routes/bands/new.js
4   create app/templates/bands/new.hbs
5 updating router
6   add route bands/new
7 installing route-test
8   create tests/unit/routes/bands/new-test.js
```

```
1  {{!!-- app/templates/bands.hbs --}}
2  {{pageTitle "Bands"}}
3
4  <div class="w-1/3 ppx-2">
5    <ul class="pl-2 pr-8">
6      {{#each @model as |band|}}
7        <li class="mb-2">
8          <LinkTo @route="bands.band.songs" @model={{band.id}}>
9            {{band.name}}
10         </LinkTo>
11       </li>
12     {{/each}}
13   </ul>
+ 14   <LinkTo
+ 15     @route="bands.new"
+ 16     class="inline-block ml-2 mt-2 p-2 rounded bg-purple-600 shadow-md
17       hover:shadow-lg hover:text-white hover:bg-purple-500 focus:outline-
18       none">
+ 17   >
+ 18     Add band
+ 19   </LinkTo>
20 </div>
21 <div class="w-2/3 ppx-2">
22   {{outlet}}
23 </div>
```

HBS

```

1  {{!!-- app/templates/bands/new.hbs --}}
2  {{pageTitle "New band"}}
3
4  <h3 class="text-lg leading-6 font-medium text-gray-100">
5    New band
6  </h3>
7  <div class="mt-6 grid grid-cols-1 row-gap-6 col-gap-4 sm:grid-cols-6">
8    <div class="sm:col-span-4">
9      <label for="name" class="block text-sm font-medium
leading-5">Name</label>
10     <div class="mt-1 relative rounded-md shadow-sm">
11       <input
12         id="name"
13         class="w-full text-gray-800 bg-white border rounded-md py-2
px-4"
14         value="{{this.name}}"
15         {{on "input" this.updateName}}>
16     />
17   </div>
18 </div>
19 <div class="sm:col-span-4">
20   <button
21     type="button"
22     class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-lg
hover:text-white"
23     {{on "click" this.saveBand}}>
24   >
25   Save
26 </button>
27 </div>
28 </div>

```

HBS

As we learned recently, the context of that template is the controller, so to handle the action, we need to create it:

```
1 $ ember g controller bands/new
2 installing controller
3   create app/controllers/bands/new.js
4 installing controller-test
5   create tests/unit/controllers/bands/new-test.js
```

We need to define the `updateName` and `saveBand` actions which should look similar to the `updateTitle` and `saveSong` actions we implemented earlier:

```
1 // app/controllers/bands/new.js
2 import Controller from '@ember/controller';
+ 3 import { action } from '@ember/object';
+ 4 import { dasherize } from '@ember/string';
+ 5 import { tracked } from '@glimmer/tracking';
+ 6 import { Band } from 'rarwe/routes/bands';
7
8 export default class BandsNewController extends Controller {
+ 9   @tracked name;
+ 10
+ 11   @action
+ 12   updateName(event) {
+ 13     this.name = event.target.value;
+ 14   }
+ 15
+ 16   @action
+ 17   saveBand() {
+ 18     new Band({ name: this.name, id: dasherize(this.name) });
+ 19   }
20 }
```

We create the band, using the `dasherize` function to create an identifier from the band's name.

The `Band` class needs to be exported so that we can import it in this file:

```
1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3 import { tracked } from '@glimmer/tracking';
4
- 5 class Band {
+ 6 export class Band {
7     @tracked name;
8     @tracked songs;
9
10    constructor({ id, name, songs }) {
11        this.id = id;
12        this.name = name;
13        this.songs = songs;
14    }
15 }
16
17 export class Song {
18    constructor({ title, rating, band }) {
19        this.title = title;
20        this.rating = rating ?? 0;
21        this.band = band;
22    }
23 }
24
25 export default class BandsRoute extends Route {
26    model() {
27        let blackDog = new Song({
28            title: 'Black Dog',
29            band: 'Led Zeppelin',
30            rating: 3,
31        });
32
33        let yellowLedbetter = new Song({
34            title: 'Yellow Ledbetter',
35            band: 'Pearl Jam',
36            rating: 4,
37        });
38    }
39}
```

```

38
39     let pretender = new Song({
40         title: 'The Pretender',
41         band: 'Foo Fighters',
42         rating: 2,
43     });
44
45     let daughter = new Song({
46         title: 'Daughter',
47         band: 'Pearl Jam',
48         rating: 5,
49     });
50
51     let ledZeppelin = new Band({
52         id: 'led-zeppelin',
53         name: 'Led Zeppelin',
54         songs: [blackDog],
55     });
56
57     let pearlJam = new Band({
58         id: 'pearl-jam',
59         name: 'Pearl Jam',
60         songs: [yellowLedbetter, daughter],
61     });
62
63     let fooFighters = new Band({
64         id: 'foo-fighters',
65         name: 'Foo Fighters',
66         songs: [pretender],
67     });
68
69     return [ledZeppelin, pearlJam, fooFighters];
70 }
71 }
```

We also shouldn't forget to reset the `name` property on the controller when leaving the page (again, just as we did for song creation):

```

1 // app/routes/bands/new.js
2 import Route from '@ember/routing/route';
3
4 export default class BandsNewRoute extends Route {
+ 5   resetController(controller) {
+ 6     controller.name = '';
+ 7   }
8 }
```

There is one problem, though. Just as it was the case when we created songs, the band will not appear in the list. However, here, we have a harder challenge to solve: the `model` hook of the bands route, responsible for showing the list of bands in the sidebar returns a static list of three bands.

When we made new songs appear, we also saw that manually updating a route's `model` (especially if it's not even the current route's) is an anti-pattern. We also can't tweak the model to return a relationship we can add to – like we did for songs.

Somehow the model hook would need to be made reactive. More precisely, the collection it returns would need to be made react to changes by re-rendering.

Related hits

- [Routable components RFC](#)
- [An alternative to Controllers RFC](#)

Next song

A fresh new approach is needed. We'll come up with a mechanism to store bands and songs and access that store in a way that reacts instantly to updates.

CHAPTER 8

Building a catalog

Tuning

Our next mission –should you accept it– is to make the list of bands reactive. Creating a new band should update the list of bands in the sidebar.

Services: useful singletons

In the [chapter on controllers](#) we learned that controllers are singletons. It was more of a hindrance than a feature as controllers are closely bound to UI state and most of the time you want UI state to be cleared when you move away from the page that piece of UI is rendered on.

There are other kinds of states in most, non-trivial applications. For example, you might want to retain data as you move between pages of your application which is not related to your domain models: we can call it application state. A typical example is maintaining a session that holds the current user.

As in most situations, the framework has us covered: services are baked into the framework for this exact reason.

The current task at hand, storing existing bands and songs, seems a fitting use case. If we had a store that can hold them and defined methods on it for access, we'd be able to overcome the problem of the model hooks being too rigid.

Building a store... I mean, a catalog

The first step is to create the store service. For reasons revealed later, we'll choose a different name for our

service and we'll call it "catalog". Our catalog will hold the bands and songs we create in the application.

We can generate a service with Ember CLI:

```
1 $ ember g service catalog
2 installing service
3   create app/services/catalog.js
4 installing service-test
5   create tests/unit/services/catalog-test.js
```

We'll need a way to add items to our catalog, and a way to access each collection:

```

1 // app/services/catalog.js
2 import Service from '@ember/service';
+ 3 import { tracked } from '@glimmer/tracking';
4
5 export default class CatalogService extends Service {
+ 6   @tracked storage = {};
+ 7
+ 8   constructor() {
+ 9     super(...arguments);
+10   this.storage.bands = [];
+11   this.storage.songs = [];
+12 }
+13
+14   add(type, record) {
+15     let collection = type === 'band' ? this.storage.bands :
+16     this.storage.songs;
+17     collection.push(record);
+18   }
+19   get bands() {
+20     return this.storage.bands;
+21   }
+22
+23   get songs() {
+24     return this.storage.songs;
+25   }
26 }
```

We've also added a shortcut pair to return all the bands and all the songs in the catalog.

In the places we create new band and song objects, we should make sure to add them to the catalog.

Let's start with bands. The problem we're trying to fix is that new bands don't appear in the list, so let's open up the bands/new controller and add the band object to the catalog upon creation:

```

1 // app/controllers/bands/new.js
2 import Controller from '@ember/controller';
3 import { action } from '@ember/object';
4 import { dasherize } from '@ember/string';
5 import { tracked } from '@glimmer/tracking';
6 import { Band } from 'rarwe/routes/bands';
+ 7 import { inject as service } from '@ember/service';
8
9 export default class BandsNewController extends Controller {
+ 10   @service catalog;
+ 11
12   @tracked name;
13
14   @action
15   updateName(event) {
16     this.name = event.target.value;
17   }
18
19   @action
20   saveBand() {
- 21     new Band({ name: this.name, id: dasherize(this.name) });
+ 22     let band = new Band({ name: this.name, id: dasherize(this.name)
});+
23     this.catalog.add('band', band);
24   }
25 }
```

The way to get access to services in the app is to inject them (the verb "inject" refers to the underlying dependency injection mechanism). A common way to do this is to alias the imported "inject" function as "service". This makes it very descriptive that it's a service we're using.

Now, we still need to change the route because at the moment we're creating 3 static bands in the route and not adding them to the catalog. We'll also have to return the bands in the catalog so that we can keep "eaching through" `@model` in the template:

```
1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3 import { tracked } from '@glimmer/tracking';
+ 4 import { inject as service } from '@ember/service';
5
6 export class Band {
7     @tracked name;
8     @tracked songs;
9
10    constructor({ id, name, songs }) {
11        this.id = id;
12        this.name = name;
13        this.songs = songs;
14    }
15}
16
17 export class Song {
18    constructor({ title, rating, band }) {
19        this.title = title;
20        this.rating = rating ?? 0;
21        this.band = band;
22    }
23}
24
25 export default class BandsRoute extends Route {
+ 26    @service catalog;
+ 27
28    model() {
29        let blackDog = new Song({
30            title: 'Black Dog',
31            band: 'Led Zeppelin',
32            rating: 3,
33        });
34
35        let yellowLedbetter = new Song({
36            title: 'Yellow Ledbetter',
37            band: 'Pearl Jam',
```

```

38         rating: 4,
39     });
40
41     let pretender = new Song({
42         title: 'The Pretender',
43         band: 'Foo Fighters',
44         rating: 2,
45     });
46
47     let daughter = new Song({
48         title: 'Daughter',
49         band: 'Pearl Jam',
50         rating: 5,
51     });
52
53     let ledZeppelin = new Band({
54         id: 'led-zeppelin',
55         name: 'Led Zeppelin',
56         songs: [blackDog],
57     });
58
59     let pearlJam = new Band({
60         id: 'pearl-jam',
61         name: 'Pearl Jam',
62         songs: [yellowLedbetter, daughter],
63     });
64
65     let fooFighters = new Band({
66         id: 'foo-fighters',
67         name: 'Foo Fighters',
68         songs: [pretender],
69     });
70
- 71     return [ledZeppelin, pearlJam, fooFighters];
+ 72     this.catalog.add('band', ledZeppelin);
+ 73     this.catalog.add('band', pearlJam);
+ 74     this.catalog.add('band', fooFighters);
+ 75

```

```
+ 76     return this.catalog.bands;
77   }
78 }
```

Bummer. The initial bands display fine but when we add a new band it still doesn't show up.

Tracking the right thing

When we mark something as `@tracked` in a class definition, we tell Ember to recompute any values that property is used in *whenever the value of the tracked property updates*. If that property is an object, this doesn't mean that all properties of the tracked object will also be tracked – those have to be tracked individually.

What's happening in our example is that `storage` itself is tracked but we don't change *its* value. We mutate the `bands` array (by pushing a new item into it) which is a property of `storage`. So to make this work, we have to mark the `bands` array as tracked.

To do that, we'll need to install an add-on that provides tracking for all kinds of objects, called `tracked-built-ins`:

```
$ ember i tracked-built-ins
```

We now have the tools to fix our tracking code in the catalog, so let's do it.

```

1 // app/services/catalog.js
2 import Service from '@ember/service';
- 3 import { tracked } from '@glimmer/tracking';
+ 4 import { tracked } from 'tracked-built-ins';
5
6 export default class CatalogService extends Service {
- 7   @tracked storage = {};
+ 8   storage = {};
9
10  constructor() {
11    super(...arguments);
- 12    this.storage.bands = [];
- 13    this.storage.songs = [];
+ 14    this.storage.bands = tracked([]);
+ 15    this.storage.songs = tracked([]);
16  }
17
18  add(type, record) {
19    let collection = type === 'band' ? this.storage.bands :
this.storage.songs;
20    collection.push(record);
21  }
22
23  get bands() {
24    return this.storage.bands;
25  }
26
27  get songs() {
28    return this.storage.songs;
29  }
30}

```

You'll notice `storage` itself doesn't need to be tracked as we don't ever change its value.

Finally, the newly added bands appear in the list.

Rock & Roll with Octane

Led Zeppelin	New band
Pearl Jam	Name
Foo Fighters	<input type="text" value="Soundgarden"/>
Soundgarden	
Add band	Save

A small change to improve user experience is to go to the new band's songs page after creating it.

Now that we're familiar with services, we can leverage one of the built-in ones: the router service. As we'll see later, the router service is a super handy tool for a whole range of things – here, we'll use it to initiate a transition to a new route.

```

1 // app/controllers/bands/new.js
2 import Controller from '@ember/controller';
3 import { action } from '@ember/object';
4 import { dasherize } from '@ember/string';
5 import { tracked } from '@glimmer/tracking';
6 import { Band } from 'rarwe/routes/bands';
7 import { inject as service } from '@ember/service';
8
9 export default class BandsNewController extends Controller {
10   @service catalog;
+ 11   @service router;
12
13   @tracked name;
14
15   @action
16   updateName(event) {
17     this.name = event.target.value;
18   }
19
20   @action
21   saveBand() {
22     let band = new Band({ name: this.name, id: dasherize(this.name) });
23     this.catalog.add('band', band);
+ 24   this.router.transitionTo('bands.band.songs', band.id);
25   }
26 }
```

Use the catalog for querying

Now that we have our band records nicely stashed in the catalog, we can switch our data lookups (currently confined to the `model` hook of routes) to use it. We already switched to it in the `bands` route, so let's now do the same in the `bands.band` route.

We need to find a band with a specific id. We don't support this kind of lookup in our catalog yet so let's add a simple way:

```

1 // app/services/catalog.js
2 import Service from '@ember/service';
3 import { tracked } from 'tracked-built-ins';
4
5 export default class CatalogService extends Service {
6   storage = {};
7
8   constructor() {
9     super(...arguments);
10    this.storage.bands = tracked([]);
11    this.storage.songs = tracked([]);
12  }
13
14  add(type, record) {
15    let collection = type === 'band' ? this.storage.bands :
16      this.storage.songs;
17    collection.push(record);
18  }
19
20  get bands() {
21    return this.storage.bands;
22  }
23
24  get songs() {
25    return this.storage.songs;
26  }
27  find(type, filterFn) {
28    let collection = type === 'band' ? this.bands : this.songs;
29    return collection.find(filterFn);
30  }
31}

```

And then using it in the route:

```

1 // app/routes/bands/band.js
2 import Route from '@ember/routing/route';
+ 3 import { inject as service } from '@ember/service';
4
5 export default class BandsBandRoute extends Route {
+ 6   @service catalog;
+ 7
8   model(params) {
- 9     let bands = this.modelFor('bands');
- 10    return bands.find((band) => band.id === params.id);
+ 11    return this.catalog.find('band', (band) => band.id === params.id);
12  }
13 }

```

Adding songs to the catalog

To wrap up this feature, we should also store our songs in the catalog. We create a handful of them in the bands route, and then whenever a new one is created by the user in the `bands.band.songs` controller.

```

1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3 import { tracked } from '@glimmer/tracking';
4 import { inject as service } from '@ember/service';
5
6 export class Band {
7     @tracked name;
8     @tracked songs;
9
10    constructor({ id, name, songs }) {
11        this.id = id;
12        this.name = name;
13        this.songs = songs;
14    }
15 }
16
17 export class Song {
18    constructor({ title, rating, band }) {
19        this.title = title;
20        this.rating = rating ?? 0;
21        this.band = band;
22    }
23 }
24
25 export default class BandsRoute extends Route {
26     @service catalog;
27
28     model() {
29         let blackDog = new Song({
30             title: 'Black Dog',
31             band: 'Led Zeppelin',
32             rating: 3,
33         });
34
35         let yellowLedbetter = new Song({
36             title: 'Yellow Ledbetter',
37             band: 'Pearl Jam',

```

```

38         rating: 4,
39     });
40
41     let pretender = new Song({
42         title: 'The Pretender',
- 43         band: 'Foo Fighters',
44         rating: 2,
45     });
46
47     let daughter = new Song({
48         title: 'Daughter',
- 49         band: 'Pearl Jam',
50         rating: 5,
51     });
52
53     let ledZeppelin = new Band({
54         id: 'led-zeppelin',
55         name: 'Led Zeppelin',
56         songs: [blackDog],
57     });
58
59     let pearlJam = new Band({
60         id: 'pearl-jam',
61         name: 'Pearl Jam',
62         songs: [yellowLedbetter, daughter],
63     });
64
65     let fooFighters = new Band({
66         id: 'foo-fighters',
67         name: 'Foo Fighters',
68         songs: [pretender],
69     });
70
+ 71     blackDog.band = ledZeppelin;
+ 72     yellowLedbetter.band = pearlJam;
+ 73     daughter.band = pearlJam;
+ 74     pretender.band = fooFighters;
+ 75

```

```
+ 76     this.catalog.add('song', blackDog);
+ 77     this.catalog.add('song', yellowLedbetter);
+ 78     this.catalog.add('song', daughter);
+ 79     this.catalog.add('song', pretender);
+ 80
81     this.catalog.add('band', ledZepelin);
82     this.catalog.add('band', pearlJam);
83     this.catalog.add('band', fooFighters);
84
85     return this.catalog.bands;
86 }
87 }
```

In the process, we also took care of changing the `band` property, which had been set to the name of the band, to the band object.

Adding newly created songs is pretty straightforward:

```

1 // app/controllers/bands/band/songs.js
2 import Controller from '@ember/controller';
3 import { tracked } from '@glimmer/tracking';
4 import { action } from '@ember/object';
5 import { Song } from 'rarwe/routes/bands';
+ 6 import { inject as service } from '@ember/service';
7
8 export default class BandsBandSongsController extends Controller {
9   @tracked showAddSong = true;
10  @tracked title = '';
11
+ 12  @service catalog;
+ 13
14  @action
15  updateTitle(event) {
16    this.title = event.target.value;
17  }
18
19  @action
20  saveSong() {
21    let song = new Song({ title: this.title, band: this.model });
+ 22    this.catalog.add('song', song);
23    this.model.songs = [...this.model.songs, song];
24    this.title = '';
25    this.showAddSong = true;
26  }
27
28  @action
29  cancel() {
30    this.title = '';
31    this.showAddSong = true;
32  }
33}

```

One bug we have to fix is that when we add a new band and then try to add a new song, we get the following error:

Uncaught TypeError: this.model.songs is not iterable

New bands are created without having their songs array properly initialized, so let's fix that:

```

1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3 import { tracked } from '@glimmer/tracking';
4 import { inject as service } from '@ember/service';
5
6 export class Band {
7   @tracked name;
8   @tracked songs;
9
10  constructor({ id, name, songs }) {
11    this.id = id;
12    this.name = name;
13    this.songs = songs;
14 +   this.songs = songs || [];
15  }
16}
17
18 export class Song {
19  constructor({ title, rating, band }) {
20    this.title = title;
21    this.rating = rating ?? 0;
22    this.band = band;
23  }
24}
25
26 export default class BandsRoute extends Route {
27   @service catalog;
28
29   model() {
30     let blackDog = new Song({
31       title: 'Black Dog',
32       rating: 3,
33     });
34
35     let yellowLedbetter = new Song({
36       title: 'Yellow Ledbetter',
37       rating: 4,

```

```

38     });
39
40     let pretender = new Song({
41         title: 'The Pretender',
42         rating: 2,
43     });
44
45     let daughter = new Song({
46         title: 'Daughter',
47         rating: 5,
48     });
49
50     let ledZeppelin = new Band({
51         id: 'led-zeppelin',
52         name: 'Led Zeppelin',
53         songs: [blackDog],
54     });
55
56     let pearlJam = new Band({
57         id: 'pearl-jam',
58         name: 'Pearl Jam',
59         songs: [yellowLedbetter, daughter],
60     });
61
62     let fooFighters = new Band({
63         id: 'foo-fighters',
64         name: 'Foo Fighters',
65         songs: [pretender],
66     });
67
68     blackDog.band = ledZeppelin;
69     yellowLedbetter.band = pearlJam;
70     daughter.band = pearlJam;
71     pretender.band = fooFighters;
72
73     this.catalog.add('song', blackDog);
74     this.catalog.add('song', yellowLedbetter);
75     this.catalog.add('song', daughter);

```

```
76     this.catalog.add('song', pretender);
77
78     this.catalog.add('band', ledZeppelin);
79     this.catalog.add('band', pearlJam);
80     this.catalog.add('band', fooFighters);
81
82     return this.catalog.bands;
83 }
84 }
```

Extracting model classes

It feels weird that we still have our domain models defined in a route, so let's extract them in their proper files:

```
1 // app/models/band.js
2 import { tracked } from '@glimmer/tracking';
3
4 export default class Band {
5   @tracked name;
6   @tracked songs;
7
8   constructor({ id, name, songs }) {
9     this.id = id;
10    this.name = name;
11    this.songs = songs || [];
12  }
13}
```

```
1 // app/models/song.js
2 export default class Song {
3   constructor({ title, rating, band }) {
4     this.title = title;
5     this.rating = rating ?? 0;
6     this.band = band;
7   }
8 }
```

In the bands route, we should remove the model definitions and import them:

```

1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
- 3 import { tracked } from '@glimmer/tracking';
4 import { inject as service } from '@ember/service';
- 5
- 6 export class Band {
- 7   @tracked name;
- 8   @tracked songs;
- 9
- 10  constructor({ id, name, songs }) {
- 11    this.id = id;
- 12    this.name = name;
- 13    this.songs = songs || [];
- 14  }
- 15}
- 16
- 17 export class Song {
- 18  constructor({ title, rating, band }) {
- 19    this.title = title;
- 20    this.rating = rating ?? 0;
- 21    this.band = band;
- 22  }
- 23}
+ 24 import Band from 'rarwe/models/band';
+ 25 import Song from 'rarwe/models/song';
26
27 export default class BandsRoute extends Route {
28   @service catalog;
29
30   model() {
31     let blackDog = new Song({
32       title: 'Black Dog',
33       rating: 3,
34     });
35
36     let yellowLedbetter = new Song({
37       title: 'Yellow Ledbetter',

```

```

38     rating: 4,
39   });
40
41   let pretender = new Song({
42     title: 'The Pretender',
43     rating: 2,
44   });
45
46   let daughter = new Song({
47     title: 'Daughter',
48     rating: 5,
49   });
50
51   let ledZeppelin = new Band({
52     id: 'led-zeppelin',
53     name: 'Led Zeppelin',
54     songs: [blackDog],
55   });
56
57   let pearlJam = new Band({
58     id: 'pearl-jam',
59     name: 'Pearl Jam',
60     songs: [yellowLedbetter, daughter],
61   });
62
63   let fooFighters = new Band({
64     id: 'foo-fighters',
65     name: 'Foo Fighters',
66     songs: [pretender],
67   });
68
69   blackDog.band = ledZeppelin;
70   yellowLedbetter.band = pearlJam;
71   daughter.band = pearlJam;
72   pretender.band = fooFighters;
73
74   this.catalog.add('song', blackDog);
75   this.catalog.add('song', yellowLedbetter);

```

```
76     this.catalog.add('song', daughter);
77     this.catalog.add('song', pretender);
78
79     this.catalog.add('band', ledZeppelin);
80     this.catalog.add('band', pearlJam);
81     this.catalog.add('band', fooFighters);
82
83     return this.catalog.bands;
84 }
85 }
```

```
1 // app/controllers/bands/new.js
2 import Controller from '@ember/controller';
3 import { action } from '@ember/object';
4 import { dasherize } from '@ember/string';
5 import { tracked } from '@glimmer/tracking';
6 import { Band } from 'rarwe/routes/bands';
+ 7 import Band from 'rarwe/models/band';
8 import { inject as service } from '@ember/service';
9
10 export default class BandsNewController extends Controller {
11     @service catalog;
12     @service router;
13
14     @tracked name;
15
16     @action
17     updateName(event) {
18         this.name = event.target.value;
19     }
20
21     @action
22     saveBand() {
23         let band = new Band({ name: this.name, id: dasherize(this.name) });
24         this.catalog.add('band', band);
25         this.router.transitionTo('bands.band.songs', band.id);
26     }
27 }
```

```

1 // app/controllers/bands/band/songs.js
2 import Controller from '@ember/controller';
3 import { tracked } from '@glimmer/tracking';
4 import { action } from '@ember/object';
5 - import { Song } from 'rarwe/routes/bands';
+ 6 import Song from 'rarwe/models/song';
7 import { inject as service } from '@ember/service';
8
9 export default class BandsBandSongsController extends Controller {
10   @tracked showAddSong = true;
11   @tracked title = '';
12
13   @service catalog;
14
15   @action
16   updateTitle(event) {
17     this.title = event.target.value;
18   }
19
20   @action
21   saveSong() {
22     let song = new Song({ title: this.title, band: this.model });
23     this.catalog.add('song', song);
24     this.model.songs = [...this.model.songs, song];
25     this.title = '';
26     this.showAddSong = true;
27   }
28
29   @action
30   cancel() {
31     this.title = '';
32     this.showAddSong = true;
33   }
34 }
```

Next song

We've built a catalog service to make our collections (and model-based UI) reactive: we'll also be able to extend the catalog should we need to add other functionality to our app.

CHAPTER 9

Talking to a back-end

Tuning

Most actual, in-the-wild apps talk to back-ends to retrieve data from and send data to, so it'd be a logical next step to make our app do so. We'll start by turning data fetches to hit a back-end and will then do some refactoring to make it nicer (and more manageable on the long-run).

The back-end

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.

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 are starting a project today, I recommend using it.

The API is available at <http://json-api.rockandrollwithemberjs.com>.

It's very important to point out that Ember doesn't require the back-end to adhere to any specific convention or to be written in any specific language. It can work with JSON:API, your home-rolled slightly RESTy PHP API, or GraphQL.

Ember CLI has a command line `proxy` option that makes it a cinch to connect to remote back-ends. If given, all network requests are proxied to the given host. This means we have to start our server differently from

now on:

```
1 $ ember s --proxy=http://json-api.rockandrollwithemberjs.com
2 Proxying to http://json-api.rockandrollwithemberjs.com
3
4 Build successful (14943ms) - Serving on http://localhost:4200/
```

Replacing the loading and displaying of bands

Following the flow of data in our application, it makes sense to start with converting the loading of bands. Instead of using the list of POJOs, we'll issue a fetch request and use the list of bands in the response as our primary data. All this takes place in the `bands` route, so let's open that file and paste in the following content:

```

1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
4 import Band from 'rarwe/models/band';
5 import fetch from 'fetch';
6
7 export default class BandsRoute extends Route {
8   @service catalog;
9
10  async model() {
11    let response = await fetch('/bands');
12    let json = await response.json();
13    for (let item of json.data) {
14      let { id, attributes } = item;
15      let record = new Band({ id, ...attributes });
16      this.catalog.add('band', record);
17    }
18    return this.catalog.bands;
19  }
20}

```

Following REST conventions, the list of bands needs to be retrieved from `/bands/`. We don't have to bother prepending the host: the `proxy` option we launched Ember CLI with takes care of that. `fetch` returns a promise and the data in the response can be had by calling its `json` method (which also returns a promise).

The response has the following format:

```

1  {
2      "data": [
3          {
4              "id": "pearl-jam",
5              "type": "bands",
6              "links": {
7                  "self": "http://localhost:4200/bands/pearl-jam"
8              },
9              "attributes": {
10                  "name": "Pearl Jam",
11                  "description": "Pearl Jam is an American rock band, formed in
Seattle, Washington in 1990."
12              },
13              "relationships": {
14                  "songs": {
15                      "links": {
16                          "self": "http://localhost:4200/bands/pearl-jam/
relationships/songs",
17                          "related": "http://localhost:4200/bands/pearl-jam/songs"
18                      }
19                  }
20              }
21          },
22          { ... },
23          { ... },
24      ]
25  }

```

js

As stated above, it follows the JSON:API convention, having a top-level `data` attribute. For each item in the response, we create a band record, and add it to the catalog so that we can then display all bands therein, just as before.

You'll notice the list of bands has changed – they are now fetched from our API:

Rock & Roll with Octane

Foo Fighters
Kaya Project
Led Zeppelin
Pearl Jam
Radiohead
Red Hot Chili Peppers

Select a band.

Add band

You might wonder why we need to import `fetch` when it's built into all modern browsers. The `ember-fetch` package (installed by default in all new Ember apps) adds a wrapper around `fetch` and that's what we import. If you want to learn more about why that's needed, see the "Why is this wrapper needed?" section in the addon's [README](#).

Creating a band – for real, this time

The tried-and-true way of creating a new record against a REST back-end is to send a POST request. JSON:API has the following to say about how to do that:

A resource can be created by sending a POST request to a URL that represents a collection of resources. The request MUST include a single resource object as primary data. The resource object MUST contain at least a type member.

(Don't be intimidated by the all-caps MUSTs. JSON:API is not shouting at you – it's a convention that denotes requirement levels.)

Since we already established that the URL that represents the collection of bands is `/bands`, we can deduce we need to send a POST request there. Let's not waste any more time:

```

1 // app/controllers/bands/new.js
2 import Controller from '@ember/controller';
3 import { action } from '@ember/object';
- 4 import { dasherize } from '@ember/string';
5 import { tracked } from '@glimmer/tracking';
6 import Band from 'rarwe/models/band';
7 import { inject as service } from '@ember/service';
+ 8 import fetch from 'fetch';

9
10 export default class BandsNewController extends Controller {
11     @service catalog;
12     @service router;
13
14     @tracked name;
15
16     @action
17     updateName(event) {
18         this.name = event.target.value;
19     }
20
21     @action
- 22     saveBand() {
- 23         let band = new Band({ name: this.name, id: dasherize(this.name) });
- 24         this.catalog.add('band', band);
- 25         this.router.transitionTo('bands.band.songs', band.id);
+ 26     async saveBand() {
+ 27         let response = await fetch('/bands', {
+ 28             method: 'POST',
+ 29             headers: {
+ 30                 'Content-Type': 'application/vnd.api+json',
+ 31             },
+ 32             body: JSON.stringify({
+ 33                 data: {
+ 34                     type: 'bands',
+ 35                     attributes: {
+ 36                         name: this.name,

```

```

+ 37     },
+ 38   },
+ 39   }) ,
+ 40 });
+ 41 let json = await response.json();
+ 42 let { id, attributes } = json.data;
+ 43 let record = new Band({ id, ...attributes });
+ 44 this.catalog.add('band', record);
+ 45 this.router.transitionTo('bands.band.songs', id);

46 }
47 }

```

JSON:API demands that create requests specify the JSON:API media type, `application/vnd.api+json` and that the payload be a JSON:API representation of the resource to be created. Other than that, the code is really similar to how we extracted client-side records when we fetched all bands. This is a hint of a possible refactoring to clear things up.

One thing to notice is that creating the id of the band has now become the responsibility of the back-end: the response contains a top-level `id` and the `attributes`. It's by putting these two together that we can create our `Band` object.

Where did the songs go? (loading related data)

It all seems fine but we quickly realize that songs are not correctly loaded for any of the bands. When we click any of the band links in the sidebar no songs are displayed. Taking a look at the network response for `/bands`, we see a `relationships` attribute for each band item. Digging further down, we find a URL for the `songs` relationship where related records should be loaded from. However, no request is made to any of those URLs – which makes sense, as we didn't write any code to do that.

Name	Headers	Preview	Response	Initiator	Timing	Cookies
bands			<pre>▼{...} ▼data: [{id: "foo-fighters", type: "bands", links: {self: "http://localhost:4200/bands/foo-fighters"}, id: "foo-fighters", name: "Foo Fighters"}, {id: "kaya-project", type: "bands", links: {self: "http://localhost:4200/bands/kaya-project"}, id: "kaya-project", name: "Kaya Project"}, {id: "led-zeppelin", type: "bands", links: {self: "http://localhost:4200/bands/led-zeppelin"}, id: "led-zeppelin", name: "Led Zeppelin"}, {id: "pearl-jam", type: "bands", links: {self: "http://localhost:4200/bands/pearl-jam"}, id: "pearl-jam", name: "Pearl Jam"}] ▶0: {id: "foo-fighters", type: "bands", links: {self: "http://localhost:4200/bands/foo-fighters"}, id: "foo-fighters", name: "Foo Fighters"} ▶1: {id: "kaya-project", type: "bands", links: {self: "http://localhost:4200/bands/kaya-project"}, id: "kaya-project", name: "Kaya Project"} ▶2: {id: "led-zeppelin", type: "bands", links: {self: "http://localhost:4200/bands/led-zeppelin"}, id: "led-zeppelin", name: "Led Zeppelin"} ▶3: {id: "pearl-jam", type: "bands", links: {self: "http://localhost:4200/bands/pearl-jam"}, id: "pearl-jam", name: "Pearl Jam"} ▶attributes: {name: "Pearl Jam", type: "band"} id: "pearl-jam" ▶links: {self: "http://localhost:4200/bands/pearl-jam"} ▼relationships: {songs: {links: {self: "http://localhost:4200/bands/pearl-jam/relationships/songs"}, id: "songs", name: "Songs", type: "relationship"}, self: "http://localhost:4200/bands/pearl-jam/relationships/songs", type: "band"} ▼songs: {links: {self: "http://localhost:4200/bands/pearl-jam/relationships/songs"}, id: "songs", name: "Songs", type: "relationship"}, self: "http://localhost:4200/bands/pearl-jam/relationships/songs", type: "band" ▶links: {self: "http://localhost:4200/bands/pearl-jam/relationships/songs"}, id: "songs", name: "Songs", type: "relationship"}, self: "http://localhost:4200/bands/pearl-jam/relationships/songs", type: "band" related: "http://localhost:4200/bands/pearl-jam/songs" self: "http://localhost:4200/bands/pearl-jam/relationships/songs", type: "band" +type: "bands" </pre>			
1 / 40 requests 717						

Since we need the band to have its related songs properly loaded to render the songs template, it makes sense to add that to the bands.band.songs route. In order to do that, though, we first have to store the relationship link for each band when we create them. This way, we can load the data for the related relationship when we need to by fetching data from the stored URL.

Let's begin by taking a second argument in our model's constructor that contains the relationship links:

```

1 // app/models/band.js
2 import { tracked } from '@glimmer/tracking';
3
4 export default class Band {
5   @tracked name;
6   @tracked songs;
7
8 -  constructor({ id, name, songs }) {
9 +  constructor({ id, name, songs }, relationships = {}) {
10    this.id = id;
11    this.name = name;
12    this.songs = songs || [];
13    this.relationships = relationships;
14  }
15}

```

We can now extract the relationships for each band. Each relationship is under a nested key in the response: `relationships.<relationshipName>.link.related`. The following code assigns each relationship link under the `relationships` attribute of Band model instances, so songs will be at `relationships.songs`:

```
1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
4 import Band from 'rarwe/models/band';
5 import fetch from 'fetch';
6
7 export default class BandsRoute extends Route {
8   @service catalog;
9
10  async model() {
11    let response = await fetch('/bands');
12    let json = await response.json();
13    for (let item of json.data) {
14      let { id, attributes } = item;
15      let record = new Band({ id, ...attributes });
16      let { id, attributes, relationships } = item;
17      let rels = {};
18      for (let relationshipName in relationships) {
19        rels[relationshipName] =
20          relationships[relationshipName].links.related;
21      }
22      let record = new Band({ id, ...attributes }, rels);
23      this.catalog.add('band', record);
24    }
25  }
26 }
```

Armed with the link to fetch data for the `songs` relationship from, we can proceed to update the `model` hook of the `bands.band.songs` route.

```

1 // app/routes/bands/band/songs.js
2 import Route from '@ember/routing/route';
+ 3 import { inject as service } from '@ember/service';
+ 4 import Song from 'rarwe/models/song';
+ 5 import fetch from 'fetch';
6
7 export default class BandsBandSongsRoute extends Route {
+ 8   @service catalog;
+ 9
+ 10  async model() {
+ 11    let band = this.modelFor('bands.band');
+ 12    let url = band.relationships.songs;
+ 13    let response = await fetch(url);
+ 14    let json = await response.json();
+ 15    let songs = [];
+ 16    for (let item of json.data) {
+ 17      let { id, attributes, relationships } = item;
+ 18      let rels = {};
+ 19      for (let relationshipName in relationships) {
+ 20        rels[relationshipName] =
relationships[relationshipName].links.related;
+ 21      }
+ 22      let song = new Song({ id, ...attributes }, rels);
+ 23      songs.push(song);
+ 24      this.catalog.add('song', song);
+ 25    }
+ 26    band.songs = songs;
+ 27    return band;
+ 28  }
+ 29
30  resetController(controller) {
31    controller.title = '';
32    controller.showAddSong = true;
33  }
34}

```

We query the url to retrieve the songs. Very similarly to how we did for bands, we go through each item in

the `data` key of the response, and create a song instance from each. The created songs need to be assigned to the `songs` attribute of the band instance as we still display `@model.songs` in the template.

Our refactoring muscles are now itching to get their well-deserved exercise but we still hold back for a little while. We first have to make the above work by enhancing the `Song` class to take an id, and the `relationships` parameter:

```
1 // app/models/song.js
2 export default class Song {
3   constructor({ title, rating, band }) {
4     constructor({ id, title, rating, band }, relationships = {}) {
5       this.id = id;
6       this.title = title;
7       this.rating = rating ?? 0;
8       this.band = band;
9       this.relationships = relationships;
10    }
11 }
```

Take a look at the list of Pearl Jam songs now:

The screenshot shows a dark-themed web application titled "Rock & Roll with Octane". On the left, there's a sidebar with a purple "Add band" button. The main content area lists bands on the left and their songs on the right, each accompanied by a star rating icon. The bands listed are Foo Fighters, Kaya Project, Led Zeppelin, Pearl Jam, Radiohead, and Red Hot Chili Peppers. Their corresponding songs are Yellow Ledbetter, Daughter, Animal, State of Love and Trust, Alive, and Inside Job. To the right of the song names are five-star rating icons, with the last two having one star missing (indicating a rating of 4.5).

Band	Song	Rating
Foo Fighters	Yellow Ledbetter	★★★★★
Kaya Project	Daughter	★★★★★
Led Zeppelin	Animal	★★★★★
Pearl Jam	State of Love and Trust	★★★★☆
Radiohead	Alive	★★★★☆
Red Hot Chili Peppers	Inside Job	★★★★☆

Creating songs

The next operation we move to the back-end is the creation of songs.

```

1 // app/controllers/bands/band/songs.js
2 import Controller from '@ember/controller';
3 import { tracked } from '@glimmer/tracking';
4 import { action } from '@ember/object';
5 import Song from 'rarwe/models/song';
6 import { inject as service } from '@ember/service';
+ 7 import fetch from 'fetch';
8
9 export default class BandsBandSongsController extends Controller {
10   @tracked showAddSong = true;
11   @tracked title = '';
12
13   @service catalog;
14
15   @action
16   updateTitle(event) {
17     this.title = event.target.value;
18   }
19
20   @action
- 21   saveSong() {
- 22     let song = new Song({ title: this.title, band: this.model });
+ 23   async saveSong() {
+ 24     let payload = {
+ 25       data: {
+ 26         type: 'songs',
+ 27         attributes: { title: this.title },
+ 28         relationships: {
+ 29           band: {
+ 30             data: {
+ 31               id: this.model.id,
+ 32               type: 'bands'
+ 33             }
+ 34           }
+ 35         }
+ 36       }
+ 37     };

```

```

+ 38     let response = await fetch('/songs', {
+ 39       method: 'POST',
+ 40       headers: {
+ 41         'Content-Type': 'application/vnd.api+json'
+ 42       },
+ 43       body: JSON.stringify(payload)
+ 44     });
+ 45
+ 46     let json = await response.json();
+ 47     let { id, attributes, relationships } = json.data;
+ 48     let rels = {};
+ 49     for (let relationshipName in relationships) {
+ 50       rels[relationshipName] =
+ 51         relationships[relationshipName].links.related;
+ 52       let song = new Song({ id, ...attributes }, rels);
+ 53       this.catalog.add('song', song);
+ 54
+ 55       this.model.songs = [...this.model.songs, song];
+ 56       this.title = '';
+ 57       this.showAddSong = true;
+ 58     }
+ 59
+ 60     @action
+ 61     cancel() {
+ 62       this.title = '';
+ 63       this.showAddSong = true;
+ 64     }
+ 65   }

```

This is a hefty chunk of code but most of it should be familiar from when we created bands just a few beers ago. The novel bit is that we're now aware that relationships should be managed and thus we create the proper `band` relationship off the bat. The designation of a related record by specifying its `id` and `type` is called "resource linkage" in the JSON:API vocabulary.

We should really start refactoring all that duplicated code, but before we do that, let's convert the last back-end operation, updating song ratings. We'll then have the whole picture of what needs to be refactored or improved departing from the "it just works" state of the code.

Updating song ratings

The current action that handles clicking a star is dead-simple: it just updates the song's rating using the `set` template helper. We'll need to make it more complicated as it needs to update the song's rating on the back-end, too. This means we'll have to write an action handler function:

```

1 // app/controllers/bands/band/songs.js
2 import Controller from '@ember/controller';
3 import { tracked } from '@glimmer/tracking';
4 import { action } from '@ember/object';
5 import Song from 'rarwe/models/song';
6 import { inject as service } from '@ember/service';
7 import fetch from 'fetch';
8
9 export default class BandsBandSongsController extends Controller {
10   @tracked showAddSong = true;
11   @tracked title = '';
12
13   @service catalog;
14
+ 15   @action
+ 16   async updateRating(song, rating) {
+ 17     song.rating = rating;
+ 18     let payload = {
+ 19       data: {
+ 20         id: song.id,
+ 21         type: 'songs',
+ 22         attributes: {
+ 23           rating
+ 24         }
+ 25       }
+ 26     };
+ 27     await fetch(`/songs/${song.id}`, {
+ 28       method: 'PATCH',
+ 29       headers: {
+ 30         'Content-Type': 'application/vnd.api+json'
+ 31       },
+ 32       body: JSON.stringify(payload)
+ 33     });
+ 34   }
+ 35
36   @action
37   updateTitle(event) {

```

```

38     this.title = event.target.value;
39 }
40
41 @action
42 async saveSong() {
43     let payload = {
44         data: {
45             type: 'songs',
46             attributes: { title: this.title },
47             relationships: {
48                 band: {
49                     data: {
50                         id: this.model.id,
51                         type: 'bands'
52                     }
53                 }
54             }
55         }
56     };
57     let response = await fetch('/songs', {
58         method: 'POST',
59         headers: {
60             'Content-Type': 'application/vnd.api+json'
61         },
62         body: JSON.stringify(payload)
63     });
64
65     let json = await response.json();
66     let { id, attributes, relationships } = json.data;
67     let rels = {};
68     for (let relationshipName in relationships) {
69         rels[relationshipName] =
relationships[relationshipName].links.related;
70     }
71     let song = new Song({ id, ...attributes }, rels);
72     this.catalog.add('song', song);
73
74     this.model.songs = [...this.model.songs, song];

```

```
75     this.title = '';
76     this.showAddSong = true;
77 }
78
79 @action
80 cancel() {
81     this.title = '';
82     this.showAddSong = true;
83 }
84 }
```

As this is a PATCH request, we only need to send the changes: the new rating.

This new `updateRating` function needs to be called from the template, so let's make the modification there:

```

1  {{!!-- app/templates/bands/band/songs.hbs --}}
2  {{pageTitle @model.name "songs | Rock & Roll with Octane"
replace=true}}
3
4  {{#if @model.songs.length}}
5    <ul>
6      {{#each @model.songs as |song|}}
7        <li class="mb-2">
8          {{song.title}}
9          <span class="float-right">
10            <StarRating
11              @rating={{song.rating}}
12              @onUpdate={{set song "rating"}}>
13              @onUpdate={{fn this.updateRating song}}>
14            />
15          </span>
16        </li>
17      {{/each}}
18    </ul>
19  {{else}}
20    <p class="text-center">
21      The band has no songs yet.
22    </p>
23  {{/if}}
24  {{#if this.showAddSong}}
25    <div class="flex justify-center mt-2">
26      <button
27        type="button"
28        class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-lg
hover:text-white hover:bg-purple-500 focus:outline-none"
29        {{on "click" (set this "showAddSong" false)}}>
30      >
31        Add song
32      </button>
33    </div>
34  {{else}}
35    <div class="mt-6 flex">

```

```

36   <input
37     type="text"
38     class="text-gray-800 bg-white rounded-md py-2 px-4"
39     placeholder="Title"
40     value={{this.title}}
41     {{on "input" this.updateTitle}}
42   />
43   <button
44     type="button"
45     class="ml-4 px-4 py-2 rounded bg-purple-600 shadow-md
46       hover:shadow-lg hover:text-white"
47     {{on "click" this.saveSong}}
48   >
49     Save
50   </button>
51   <button
52     type="button"
53     class="ml-2 px-4 p-2 rounded bg-white border border-bg-
54       purple-600 shadow-md text-purple-600 hover:shadow-lg"
55     {{on "click" this.cancel}}
56   >
57     Cancel
58   </button>
59 </div>
60 {{/if}}

```

We might recall that the passed in `onUpdate` function will be called with the star value the user clicked on (= the new rating). We saw in [the Components chapter](#) that the `fn` helper creates a partially applied function: `updateRating` with its first argument being bound to `song`. This is the function that'll be called with the new rating, which will become the second argument. This is exactly how we need it as you can see from the `updateRating` function's signature.

When we test this change by clicking on one of the stars, nothing happens: the number of highlighted stars doesn't change.

The code line `star.rating = rating` updates the star object's rating to the new value but since the `rating` property is not designated to be tracked, the template doesn't update. So let's make it so:

```

1 // app/models/song.js
+ 2 import { tracked } from '@glimmer/tracking';
+ 3
4 export default class Song {
+ 5   @tracked rating;
+ 6
7   constructor({ id, title, rating, band }, relationships = {}) {
8     this.id = id;
9     this.title = title;
10    this.rating = rating ?? 0;
11    this.band = band;
12    this.relationships = relationships;
13  }
14}

```

Making it shippable

Before embarking on our refactoring spree, let's take stock of all the things we want to improve:

- Fetching data from the back-end, and loading records extracted from the response into the catalog
- Creation of model instances, and saving them to the back-end
- Updating existing model instances
- Fetching relationships

On top of refactorings, there's also currently a bug. When we create a band, we don't arrive at its songs page due to an error on the console: since the URL for the `songs` relationship is not set for the new record, it fails to load data from there. The fix for that bug might fall out of our refactoring, though. One of the advantages of refactoring is that –if done properly– we can't forget to copy-paste a given chunk of code.

Given that our data operations live on the catalog, it makes sense to carry out most of the refactoring outlined above by adding methods to the catalog.

Fetching data from the back-end

Let's call the new catalog method `fetchAll`:

```

1 // app/services/catalog.js
2 import Service from '@ember/service';
+ 3 import Band from 'rarwe/models/band';
4 import { tracked } from 'tracked-built-ins';
5
+ 6 function extractRelationships(object) {
+ 7   let relationships = {};
+ 8   for (let relationshipName in object) {
+ 9     relationships[relationshipName] =
10       object[relationshipName].links.related;
+ 11   }
12   return relationships;
+ 13 }
14
14 export default class CatalogService extends Service {
15   storage = {};
16
17   constructor() {
18     super(...arguments);
19     this.storage.bands = tracked([]);
20     this.storage.songs = tracked([]);
21   }
22
+ 23   async fetchAll() {
+ 24     let response = await fetch('/bands');
+ 25     let json = await response.json();
+ 26     for (let item of json.data) {
+ 27       let { id, attributes, relationships } = item;
+ 28       let rels = extractRelationships(relationships);
+ 29       let record = new Band({ id, ...attributes}, rels);
+ 30       this.add('band', record);
+ 31     }
+ 32     return this.bands;
+ 33   }
+ 34 }
35
35 add(type, record) {
36   let collection = type === 'band' ? this.storage.bands :

```

```

36     this.storage.songs;
37     collection.push(record);
38   }
39
40   get bands() {
41     return this.storage.bands;
42   }
43
44   get songs() {
45     return this.storage.songs;
46   }
47
48   find(type, filterFn) {
49     let collection = type === 'band' ? this.bands : this.songs;
50     return collection.find(filterFn);
51   }
52 }

```

We mostly just moved the code to the catalog, while extracting an `extractRelationships` function which we also expect to use from other places.

We can really slim down the `model` hook of the route which this way becomes very intention revealing:

```

1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
- 4 import Band from 'rarwe/models/band';
- 5 import fetch from 'fetch';
6
7 export default class BandsRoute extends Route {
8   @service catalog;
9
- 10  async model() {
- 11    let response = await fetch('/bands');
- 12    let json = await response.json();
- 13    for (let item of json.data) {
- 14      let { id, attributes, relationships } = item;
- 15      let rels = {};
- 16      for (let relationshipName in relationships) {
- 17        rels[relationshipName] =
- 18          relationships[relationshipName].links.related;
- 19        let record = new Band({ id, ...attributes }, rels);
- 20        this.catalog.add('band', record);
- 21      }
- 22      return this.catalog.bands;
+ 23    model() {
+ 24      return this.catalog.fetchAll();
25    }
26  }

```

Although at the moment we don't need to fetch all songs in the app, the hard-coded assumption about always fetching bands in `fetchAll` seems weird, so let's adapt it to take a `type` parameter:

```

1 // app/services/catalog.js
2 import Service from '@ember/service';
3 import Band from 'rarwe/models/band';
+ 4 import Song from 'rarwe/models/song';
5 import { tracked } from 'tracked-built-ins';
6
7 function extractRelationships(object) {
8     let relationships = {};
9     for (let relationshipName in object) {
10         relationships[relationshipName] =
11             object[relationshipName].links.related;
12     }
13     return relationships;
14 }
15
16 export default class CatalogService extends Service {
17     storage = {};
18
19     constructor() {
20         super(...arguments);
21         this.storage.bands = tracked([]);
22         this.storage.songs = tracked([]);
23     }
24
25     async fetchAll() {
26         let response = await fetch('/bands');
27         let json = await response.json();
28         for (let item of json.data) {
29             let { id, attributes, relationships } = item;
30             let rels = extractRelationships(relationships);
31             let record = new Band({ id, ...attributes}, rels);
32             this.add('band', record);
33     }
34     if (type === 'bands') {
35         let response = await fetch('/bands');
36         let json = await response.json();
37         for (let item of json.data) {

```

```

+ 37         let { id, attributes, relationships } = item;
+ 38         let rels = extractRelationships(relationships);
+ 39         let record = new Band({ id, ...attributes}, rels);
+ 40         this.add('band', record);
+ 41     }
+ 42     return this.bands;
+ 43 }
+ 44 if (type === 'songs') {
+ 45     let response = await fetch('/songs');
+ 46     let json = await response.json();
+ 47     for (let item of json.data) {
+ 48         let { id, attributes, relationships } = item;
+ 49         let rels = extractRelationships(relationships);
+ 50         let record = new Song({ id, ...attributes}, rels);
+ 51         this.add('song', record);
+ 52     }
+ 53     return this.songs;
54 }
- 55 return this.bands;
56 }
57
58 add(type, record) {
59     let collection = type === 'band' ? this.storage.bands :
this.storage.songs;
60     collection.push(record);
61 }
62
63 get bands() {
64     return this.storage.bands;
65 }
66
67 get songs() {
68     return this.storage.songs;
69 }
70
71 find(type, filterFn) {
72     let collection = type === 'band' ? this.bands : this.songs;
73     return collection.find(filterFn);

```

```
74      }
75  }
```

While this will work, there's still a lot of unnecessary duplication so let's go deeper and also define a separate function that creates new records from json data, and adds them to the catalog.

```

1 // app/services/catalog.js
2 import Service from '@ember/service';
3 import Band from 'rarwe/models/band';
4 import Song from 'rarwe/models/song';
5 import { tracked } from 'tracked-built-ins';
6
7 function extractRelationships(object) {
8     let relationships = {};
9     for (let relationshipName in object) {
10         relationships[relationshipName] =
11             object[relationshipName].links.related;
12     }
13     return relationships;
14 }
15
16 export default class CatalogService extends Service {
17     storage = {};
18
19     constructor() {
20         super(...arguments);
21         this.storage.bands = tracked([]);
22         this.storage.songs = tracked([]);
23     }
24
25     async fetchAll(type) {
26         if (type === 'bands') {
27             let response = await fetch('/bands');
28             let json = await response.json();
29             for (let item of json.data) {
30                 let { id, attributes, relationships } = item;
31                 let rels = extractRelationships(relationships);
32                 let record = new Band({ id, ...attributes }, rels);
33                 this.add('band', record);
34             }
35             this.loadAll(json);
36         }
37     }
38 }
```

```

37     if (type === 'songs') {
38         let response = await fetch('/songs');
39         let json = await response.json();
40         for (let item of json.data) {
41             let { id, attributes, relationships } = item;
42             let rels = extractRelationships(relationships);
43             let record = new Song({ id, ...attributes }, rels);
44             this.add('song', record);
45         }
46         this.loadAll(json);
47         return this.songs;
48     }
49 }
50
+ 51     loadAll(json) {
+ 52         let records = [];
+ 53         for (let item of json.data) {
+ 54             records.push(this._loadResource(item));
+ 55         }
+ 56         return records;
+ 57     }
+ 58
+ 59     _loadResource(data) {
+ 60         let record;
+ 61         let { id, type, attributes, relationships } = data;
+ 62         if (type === 'bands') {
+ 63             let rels = extractRelationships(relationships);
+ 64             record = new Band({ id, ...attributes }, rels);
+ 65             this.add('band', record);
+ 66         }
+ 67         if (type === 'songs') {
+ 68             let rels = extractRelationships(relationships);
+ 69             record = new Song({ id, ...attributes }, rels);
+ 70             this.add('song', record);
+ 71         }
+ 72         return record;
+ 73     }
+ 74 }
```

```

75     add(type, record) {
76       let collection = type === 'band' ? this.storage.bands :
77         this.storage.songs;
78       collection.push(record);
79     }
80   }
81
82   get bands() {
83     return this.storage.bands;
84   }
85
86   get songs() {
87     return this.storage.songs;
88   }
89
90   find(type, filterFn) {
91     let collection = type === 'band' ? this.bands : this.songs;
92     return collection.find(filterFn);
93   }
94 }
```

The low-level `_loadResource` takes care of all the details of creating a new model record and adding it to the catalog so `fetchAll` can be kept rather tidy. The method takes a single parameter, the type of resources to be fetched:

```

1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
4
5 export default class BandsRoute extends Route {
6   @service catalog;
7
8   model() {
9     return this.catalog.fetchAll();
+ 10    return this.catalog.fetchAll('bands');
11  }
12}
```

This means we're done with the first item on our list:

- ~~Fetching data from the back-end, and loading records extracted from the response into the catalog~~
- Creating model instances, and saving them to the back-end
- Updating existing model instances
- Fetching relationships

Creating model instances

While building the "just works" version of adding communication with the back-end, we realized some code is duplicated between saving a band and a song. The catalog again seems a natural place to add this functionality as we made it be the responsible party for back-end communication. Let's name the method that creates a new model instance and saves it to the back-end `create`.

```

1 // app/services/catalog.js
2 import Service from '@ember/service';
3 import Band from 'rarwe/models/band';
4 import Song from 'rarwe/models/song';
5 import { tracked } from 'tracked-built-ins';
6
7 function extractRelationships(object) {
8     let relationships = {};
9     for (let relationshipName in object) {
10         relationships[relationshipName] =
11             object[relationshipName].links.related;
12     }
13     return relationships;
14 }
15
16 export default class CatalogService extends Service {
17     storage = {};
18
19     constructor() {
20         super(...arguments);
21         this.storage.bands = tracked([]);
22         this.storage.songs = tracked([]);
23     }
24
25     async fetchAll(type) {
26         if (type === 'bands') {
27             let response = await fetch('/bands');
28             let json = await response.json();
29             this.loadAll(json);
30             return this.bands;
31         }
32         if (type === 'songs') {
33             let response = await fetch('/songs');
34             let json = await response.json();
35             this.loadAll(json);
36             return this.songs;
37     }

```

```

37     }
38
39     loadAll(json) {
40         let records = [];
41         for (let item of json.data) {
42             records.push(this._loadResource(item));
43         }
44         return records;
45     }
46
+ 47     load(response) {
+ 48         return this._loadResource(response.data);
+ 49     }
+ 50
51     _loadResource(data) {
52         let record;
53         let { id, type, attributes, relationships } = data;
54         if (type === 'bands') {
55             let rels = extractRelationships(relationships);
56             record = new Band({ id, ...attributes }, rels);
57             this.add('band', record);
58         }
59         if (type === 'songs') {
60             let rels = extractRelationships(relationships);
61             record = new Song({ id, ...attributes }, rels);
62             this.add('song', record);
63         }
64         return record;
65     }
66
+ 67     async create(type, attributes, relationships = {}) {
+ 68         let payload = {
+ 69             data: {
+ 70                 type: type === 'band' ? 'bands' : 'songs',
+ 71                 attributes,
+ 72                 relationships,
+ 73             },
+ 74         };

```

```

+ 75     let response = await fetch(type === 'band' ? '/bands' : '/songs',
+ 76     {
+ 77       method: 'POST',
+ 78       headers: {
+ 79         'Content-Type': 'application/vnd.api+json',
+ 80       },
+ 81       body: JSON.stringify(payload),
+ 82     });
+ 83     let json = await response.json();
+ 84     return this.load(json);
+ 85   }

86   add(type, record) {
87     let collection = type === 'band' ? this.storage.bands :
this.storage.songs;
88     collection.push(record);
89   }

90   get bands() {
91     return this.storage.bands;
92   }

93   get songs() {
94     return this.storage.songs;
95   }

96   find(type, filterFn) {
97     let collection = type === 'band' ? this.bands : this.songs;
98     return collection.find(filterFn);
99   }
100 }
```

The method is ready to be used both for the creation of bands and that of songs. We also added a `load` method that complements the `loadAll` method and extracts and loads a single record to the catalog from the response. We also make `create` (and `load`) return the record that has been created.

We'll have to switch over the save operations to use the `create` method on the catalog:

```

1 // app/controllers/bands/new.js
2 import Controller from '@ember/controller';
3 import { action } from '@ember/object';
4 import { tracked } from '@glimmer/tracking';
- 5 import Band from 'rarwe/models/band';
6 import { inject as service } from '@ember/service';
- 7 import fetch from 'fetch';
8
9 export default class BandsNewController extends Controller {
10   @service catalog;
11   @service router;
12
13   @tracked name;
14
15   @action
16   updateName(event) {
17     this.name = event.target.value;
18   }
19
20   @action
21   async saveBand() {
- 22     let response = await fetch('/bands', {
- 23       method: 'POST',
- 24       headers: {
- 25         'Content-Type': 'application/vnd.api+json',
- 26       },
- 27       body: JSON.stringify({
- 28         data: {
- 29           type: 'bands',
- 30           attributes: {
- 31             name: this.name,
- 32           },
- 33         },
- 34       }),
- 35     });
- 36     let json = await response.json();
- 37     let { id, attributes } = json.data;

```

```
- 38     let record = new Band({ id, ...attributes });
- 39     this.catalog.add('band', record);
- 40     this.router.transitionTo('bands.band.songs', id);
+ 41     let band = await this.catalog.create('band', { name: this.name });
+ 42     this.router.transitionTo('bands.band.songs', band.id);
43   }
44 }
```

That piece of refactoring indeed fixed the bug of transitioning to the new band's songs page. Using `catalog.create` for creating a song is next:

```

1 // app/controllers/bands/band/songs.js
2 import Controller from '@ember/controller';
3 import { tracked } from '@glimmer/tracking';
4 import { action } from '@ember/object';
5 import Song from 'rarwe/models/song';
6 import { inject as service } from '@ember/service';
7 import fetch from 'fetch';
8
9 export default class BandsBandSongsController extends Controller {
10   @tracked showAddSong = true;
11   @tracked title = '';
12
13   @service catalog;
14
15   @action
16   async updateRating(song, rating) {
17     song.rating = rating;
18     let payload = {
19       data: {
20         id: song.id,
21         type: 'songs',
22         attributes: {
23           rating
24         }
25       }
26     };
27     await fetch(`/songs/${song.id}`, {
28       method: 'PATCH',
29       headers: {
30         'Content-Type': 'application/vnd.api+json'
31       },
32       body: JSON.stringify(payload)
33     });
34   }
35
36   @action
37   updateTitle(event) {

```

```

38     this.title = event.target.value;
39 }
40
41 @action
42 async saveSong() {
- 43     let payload = {
- 44         data: {
- 45             type: 'songs',
- 46             attributes: { title: this.title },
- 47             relationships: {
- 48                 band: {
- 49                     data: {
- 50                         id: this.model.id,
- 51                         type: 'bands'
- 52                     }
- 53                 }
- 54             }
- 55         }
- 56     };
- 57     let response = await fetch('/songs', {
- 58         method: 'POST',
- 59         headers: {
- 60             'Content-Type': 'application/vnd.api+json'
- 61         },
- 62         body: JSON.stringify(payload)
- 63     });
- 64
- 65     let json = await response.json();
- 66     let { id, attributes, relationships } = json.data;
- 67     let rels = {};
- 68     for (let relationshipName in relationships) {
- 69         rels[relationshipName] =
- 70             relationships[relationshipName].links.related;
- 71     let song = new Song({ id, ...attributes }, rels);
- 72     this.catalog.add('song', song);
- 73
+ 74     let song = await this.catalog.create(

```

```

+ 75     'song',
+ 76     { title: this.title },
+ 77     { band: { data: { id: this.model.id, type: 'bands' } } }
+ 78   );
79   this.model.songs = [...this.model.songs, song];
80   this.title = '';
81   this.showAddSong = true;
82 }
83
84 @action
85 cancel() {
86   this.title = '';
87   this.showAddSong = true;
88 }
89 }
```

Ah, there ain't nothing that warms our hearts as much as the sight of huge chunks of removed code.

- Fetching data from the back-end, and loading records extracted from the response into the catalog
- Creation of model instances, and saving them to the back-end
- Updating existing model instances
- Fetching relationships

Updating existing model instances

In similar fashion as we did for creating new records, we define an `update` method on the catalog service:

```

1 // app/services/catalog.js
2 import Service from '@ember/service';
3 import Band from 'rarwe/models/band';
4 import Song from 'rarwe/models/song';
5 import { tracked } from 'tracked-built-ins';
6
7 function extractRelationships(object) {
8     let relationships = {};
9     for (let relationshipName in object) {
10         relationships[relationshipName] =
11             object[relationshipName].links.related;
12     }
13     return relationships;
14 }
15
16 export default class CatalogService extends Service {
17     storage = {};
18
19     constructor() {
20         super(...arguments);
21         this.storage.bands = tracked([]);
22         this.storage.songs = tracked([]);
23     }
24
25     async fetchAll(type) {
26         if (type === 'bands') {
27             let response = await fetch('/bands');
28             let json = await response.json();
29             this.loadAll(json);
30             return this.bands;
31         }
32         if (type === 'songs') {
33             let response = await fetch('/songs');
34             let json = await response.json();
35             this.loadAll(json);
36             return this.songs;
37     }

```

```

37     }
38
39     loadAll(json) {
40         let records = [];
41         for (let item of json.data) {
42             records.push(this._loadResource(item));
43         }
44         return records;
45     }
46
47     load(response) {
48         return this._loadResource(response.data);
49     }
50
51     _loadResource(data) {
52         let record;
53         let { id, type, attributes, relationships } = data;
54         if (type === 'bands') {
55             let rels = extractRelationships(relationships);
56             record = new Band({ id, ...attributes }, rels);
57             this.add('band', record);
58         }
59         if (type === 'songs') {
60             let rels = extractRelationships(relationships);
61             record = new Song({ id, ...attributes }, rels);
62             this.add('song', record);
63         }
64         return record;
65     }
66
67     async create(type, attributes, relationships = {}) {
68         let payload = {
69             data: {
70                 type: type === 'band' ? 'bands' : 'songs',
71                 attributes,
72                 relationships,
73             },
74         };

```

```

75     let response = await fetch(type === 'band' ? '/bands' : '/songs', {
76       method: 'POST',
77       headers: {
78         'Content-Type': 'application/vnd.api+json',
79       },
80       body: JSON.stringify(payload),
81     });
82     let json = await response.json();
83     return this.load(json);
84   }
85
+ 86   async update(type, record, attributes) {
+ 87     let payload = {
+ 88       data: {
+ 89         id: record.id,
+ 90         type: type === 'band' ? 'bands' : 'songs',
+ 91         attributes,
+ 92       },
+ 93     };
+ 94     let url = type === 'band' ? `/bands/${record.id}` :
+ 95       `/songs/${record.id}`;
+ 96     await fetch(url, {
+ 97       method: 'PATCH',
+ 98       headers: {
+ 99         'Content-Type': 'application/vnd.api+json',
+100       },
+101       body: JSON.stringify(payload),
+102     });
+103   }
104   add(type, record) {
105     let collection = type === 'band' ? this.storage.bands :
this.storage.songs;
106     collection.push(record);
107   }
108
109   get bands() {
110     return this.storage.bands;

```

```
111     }
112
113     get songs() {
114         return this.storage.songs;
115     }
116
117     find(type, filterFn) {
118         let collection = type === 'band' ? this.bands : this.songs;
119         return collection.find(filterFn);
120     }
121 }
```

We currently only have one update operation, for changing song ratings:

```

1 // app/controllers/bands/band/songs.js
2 import Controller from '@ember/controller';
3 import { tracked } from '@glimmer/tracking';
4 import { action } from '@ember/object';
5 import { inject as service } from '@ember/service';
- 6 import fetch from 'fetch';
7
8 export default class BandsBandSongsController extends Controller {
9   @tracked showAddSong = true;
10  @tracked title = '';
11
12  @service catalog;
13
14  @action
15  async updateRating(song, rating) {
16    song.rating = rating;
- 17    let payload = {
- 18      data: {
- 19        id: song.id,
- 20        type: 'songs',
- 21        attributes: {
- 22          rating
- 23        }
- 24      }
- 25    };
- 26    await fetch(`/songs/${song.id}`, {
- 27      method: 'PATCH',
- 28      headers: {
- 29        'Content-Type': 'application/vnd.api+json'
- 30      },
- 31      body: JSON.stringify(payload)
- 32    });
+ 33    this.catalog.update('song', song, { rating });
34  }
35
36  @action
37  updateTitle(event) {

```

```

38     this.title = event.target.value;
39 }
40
41 @action
42 async saveSong() {
43     let song = await this.catalog.create(
44         'song',
45         { title: this.title },
46         { band: { data: { id: this.model.id, type: 'bands' } } }
47     );
48     this.model.songs = [...this.model.songs, song];
49     this.title = '';
50     this.showAddSong = true;
51 }
52
53 @action
54 cancel() {
55     this.title = '';
56     this.showAddSong = true;
57 }
58 }
```

That looks swell, it can be seen at a glance that our `updateRating` really does what it says on the tin.

- Fetching data from the back-end, and loading records extracted from the response into the catalog
- Creation of model instances, and saving them to the back-end
- Updating existing model instances
- Fetching relationships

Fetching relationships

We're making great strides: the only place outside the catalog where we do a manual `fetch` is in the `songs` route where we query the relationship URL previously stored for the band. We know the drill – since we make all network requests through the `catalog` service, we'll refactor the by adding a method onto the service and using that in the `songs` route.

How about `fetchRelated` for the method name?

```
1 // app/services/catalog.js
2 import Service from '@ember/service';
3 import Band from 'rarwe/models/band';
4 import Song from 'rarwe/models/song';
5 import { tracked } from 'tracked-built-ins';
+ 6 import { isArray } from '@ember/array';
7
8 function extractRelationships(object) {
9     let relationships = {};
10    for (let relationshipName in object) {
11        relationships[relationshipName] =
12            object[relationshipName].links.related;
13    }
14    return relationships;
15}
16
17 export default class CatalogService extends Service {
18     storage = {};
19
20     constructor() {
21         super(...arguments);
22         this.storage.bands = tracked([]);
23         this.storage.songs = tracked([]);
24     }
25
26     async fetchAll(type) {
27         if (type === 'bands') {
28             let response = await fetch('/bands');
29             let json = await response.json();
30             this.loadAll(json);
31             return this.bands;
32         }
33         if (type === 'songs') {
34             let response = await fetch('/songs');
35             let json = await response.json();
36             this.loadAll(json);
37             return this.songs;
38     }
39 }
```

```

37         }
38     }
39
40     loadAll(json) {
41         let records = [];
42         for (let item of json.data) {
43             records.push(this._loadResource(item));
44         }
45         return records;
46     }
47
48     load(response) {
49         return this._loadResource(response.data);
50     }
51
52     _loadResource(data) {
53         let record;
54         let { id, type, attributes, relationships } = data;
55         if (type === 'bands') {
56             let rels = extractRelationships(relationships);
57             record = new Band({ id, ...attributes }, rels);
58             this.add('band', record);
59         }
60         if (type === 'songs') {
61             let rels = extractRelationships(relationships);
62             record = new Song({ id, ...attributes }, rels);
63             this.add('song', record);
64         }
65         return record;
66     }
67
+ 68     async fetchRelated(record, relationship) {
+ 69         let url = record.relationships[relationship];
+ 70         let response = await fetch(url);
+ 71         let json = await response.json();
+ 72         if (isArray(json.data)) {
+ 73             record[relationship] = this.loadAll(json);
+ 74         } else {

```

```

+ 75         record[relationship] = this.load(json);
+ 76     }
+ 77     return record[relationship];
+ 78 }
+ 79
80     async create(type, attributes, relationships = {}) {
81         let payload = {
82             data: {
83                 type: type === 'band' ? 'bands' : 'songs',
84                 attributes,
85                 relationships,
86             },
87         };
88         let response = await fetch(type === 'band' ? '/bands' : '/songs', {
89             method: 'POST',
90             headers: {
91                 'Content-Type': 'application/vnd.api+json',
92             },
93             body: JSON.stringify(payload),
94         });
95         let json = await response.json();
96         return this.load(json);
97     }
98
99     async update(type, record, attributes) {
100         let payload = {
101             data: {
102                 id: record.id,
103                 type: type === 'band' ? 'bands' : 'songs',
104                 attributes,
105             },
106         };
107         let url = type === 'band' ? `/bands/${record.id}` :
`/songs/${record.id}`;
108         await fetch(url, {
109             method: 'PATCH',
110             headers: {
111                 'Content-Type': 'application/vnd.api+json',

```

```

112     } ,
113     body: JSON.stringify(payload) ,
114   ) ;
115 }
116
117 add(type, record) {
118   let collection = type === 'band' ? this.storage.bands :
119   this.storage.songs;
120   collection.push(record);
121 }
122
123 get bands() {
124   return this.storage.bands;
125 }
126
127 get songs() {
128   return this.storage.songs;
129 }
130
131 find(type, filterFn) {
132   let collection = type === 'band' ? this.bands : this.songs;
133   return collection.find(filterFn);
134 }

```

The method does two things. First, it requests the back-end for the related records and loads them to the storage of the catalog. Second, it assigns the created, related records to a "relationship" property, like the songs property of Band model instances.

We also prepared the method to be able to retrieve a 1:1 relationship – when the response only contains one item (and is thus not an array). We don't need it at the moment as we only need to use `fetchRelated` to fetch the songs related to a band. This happens in the `bands.band.songs` route, so let's use our freshly baked method there:

```

1 // app/routes/bands/band/songs.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
- 4 import Song from 'rarwe/models/song';
- 5 import fetch from 'fetch';
6
7 export default class BandsBandSongsRoute extends Route {
8   @service catalog;
9
10  async model() {
11    let band = this.modelFor('bands.band');
- 12    let url = band.relationships.songs;
- 13    let response = await fetch(url);
- 14    let json = await response.json();
- 15    let songs = [];
- 16    for (let item of json.data) {
- 17      let { id, attributes, relationships } = item;
- 18      let rels = {};
- 19      for (let relationshipName in relationships) {
- 20        rels[relationshipName] =
relationships[relationshipName].links.related;
- 21      }
- 22      let song = new Song({ id, ...attributes }, rels);
- 23      songs.push(song);
- 24      this.catalog.add('song', song);
- 25    }
- 26    band.songs = songs;
+ 27    await this.catalog.fetchRelated(band, 'songs');
- 28    return band;
- 29  }
30
31  resetController(controller) {
32    controller.title = '';
33    controller.showAddSong = true;
34  }
35}

```

Looks like we're done, the last one of our todo items is now completed:

- Fetching data from the back-end, and loading records extracted from the response into the catalog
- Creation of model instances, and saving them to the back-end
- Updating existing model instances
- Fetching relationships

Too much (more precisely, too many) of a good song

It gives us a nice, cozy feeling to see how our catalog service handles all back-end operations and serves as storage for the models, bands and songs. The pieces of our app that are "closer" to the UI don't have any low-level network plumbing code, and they are, for the most part, quite descriptive of what they accomplish.

If we strain our minds somewhat to see if anything is wrong, we might find something, though. We use the "typed" collections in the catalog to display bands and songs. However, we're somewhat lax about adding new records to these collections.

Think about it: every time we go to the songs page of a band, we'll trigger the `model` hook of the route, and call `fetchRelated`. It will then diligently fetch the related song records from the back-end and load them into the catalog. Does it care if those song records are already in the store? Not a bit. We don't see something is wrong because we display `band.songs` in the template, which gets set in the freshly returned array of songs. Nevertheless, it's still wrong to have an evergrowing number of (identical) song records in the catalog every time we switch to another songs page.

Can the same happen with bands? The `bands` route is the top-most route we can navigate to in the app currently. When we visit it the first time, all the bands are fetched from the back-end and loaded into the catalog in the model hook of the `bands` route. Given how Ember's routing works (as we saw in [the Nested routes chapter](#)), we can't re-trigger the model hook.

Again, the fact that we can't currently make the error evident doesn't mean we shouldn't fix it while we still have the whole context loaded into our head. To give you an example of a scenario where the error could be observed let's assume we had the following route structure:

```
1 Router.map(function() {  
2   this.route('bands', ...);  
3   this.route('musicians', ...);  
4 });
```

js

If we could navigate between `bands` and `musicians`, we'd load all the bands from the back-end every time we visited the `bands` route and display them as many times as the number of visits.

Adding an identity map

The key to fixing the problem is the realization that we want records to be unique within their collection. So there should only be one band with a particular id, and the same should hold for songs. That's good news because the fix then should only consist of making sure we're not adding a record with the same id multiple times.

Since we add new records to the storage in the `add` method, we can just add the check there:

```

1 // app/services/catalog.js
2 import Service from '@ember/service';
3 import Band from 'rarwe/models/band';
4 import Song from 'rarwe/models/song';
5 import { tracked } from 'tracked-built-ins';
6 import { isArray } from '@ember/array';
7
8 function extractRelationships(object) {
9     let relationships = {};
10    for (let relationshipName in object) {
11        relationships[relationshipName] =
12            object[relationshipName].links.related;
13    }
14    return relationships;
15}
16
17 export default class CatalogService extends Service {
18    storage = {};
19
20    constructor() {
21        super(...arguments);
22        this.storage.bands = tracked([]);
23        this.storage.songs = tracked([]);
24    }
25
26    async fetchAll(type) {
27        if (type === 'bands') {
28            let response = await fetch('/bands');
29            let json = await response.json();
30            this.loadAll(json);
31            return this.bands;
32        }
33        if (type === 'songs') {
34            let response = await fetch('/songs');
35            let json = await response.json();
36            this.loadAll(json);
37            return this.songs;
38        }
39    }
40
41    loadAll(json) {
42        this.bands = json.map(band => {
43            let bandObject = band;
44            bandObject.id = band._id;
45            bandObject.name = band.name;
46            bandObject.url = band.url;
47            bandObject.image = band.image;
48            bandObject.songs = band.songs.map(song => {
49                let songObject = song;
50                songObject.id = song._id;
51                songObject.title = song.title;
52                songObject.duration = song.duration;
53                songObject.band = bandObject;
54                return songObject;
55            });
56            return bandObject;
57        });
58        this.songs = json.map(song => {
59            let songObject = song;
60            songObject.id = song._id;
61            songObject.title = song.title;
62            songObject.duration = song.duration;
63            songObject.band = this.bands.find(b => b.id === song.band);
64            return songObject;
65        });
66    }
67}

```

```

37     }
38 }
39
40 loadAll(json) {
41     let records = [];
42     for (let item of json.data) {
43         records.push(this._loadResource(item));
44     }
45     return records;
46 }
47
48 load(response) {
49     return this._loadResource(response.data);
50 }
51
52 _loadResource(data) {
53     let record;
54     let { id, type, attributes, relationships } = data;
55     if (type === 'bands') {
56         let rels = extractRelationships(relationships);
57         record = new Band({ id, ...attributes }, rels);
58         this.add('band', record);
59     }
60     if (type === 'songs') {
61         let rels = extractRelationships(relationships);
62         record = new Song({ id, ...attributes }, rels);
63         this.add('song', record);
64     }
65     return record;
66 }
67
68 async fetchRelated(record, relationship) {
69     let url = record.relationships[relationship];
70     let response = await fetch(url);
71     let json = await response.json();
72     if (isArray(json.data)) {
73         record[relationship] = this.loadAll(json);
74     } else {

```

```

75         record[relationship] = this.load(json);
76     }
77     return record[relationship];
78 }
79
80 async create(type, attributes, relationships = {}) {
81     let payload = {
82         data: {
83             type: type === 'band' ? 'bands' : 'songs',
84             attributes,
85             relationships,
86         },
87     };
88     let response = await fetch(type === 'band' ? '/bands' : '/songs', {
89         method: 'POST',
90         headers: {
91             'Content-Type': 'application/vnd.api+json',
92         },
93         body: JSON.stringify(payload),
94     });
95     let json = await response.json();
96     return this.load(json);
97 }
98
99 async update(type, record, attributes) {
100     let payload = {
101         data: {
102             id: record.id,
103             type: type === 'band' ? 'bands' : 'songs',
104             attributes,
105         },
106     };
107     let url = type === 'band' ? `/bands/${record.id}` :
`/songs/${record.id}`;
108     await fetch(url, {
109         method: 'PATCH',
110         headers: {
111             'Content-Type': 'application/vnd.api+json',

```

```

112         },
113         body: JSON.stringify(payload),
114     });
115 }
116
117 add(type, record) {
118     let collection = type === 'band' ? this.storage.bands :
119     this.storage.songs;
120     collection.push(record);
121     let recordIds = collection.map((record) => record.id);
122     if (!recordIds.includes(record.id)) {
123         collection.push(record);
124     }
125
126     get bands() {
127         return this.storage.bands;
128     }
129
130     get songs() {
131         return this.storage.songs;
132     }
133
134     find(type, filterFn) {
135         let collection = type === 'band' ? this.bands : this.songs;
136         return collection.find(filterFn);
137     }
138 }
```

With this change, our collections have become what's usually referred to as "identity maps" – items in the collection are unique on their ids. Technically, our collections are arrays and not maps (which is less than ideal) but they implement the idea behind the identity maps just the same.

We have thus eliminated the potential bugs that would result from having multiple identical records in the catalog.

Are we there yet?

We have a nifty catalog that handles our data operations and keeps copies of the records on the back-end (a cache, if you will). That said, it's far from finished (is code ever finished, though?). There are loads of things we could add, or change, to improve it. Here is a list of things I thought of:

- Collections should be hashes (= POJOs), or Maps for quicker look-up
- Some refactoring would be valuable to have fewer conditional branches based on type. A lot of the methods can only deal with `bands` and `songs` and so are not prepared for other model types.
- `add` could return the record just added (or already found) in the catalog
- This is not strictly for the catalog but we could enhance our model classes with a `save` method that would call the current `update` method of the catalog. Instead of writing `catalog.update('song', song, attributes)` we could then write `song.save(attributes)`, greatly improving developer ergonomics.
- We could add `fetch` to complement `fetchAll`. We don't need to query a single record from the back-end in the current state of the app because by the time we enter the `bands.band` route we can be sure to have fetched all the band records from the back-end so we can just look up the one we need from the catalog. However, in most applications the need to fetch a single record is a very common one.
- Reverse relationships are not automatically updated. If we call `catalog.fetchRelated(band, 'songs')`, we know that each song's `band` relationship should be set to `band`, but this is not done.
- Similarly, when we save a song to the back-end where the `band` relationship is set, the `songs` relationship of the band could be updated automatically to include the song.

I encourage you, dear reader, to implement some of the above as an exercise.

What about Ember Data?

You might have heard about the standard data library for Ember, Ember Data. It is a mature project, almost as old as Ember.js itself and implements what we have in the catalog service and a whole lot more.

The reason I haven't started using it right off-the-bat is that I wanted to demonstrate that it's perfectly possible to build an app without Ember Data. It is a common misconception that you *need* to use Ember Data with Ember, in part because it comes installed with new Ember projects.

The more complex the data scenarios of your app are, the more sense it makes to use ED. Because it's an "in-house" project (not a 3rd party add-on), it's quite well maintained and developed, so you might decide to adopt it from the start. The (possibly only) drawback of this decision is its size: removing the package from the project reduced both the compressed and uncompressed vendor build by about 20%, the uncompressed size being reduced by 164kb. That can be substantial in some apps.

The Ember Data team is well aware of the all-or-nothing nature of the project, and have launched [Project Trim](#). Its goal is to decompose it to smaller packages that each implement a specific feature set (models, relationships, etc.) so that only the required parts need to be installed.

Since we don't currently use it, let's go ahead and remove the ember-data package:

```
$ npm uninstall ember-data
```

Related hits

- [Project Trim \(Ember Data\)](#)

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 10

Advanced routing

Tuning

It's not happenstance that this is the 3rd chapter about routing. Ember's routing mechanism is very powerful and, if you know its ins and outs, you can wield it as a weapon against different types of challenges that come your way.

In this chapter, we'll improve our app in a few different ways and use some routing concept each time to achieve it. But first, let's learn (more) about route hooks and actions.

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 route transition:

```
beforeModel(transition) { ... }
```

js

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 tree:

```
model(params, transition) {...}
```

js

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

```
afterModel(model, transition) {...}
```

js

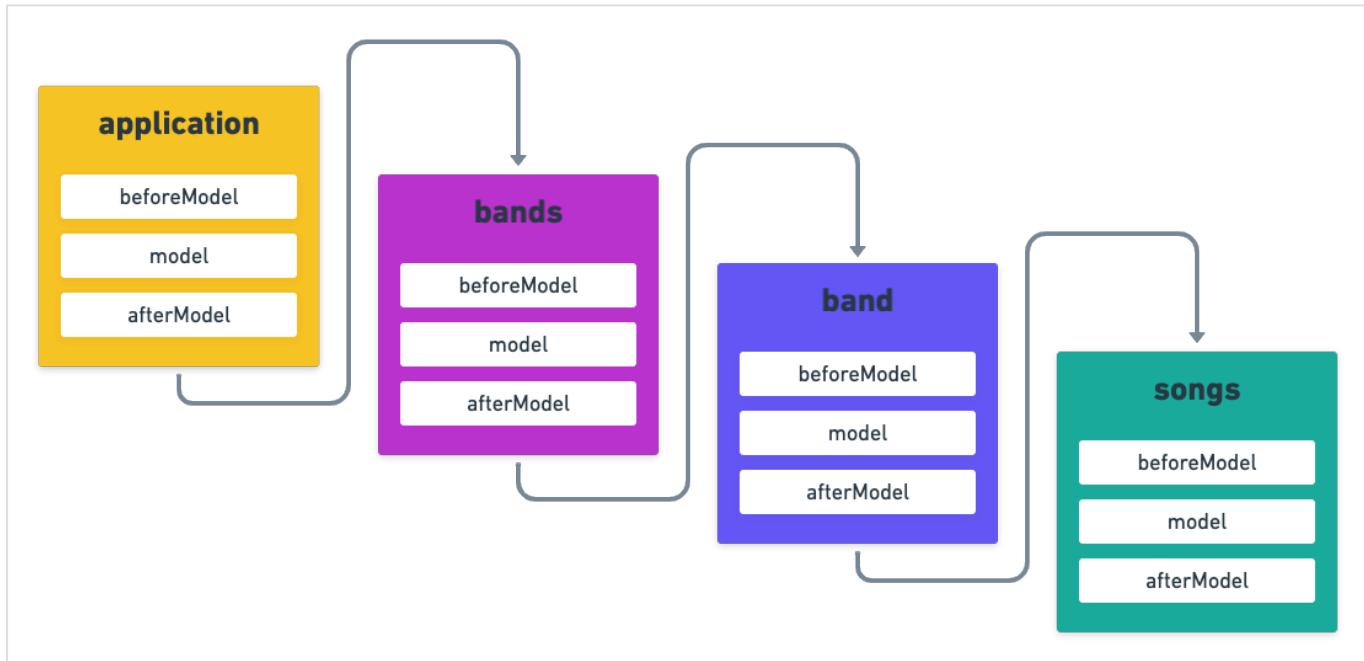
So at each level of a route transition, the `beforeModel`-model-`afterModel` hooks are called for the particular route (with one exception that we are going to see later). When these have all run for all levels of the transition, the transition is considered validated. It is only then that the next hook,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

```
setupController(controller, model, transition) {...}
```

js

We briefly saw (and used) the `setupController` hook in [the Controllers chapter](#). 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 need not to be provided.

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



Once these hooks have been 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`.

Routing events

Above the route hooks, there also exist "routing events" that are triggered at the beginning and the end of a transition on the router service.

They are called `routeWillChange` and `routeDidChange`.

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 events to improve our application.

Showing something on the index page

Remember, since we connect the app to a remote back-end, we have to pass the `proxy` option to start the Ember server:

```
1 $ ember s --proxy=http://json-api.rockandrollwithemberjs.com
2 Proxying to http://json-api.rockandrollwithemberjs.com
3
4 Build successful (20349ms) - Serving on http://localhost:4200/
```

When we visit the root of our application, `http://localhost:4200`, nothing is shown except the header:



Rock & Roll with Octane

We want to show the list of bands in this case, the page we see if we manually update the URL bar to `http://localhost:4200/bands`. In other words, we want to show the `bands` route as the `index` route.

There's a neat little trick that achieves this in a very simple way. As we said in [Nested routes](#), each of our route definitions in `router.js` takes an optional `path` parameter that tells Ember's router what URL path to generate for that route.

When the router loads the app, it matches each segment in the URL, so when we navigate to `http://localhost:4200/`, there's nothing to match. Another way to see this is that there's an empty path, `/`, to match – and that's what we can leverage. We can define `{ path: '/' }` for the `bands` route:

```

1 // app/router.js
2 import EmberRouter from '@ember/routing/router';
3 import config from 'rarwe/config/environment';
4
5 export default class Router extends EmberRouter {
6   location = config.locationType;
7   rootURL = config.rootURL;
8 }
9
10 Router.map(function () {
- 11   this.route('bands', function () {
- 12     this.route('band', { path: ':id' }, function () {
+ 13     this.route('bands', { path: '/' }, function () {
+ 14       this.route('band', { path: 'bands/:id' }, function () {
15         this.route('songs');
16       });
- 17     this.route('new');
+ 18     this.route('new', { path: 'bands/new' });
19   });
20 });

```

That makes our `bands` route be served on `/`, the app's index page.

You'll also notice we've amended the paths for the next level down, for the `bands.band` and `bands.new` routes. That's because we want those route to be served on `/bands/:id` and `bands/new`, respectively, instead of just `/:id` and `/new`. As we've just removed the generated `/bands` URL segment with the first change, we need to add it back. (For a refresher on how paths of nested routes map to URL segments, see [the Nested routes chapter](#)).

We can use the same "empty path" trick at any route level. Let's assume we have the following routes further down the tree:

```
1 Router.map(function() {  
2   this.route('bands', { path: '/' }, function() {  
3     this.route('band', { path: '/bands/:id' }, function() {  
4       this.route('albums');  
5       this.route('history', { path: '/' });  
6       this.route('members');  
7       this.route('songs');  
8     });  
9   });  
10});
```

js

With this (imagined) structure, if we visited `/bands/24/`, the app would enter the `bands.band.history` route since that one has the empty path defined.

Now let's also add the ability to go to the landing page from anywhere. Almost universally, the way to do this is to turn the site header into a link, so let's go with the crowd:

```

1  {{!!-- app/templates/application.hbs --}}
2  {{pageTitle "Rock & Roll with Octane"}}
3
4  <div class="bg-blue-900 text-gray-100 p-8 h-screen">
5    <div class="mx-auto mb-4">
6      <div class="h-12 flex items-center mb-4 border-b-2">
- 7        <a href="#">
+ 8        <LinkTo @route="bands">
9          <h1 class="font-bold text-2xl">
10         Rock & Roll <span class="font-normal">with Octane</span>
11       </h1>
- 12     </a>
+ 13   </LinkTo>
14   </div>
15   <div class="flex flex-wrap -mx-2">
16     {{outlet}}
17   </div>
18 </div>
19 </div>

```

HBS

Redirecting

Let's say we also want to show a description of each band on a new page that we'll call the band details page. As a first step, let's introduce tabbed navigation for each band. One tab will list the songs (what we already have) while the other one will be called "Details" and show a short description of the band.

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

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

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

```
1  {{!-- app/templates/bands/band.hbs --}}
```

```
+ 2  <nav
```

```
+ 3    class="flex mb-8 text-lg"
```

```
+ 4    role="navigation"
```

```
+ 5  >
```

```
+ 6    <div class="pb-1">
```

```
+ 7      <LinkTo @route="bands.band.details">
```

```
+ 8        Details
```

```
+ 9      </LinkTo>
```

```
+ 10     </div>
```

```
+ 11
```

```
+ 12     <div class="pb-1 ml-4">
```

```
+ 13      <LinkTo @route="bands.band.songs">
```

```
+ 14        Songs
```

```
+ 15      </LinkTo>
```

```
+ 16     </div>
```

```
+ 17   </nav>
```

```
18   {{outlet}}
```

HBS

Since we want to show some properties of the band on the Details tab, the model of the new `bands.band.details` route should be the band itself. Using the `modelFor` method introduced in the [Nested routes chapter](#), we can easily accomplish this:

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

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

```
3
```

```
4 export default class BandsBandDetailsRoute extends Route {
```

```
+ 5   model() {
```

```
+ 6     return this.modelFor('bands.band');
```

```
+ 7   }
```

```
8 }
```

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, child routes inherit the model of their parent by default. This means we could remove the above file altogether and the page would still work. We'll only remove the `model` hook for now since we'll need to add code to this route later in this chapter:

```
1 // app/routes/bands/band/details.js
2 import Route from '@ember/routing/route';
3
- 4 export default class BandsBandDetailsRoute extends Route {
- 5   model() {
- 6     return this.modelFor('bands.band');
- 7   }
- 8 }
+ 9 export default class BandsBandDetailsRoute extends Route {}
```

Next, 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:

```

1 // app/models/band.js
2 import { tracked } from '@glimmer/tracking';
3
4 export default class Band {
5   @tracked name;
6   @tracked songs;
7
8 - constructor({ id, name, songs }, relationships = {}) {
9 + constructor({ id, name, songs, description }, relationships = {}) {
10   this.id = id;
11   this.name = name;
12   this.songs = songs || [];
13   this.relationships = relationships;
14 +   this.description = description;
15 }
16 }
```

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`:

```

1  {{!!-- app/templates/bands.hbs --}}
2  {{pageTitle "Bands"}}
3
4  <div class="w-1/3 px-2">
5    <ul class="pl-2 pr-8">
6      {{#each @model as |band|}}
7        <li class="mb-2">
8          <LinkTo @route="bands.band.songs" @model={{band.id}}>
+ 9          <LinkTo @route="bands.band" @model={{band.id}}>
10            {{band.name}}
11          </LinkTo>
12        </li>
13      {{/each}}
14    </ul>
15    <LinkTo
16      @route="bands.new"
17      class="inline-block ml-2 mt-2 p-2 rounded bg-purple-600 shadow-md
18      hover:shadow-lg hover:text-white hover:bg-purple-500 focus:outline-
19      none">
20      Add band
21    </LinkTo>
22  </div>
23  {{outlet}}
24 </div>

```

HBS

Third, we need to make the redirection in the `bands.band` route based on whether the band has a description or not.

There are two ways to do this. In cases where the model of the route is not needed in order to decide on the redirection, we can use the `beforeModel` hook. One common use case is authentication: if the user is not properly authenticated to see the page, we can already redirect them at this point.

In case the model is needed for the redirection, the proper place to do the redirection is the aptly named `redirect` hook which gets passed the resolved model as the first parameter:

```

1 // app/routes/bands/band.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
4
5 export default class BandsBandRoute extends Route {
6   @service catalog;
+ 7   @service router;
8
9   model(params) {
10     return this.catalog.find('band', (band) => band.id === params.id);
11   }
+ 12
+ 13   redirect(band) {
+ 14     if (band.description) {
+ 15       this.router.transitionTo('bands.band.details');
+ 16     } else {
+ 17       this.router.transitionTo('bands.band.songs');
+ 18     }
+ 19   }
+ 20
21 }

```

Now, let's display the band's name and description on the band details page:

```

1 {{!-- app/templates/bands/band/details.hbs --}}
- 2 {{page-title "Details"}}
- 3 {{outlet}}
+ 4 {{page-title @model.name " details | Rock & Roll with Octane"}}
  replace=true}
+ 5
+ 6 <h2 class="mb-4 text-lg">{{@model.name}}</h2>
+ 7 <p>{{@model.description}}</p>

```

HBS

If we 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 Pearl Jam has a description:

The screenshot shows a dark-themed web application interface. At the top, the title "Rock & Roll with Octane" is displayed. Below it is a horizontal line. A list of band names is shown on the left, each with a "Details" and "Songs" link to its right. The "Pearl Jam" entry is highlighted with a purple background. To the right of the band list, there is a descriptive paragraph about Pearl Jam. At the bottom left is a purple "Add band" button.

Band Name	Details	Songs
Foo Fighters		
Kaya Project		
Led Zeppelin		
Pearl Jam		
Radiohead		
Red Hot Chili Peppers		

Add band

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

This screenshot shows the same application state as the previous one, but the "Pearl Jam" row now has a purple background, indicating it is selected. The descriptive paragraph to the right of the band list is no longer visible, replaced by a large empty white space.

That's really weird, how can this happen?

The Ember router can only transition to leaf routes. `bands.band` has three child routes, `bands.band.songs`, `bands.band.details` and the implicit `bands.band.index`. When we instruct Ember to go to a route which has subroutes (like `bands.band`), the actual transition will be made to its index route (`bands.band.index`).

That means that clicking the above `<LinkTo @route="bands.band" ...>` will actually attempt a

transition from `bands.band.details` to `bands.band.index`. Since this transition does not need to pass by the common parent route `bands.band` to complete this transition, the redirection code defined in that 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 g route bands/band/index
```

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

```
1 // app/routes/bands/band.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
4
5 export default class BandsBandRoute extends Route {
6   @service catalog;
7   @service router;
8
9   model(params) {
10     return this.catalog.find('band', (band) => band.id === params.id);
11   }
12
13   redirect(band) {
14     if (band.description) {
15       this.router.transitionTo('bands.band.details');
16     } else {
17       this.router.transitionTo('bands.band.songs');
18     }
19   }
20
21 }
```

```

1 // app/routes/bands/band/index.js
2 import Route from '@ember/routing/route';
+ 3 import { inject as service } from '@ember/service';
4
5 export default class BandsBandIndexRoute extends Route {
+ 6   @service router;
+ 7
+ 8   redirect(band) {
+ 9     if (band.description) {
+10       this.router.transitionTo('bands.band.details');
+11     } else {
+12       this.router.transitionTo('bands.band.songs');
+13     }
+14   }
15 }
```

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.

You could also argue that since the transition is to a leaf route, we should make that explicit in our template, by passing `@route="bands.band.index"`, like this: `<LinkTo @route="bands.band.index" @model={{band.id}} >`.

This would work, but it'd break the automatic setting of active `LinkTos` by Ember's routes.

What am I talking about? Ember automatically adds an `active` CSS class to all `LinkTos` which are currently active. More precisely, to all those whose `@route` points to a route which is the currently active route *or its ancestor*. If the current route is `bands.band.songs`, any `LinkTo` that points to `bands.band.songs`, `bands.band`, or `bands` will be marked active.

You now see what the problem would be if we pointed the band links to `bands.band.index`. Since `bands.band.index` is not an ancestor of `bands.band.songs` or `bands.band.details`, the band links wouldn't be marked as active, even though we'd probably want them to.

Which reminds us that we don't visually show the active links as such, so let's do that next.

Marking links as active

In many cases, the built-in method I described above is totally fine. The `<a>` tags rendered by `LinkTo` have an `active` class, and the tag is styled with CSS to highlight it as active. In other cases, however, this is not flexible enough. What if it's not the `a` tag that should have the `active` class? Or, if the project uses a utility-first approach and elements are not styled via custom class names?

In our app, both of these hold true so we should look for another solution.

We can rely on our trustworthy ally, the router service. It has an `isActive` method that tells us whether a certain route with the passed parameters is active or not. Because the band navigation markup is in the `bands.band` template, we could create the related controller and the "activeness" code there.

However, a cleaner solution is to extract a `BandNavigation` component to implement it in. This way, all the code will be nicely encapsulated in the component whose name also exactly describes its purpose.

Let's first generate the component and move the current band navigation markup there:

```
$ ember g component band-navigation
```

```
1 {{!!-- app/components/band-navigation.hbs --} }
- 2 {{yield}}
+ 3 <nav
+ 4   class="flex mb-8 text-lg"
+ 5   role="navigation"
+ 6 >
+ 7   <div class="pb-1">
+ 8     <LinkTo @route="bands.band.details">
+ 9       Details
+ 10      </LinkTo>
+ 11    </div>
+ 12
+ 13   <div class="pb-1 ml-4">
+ 14     <LinkTo @route="bands.band.songs">
+ 15       Songs
+ 16     </LinkTo>
+ 17   </div>
+ 18 </nav>
```

HBS

```

1  {{!-- app/templates/bands/band.hbs --}}
- 2 <nav
- 3   class="flex mb-8 text-lg"
- 4   role="navigation"
- 5 >
- 6   <div class="pb-1">
- 7     <LinkTo @route="bands.band.details">
- 8       Details
- 9     </LinkTo>
- 10    </div>
- 11
- 12    <div class="pb-1 ml-4">
- 13      <LinkTo @route="bands.band.songs">
- 14        Songs
- 15      </LinkTo>
- 16    </div>
- 17  </nav>
+ 18 <BandNavigation @band={{@model}} />
  19 {{outlet}}

```

HBS

Everything is the same as before but we now have the component to implement the activeness in. We'll have to write some JavaScript so we need to create the backing class, too:

```
$ ember g component-class band-navigation
```

We'll define a single `isActive` property that tells which link is active. `router.isActive` needs not only the route name but also all route models for that route (if there are any). For example, our `bands.band` route is only ever active for one specific band, so it makes sense that the band needs to be passed in to that method.

```
1 // app/components/band-navigation.js
2 import Component from '@glimmer/component';
+ 3 import { inject as service } from '@ember/service';
4
5 export default class BandNavigationComponent extends Component {
+ 6   @service router;
+ 7
+ 8   get isActive() {
+ 9     return {
+10       details: this.router.isActive('bands.band.details',
+11         this.args.band),
+12     };
+13   }
14 }
```

That means we can use `isActive.details` and `isActive.songs` in the component template:

```

1  {{!!-- app/components/band-navigation.hbs --} }
2  <nav
3      class="flex mb-8 text-lg"
4      role="navigation"
5  >
6  -   <div class="pb-1">
7 +   <div class="pb-1 {{if this.isActive.details "border-b-4 border-
purple-400"}}">
8      <LinkTo @route="bands.band.details">
9          Details
10         </LinkTo>
11     </div>
12
13 -   <div class="pb-1 ml-4">
14 +   <div class="ml-4 pb-1 {{if this.isActive.songs "border-b-4 border-
purple-400"}}">
15      <LinkTo @route="bands.band.songs">
16          Songs
17         </LinkTo>
18     </div>
19 </nav>

```

HBS

Look at how cool it is: the active link now has a nice purple border at the bottom:

The screenshot shows a dark blue header with the title "Rock & Roll with Octane". Below the header is a navigation bar with two items: "Details" and "Songs". The "Songs" item is underlined, indicating it is the active link. To the right of the navigation bar, there is a list of bands and their songs, each accompanied by a star rating. At the bottom left is a purple button labeled "Add band", and at the bottom right is another purple button labeled "Add song".

Band	Song	Rating
Foo Fighters		★★★★★
Kaya Project		★★★★★
Led Zeppelin	Black Dog	★★★★★
Pearl Jam	Achilles Last Stand	★★★★★
Radiohead	Immigrant Song	★★★★★
Red Hot Chili Peppers	Whole Lotta Love	★★★★★

Highlighting the active band link

While we're at it, we should also highlight the active band link in the sidebar. The process is somewhat similar so feel free to give it a go. I'll give my solution below so that you can check yours against it.

This time, we'll generate both the component template and class at the same time:

```
$ ember g component band-list --with-component-class
```

The idea is to augment each band item in the list with an `isActive` property:

```
1 // app/components/band-list.js
2 import Component from '@glimmer/component';
+ 3 import { inject as service } from '@ember/service';
4
5 export default class BandListComponent extends Component {
+ 6   @service router;
+ 7
+ 8   get bands() {
+ 9     return this.args.bands.map((band) => {
+10     return {
+11       band,
+12       isActive: this.router.isActive('bands.band', band),
+13     };
+14   });
+15 }
16 }
```

This way, each item will have access both to the band itself (via the `band` property) and whether it's the active route.

```
1 {{!!-- app/components/band-list.hbs --} }
- 2 {{yield}}
+ 3 <ul class="pl-2 pr-8">
+ 4   {{#each this.bands as |item|}}
+ 5     <li class="mb-2">
+ 6       <LinkTo
+ 7         class="{{if item.isActive "border-purple-400 border-l-4 pl-2" }}"
+ 8         @route="bands.band"
+ 9         @model="{{item.band.id}}"
+10       >
+11       {{item.band.name}}
+12     </LinkTo>
+13   </li>
+14 {{/each}}
+15 </ul>
```

HBS

Notice how we're iterating through `this.bands`, the property defined in the backing JavaScript class. It's a great example of how `@bands` and `this.bands` refer to two different things in component templates. `@bands` is an argument that was passed in, while `this.bands` is a property defined in the JavaScript class.

We now have to invoke the component from the bands template, passing the bands into it:

```

1  {{!!-- app/templates/bands.hbs --}}
2  {{pageTitle "Bands"}}
3
4  <div class="w-1/3 px-2">
- 5    <ul class="pl-2 pr-8">
- 6      {{#each @model as |band|}}
- 7        <li class="mb-2">
- 8          <LinkTo @route="bands.band" @model={{band.id}}>
- 9            {{band.name}}
- 10           </LinkTo>
- 11        </li>
- 12      {{/each}}
- 13    </ul>
+ 14    <BandList @bands={{@model}} />
15    <LinkTo
16      @route="bands.new"
17      class="inline-block ml-2 mt-2 p-2 rounded bg-purple-600 shadow-md
18      hover:shadow-lg hover:text-white hover:bg-purple-500 focus:outline-
19      none">
20      Add band
21    </LinkTo>
22  </div>
23  {{outlet}}
24 </div>

```

HBS

The active band is correctly highlighted in the sidebar:

The screenshot shows a dark-themed web application interface. At the top, there's a header with the title "Rock & Roll with Octane". Below the header is a list of band names: Foo Fighters, Kaya Project, Led Zeppelin, Pearl Jam, Radiohead, and Red Hot Chili Peppers. The "Pearl Jam" card is highlighted with a purple border and has a purple "Details" button above it. A purple "Songs" button is also present. Below the card, a purple box contains the text "Pearl Jam is an American rock band, formed in Seattle, Washington in 1990.". At the bottom left, there's a purple "Add band" button.

Skipping model resolution (don't)

Since we are talking about router subtleties in this chapter, let me present you with a rookie trap.

Make the following tiny change in the `band-list` component template:

```
1  {{!-- app/components/band-list.hbs --}}
2  <ul class="pl-2 pr-8">
3    {{#each this.bands as |item|}}
4      <li class="mb-2">
5        <LinkTo
6          class={{if item.isActive "border-purple-400 border-l-4 pl-2"}}
7          @route="bands.band"
8          @model={{item.band.id}}>
+ 9          @model={{item.band}}>
10         >
11           {{item.band.name}}
12         </LinkTo>
13       </li>
14     {{/each}}
15   </ul>
```

HBS

Instead of passing the `id` of each band to the `LinkTo`, we pass the band object itself.

Everything keeps working as before, yet there's something strange going on. If you put a breakpoint in the `model` hook of the `bands.band` route, you'll find that when you click the `LinkTo` we just changed (if you click any of the band links), the breakpoint is not hit! The model hook is simply not called.

At the same time, if you start the application on the `bands/pearl-jam` URL, execution will stop at the breakpoint, as expected.

The reason for this is the following:

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

This might trip up a lot of people, but in a way 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.

Having said that, this behavior can be very confusing and there is currently [an RFC](#) (Request For Comments, a proposal to change it) to always run the model hook, no matter what is passed in.

Another way to skip the model hook is from "application code.", using one of the transition methods: `router.transitionTo`, `router.replaceWith`, or the appropriate methods on routes and controllers.

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.

Better still, you simply shouldn't use a context object in links and manual transitions and just use the expected property of the object, which, in most cases is a unique identifier. Let's thus revert the small change we've made:

```

1  {{!-- app/components/band-list.hbs --}}
2  <ul class="pl-2 pr-8">
3    {{#each this.bands as |item|}}
4      <li class="mb-2">
5        <LinkTo
6          class={{if item.isActive "border-purple-400 border-l-4 pl-2"}}
7          @route="bands.band"
8          @model={{item.band}}>
9          @model={{item.band.id}}>
10         >
11           {{item.band.name}}
12         </LinkTo>
13       </li>
14     {{/each}}
15   </ul>

```

HBS

Other route hooks

There are a couple of other route hooks we haven't discussed.

`activate` gets invoked when a new route is entered but not when the route's model changes. In our app, going from the songs page of one band to another wouldn't trigger the hook, but going back to the bands page would.

`deactivate` is the exiting pair of `activate`. It's fired when a route is completely exited, and is not fired when only the route's model changes.

Leaving a route

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 subscribing to events of the router service.

There are two such events, `routeDidChange` and `routeWillChange`. We'll use `routeWillChange` to warn the user about losing unsaved changes.

If the user starts typing the name of a new band and then clicks another link, the app transitions away from the page. The next time the user returns to it, the input has been cleared. It would probably improve the user experience if we warned them about losing their changes.

Before a transition is made, a `routeWillChange` event is triggered on the router service. It receives the transition object which includes both the source and destination routes (among other useful pieces of data). We'll set up a listener upon the creation of the controller.

```

1 // app/controllers/bands/new.js
2 import Controller from '@ember/controller';
3 import { action } from '@ember/object';
4 import { tracked } from '@glimmer/tracking';
5 import { inject as service } from '@ember/service';
6
7 export default class BandsNewController extends Controller {
8   @service catalog;
9   @service router;
10
11   @tracked name;
12
+ 13   constructor() {
+ 14     super(...arguments);
+ 15     this.router.on('routeWillChange', (transition) => {
+ 16       if (transition.from.name === 'bands.new') {
+ 17         if (this.name) {
+ 18           let leave = window.confirm('You have unsaved changes. Are
you sure?');
+ 19           if (!leave) {
+ 20             transition.abort();
+ 21           }
+ 22         }
+ 23       }
+ 24     });
+ 25   }
+ 26
27   @action
28   updateName(event) {
29     this.name = event.target.value;
30   }
31
32   @action
33   async saveBand() {
34     let band = await this.catalog.create('band', { name: this.name });
35     this.router.transitionTo('bands.band.songs', band.id);
36   }

```

```
37 }
```

`transition.from` is the source route object (the route we're moving away from), while `transition.to` is the one we're hoping to reach. We only care about moving away from *this* route, hence the first `if`. The `transition.abort` method cancels the transition which is what we want when the user clicks the Cancel button on the dialog.

Let's try putting in a name in the input for creating a new band but instead of hitting the Save button, let's click a band link. The confirm box comes up but for some reason, if we click "Ok", it comes up again and it only gets the message (so to speak) after confirming once more. But why?

If you recall we have another piece of redirection logic in the `bands.band.index` route:

```
1 // app/routes/bands/band/index.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
4
5 export default class BandsBandIndexRoute extends Route {
6   @service router;
7
8   redirect(band) {
9     if (band.description) {
10       this.router.transitionTo('bands.band.details');
11     } else {
12       this.router.transitionTo('bands.band.songs');
13     }
14   }
15 }
```

`router.transitionTo` implicitly cancels the current transition (if there's one) and starts a new one. So in our case, the clicking of the band link initiates a transition from `bands.new` to `bands.band.index`, and then the `router.transitionTo` aborts that one and starts one from `bands.new` to `bands.band.songs` (or `bands.band.details`, depending on the band). That's why the confirm dialog is brought up twice.

Let's add some code to make sure confirming once is enough.

```

1 // app/controllers/bands/new.js
2 import Controller from '@ember/controller';
3 import { action } from '@ember/object';
4 import { tracked } from '@glimmer/tracking';
5 import { inject as service } from '@ember/service';
6
7 export default class BandsNewController extends Controller {
8   @service catalog;
9   @service router;
10
11   @tracked name;
12
13   constructor() {
14     super(...arguments);
15     this.router.on('routeWillChange', (transition) => {
+ 16       if (this.confirmedLeave) {
+ 17         return;
+ 18       }
19       if (transition.from.name === 'bands.new') {
20         if (this.name) {
21           let leave = window.confirm('You have unsaved changes. Are
you sure?');
- 22         if (!leave) {
+ 23         if (leave) {
+ 24           this.confirmedLeave = true;
+ 25         } else {
26           transition.abort();
27         }
28       }
29     });
30   }
31
32   @action
33   updateName(event) {
34     this.name = event.target.value;
35   }
36 }
```

```
37
38     @action
39     async saveBand() {
40         let band = await this.catalog.create('band', { name: this.name });
41         this.router.transitionTo('bands.band.songs', band.id);
42     }
43 }
```

If `this.confirmedLeave` is true, we just let the transition happen. However, we have to make sure that when the user revisits the `bands.new` page, the warning code will run again. If `this.confirmedLeave` is still true, it'll be short-circuited which is not what we want.

A good place to reset the property is the `resetController` hook we're now familiar with:

```
1 // app/routes/bands/new.js
2 import Route from '@ember/routing/route';
3
4 export default class BandsNewRoute extends Route {
5     resetController(controller) {
6         controller.name = '';
+ 7         controller.confirmedLeave = false;
8     }
9 }
```

We should also take care to allow clicking the Save button. As it stands now, we also get the confirmation dialog in that case, since it also triggers a router transition. Simply setting the `confirmedLeave` property will do:

```

1 // app/controllers/bands/new.js
2 import Controller from '@ember/controller';
3 import { action } from '@ember/object';
4 import { tracked } from '@glimmer/tracking';
5 import { inject as service } from '@ember/service';
6
7 export default class BandsNewController extends Controller {
8   @service catalog;
9   @service router;
10
11   @tracked name;
12
13   constructor() {
14     super(...arguments);
15     this.router.on('routeWillChange', (transition) => {
16       if (this.confirmedLeave) {
17         return;
18       }
19       if (transition.from.name === 'bands.new') {
20         if (this.name) {
21           let leave = window.confirm('You have unsaved changes. Are
you sure?');
22           if (leave) {
23             this.confirmedLeave = true;
24           } else {
25             transition.abort();
26           }
27         }
28       }
29     });
30   }
31
32   @action
33   updateName(event) {
34     this.name = event.target.value;
35   }
36

```

```
37     @action
38     async saveBand() {
39         let band = await this.catalog.create('band', { name: this.name });
+ 40         this.confirmedLeave = true;
41         this.router.transitionTo('bands.band.songs', band.id);
42     }
43 }
```

Now it's the Cancel button on the confirm dialog that gives us some grief. You can't seem to be able to send the dialog away ever by clicking Cancel, no matter how hard you try.

Turns out when `transition.abort()` is called, an aborted transition is created and that also triggers the `routeWillChange` event. Consequently, we have to check whether the transition is an aborted one, and then bail out early:

```

1 // app/controllers/bands/new.js
2 import Controller from '@ember/controller';
3 import { action } from '@ember/object';
4 import { tracked } from '@glimmer/tracking';
5 import { inject as service } from '@ember/service';
6
7 export default class BandsNewController extends Controller {
8   @service catalog;
9   @service router;
10
11   @tracked name;
12
13   constructor() {
14     super(...arguments);
15     this.router.on('routeWillChange', (transition) => {
+ 16       if (transition.isAborted) {
+ 17         return;
+ 18       }
19       if (this.confirmedLeave) {
20         return;
21       }
22       if (transition.from.name === 'bands.new') {
23         if (this.name) {
24           let leave = window.confirm('You have unsaved changes. Are
you sure?');
25           if (leave) {
26             this.confirmedLeave = true;
27           } else {
28             transition.abort();
29           }
30         }
31       }
32     });
33   }
34
35   @action
36   updateName(event) {

```

```

37     this.name = event.target.value;
38 }
39
40 @action
41 async saveBand() {
42     let band = await this.catalog.create('band', { name: this.name });
43     this.confirmedLeave = true;
44     this.router.transitionTo('bands.band.songs', band.id);
45 }
46 }
```

The `on` method creates a listener that can be torn down by calling `off` on the same object, with the same method. The advantage of having added our listener on the controller is that we don't have to bother tearing it down. The controller is only created once and lives as long as the app is not closed, so we're fine. However, if we added the listener on a component, we'd have to take care to properly unregister it.

Related hits

- [Ember Diagonal by Alex Speller](#)
- [Ember Route Hooks - A Complete Look by Alex DiLiberto](#)
- [The router service RFC](#)
- [Don't use afterModel for redirection](#)
- [Setting link activeness – the modern way](#)

Next song

Our app has come a long way, and it is starting to take shape. It would be a shame if something happened to it (have I successfully channeled my inner Don Corleone?) 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.

We'll also see how functional building blocks of our app can be tested with lower-level, integration and unit,

tests.

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 automate 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.

From Ember 3.0 on, these tests are also called application tests.

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.

Mocking makes tests lighter. Since we can mock out network 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 back-end 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 back-end applications, because database tables can be truncated or recreated relatively easily there. However, in our front-end app, we would have to make the back-end 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.

Integration tests

Integration tests are also called rendering tests in Ember.js and are a good fit for component and helper tests as these tests verify what they render when invoked with different parameters and how they react to events.

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 ones), 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.

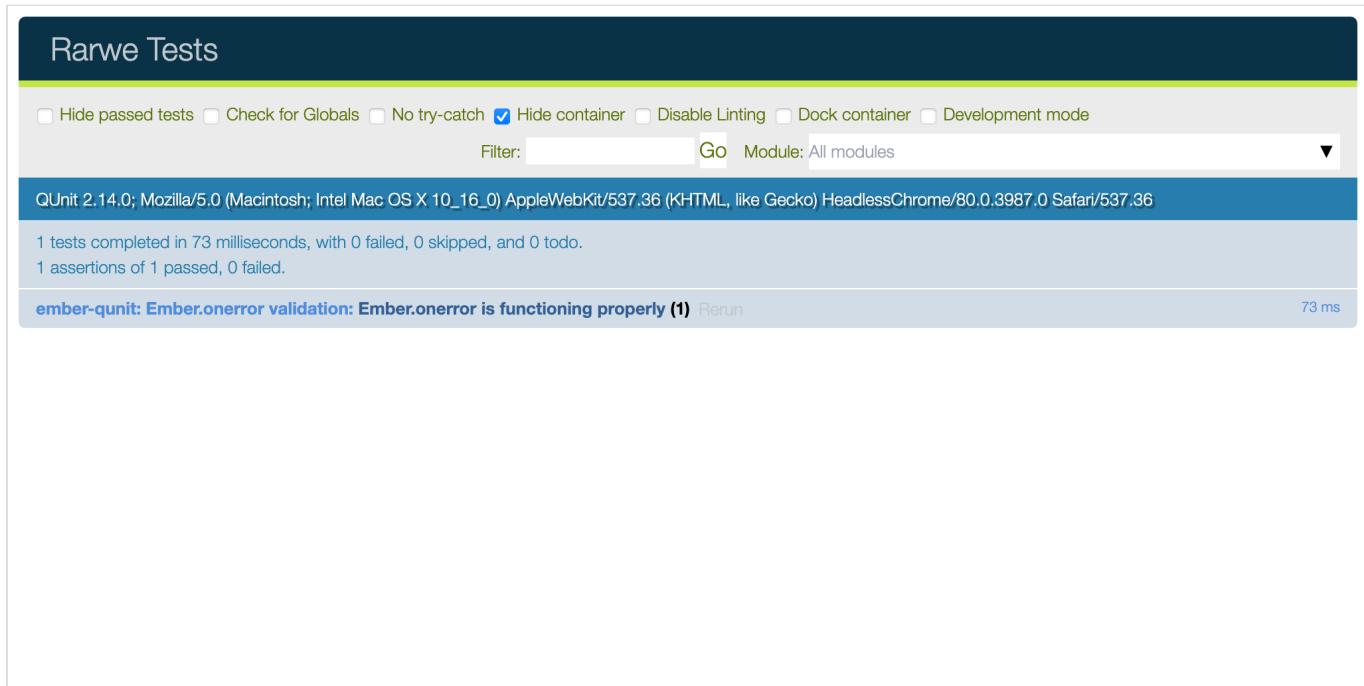
Writing acceptance tests

With the theory nailed down, it's time to roll up our sleeves and get to work.

Tests can be run by issuing the `ember test` command (or `ember t`, for short) from the command line. This runs the whole test suite – all of the tests. If the `--server` switch is added, the tests are run in interactive mode, which opens a browser with the tests and reruns them each time we change the code. We thus get the same developer experience of a tight feedback cycle as when we work with the application in development mode.

So let's run the `ember t --server` command now and see what it gives.

We have a single, auto-generated test that verifies that `Ember.onerror` works properly.



The screenshot shows the Rarwe Tests interface. At the top, there is a toolbar with several checkboxes: "Hide passed tests" (unchecked), "Check for Globals" (unchecked), "No try-catch" (unchecked), "Hide container" (checked), "Disable Linting" (unchecked), "Dock container" (unchecked), and "Development mode" (unchecked). Below the toolbar are buttons for "Filter:" (with an input field), "Go" (in green), and "Module:" (set to "All modules"). A dropdown menu is open on the right side of the toolbar. The main area has a dark header bar with the text "Rarwe Tests". Below the header, a blue bar displays the environment: "QUnit 2.14.0; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_16_0) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/80.0.3987.0 Safari/537.36". The main content area shows a message: "1 tests completed in 73 milliseconds, with 0 failed, 0 skipped, and 0 todo." followed by "1 assertions of 1 passed, 0 failed.". At the bottom of this section, there is a link "ember-qunit: Ember.onerror validation: Ember.onerror is functioning properly (1) Rerun" and a timestamp "73 ms".

By default all tests are run, but any single test module can be selected with the multi-select "Module" dropdown on the right. If you want to drill down further, you can click on the "Rerun" link next to each test case.

Let's break the build by adding our first acceptance test.

```
1 $ ember g acceptance-test bands
2 installing acceptance-test
3     create tests/acceptance/bands-test.js
```

This generates a file at `tests/acceptance/bands-test.js`. If we take a look inside, here is what we find:

```
1 // tests/acceptance/bands-test.js
2 import { module, test } from 'qunit';
3 import { visit, currentURL } from '@ember/test-helpers';
4 import { setupApplicationTest } from 'ember-qunit';
5
6 module('Acceptance | bands', function(hooks) {
7   setupApplicationTest(hooks);
8
9   test('visiting /bands', async function(assert) {
10     await visit('/bands');
11
12     assert.equal(currentURL(), '/bands');
13   });
14 }) ;
```

Wow, there are a load of new things here, so let's look at them one by one.

Ember CLI uses the [QUnit](#) testing framework by default, which explains the first import line. (If you prefer Mocha & Chai, it's easy to switch by installing the `ember-mocha` add-on).

The next line imports the needed test helpers from the official ember package. We'll use more of these as we expand our tests.

Next, `setupApplicationTest` is where setting up the application for acceptance tests is delegated to. Happily, it "just works," but if you're interested, you can take a peek by checking out the source of the [ember-test-helpers](#) add-on. This setup also ensures that each test receives a fresh application instance so that the test cases don't influence each other and all tests run with a blank slate.

The first (and so far, only) test block has the name of the test as first argument and an async function as the second argument inside which the test steps and assertions are declared.

The generated test verifies that when we go to /bands, we'll be on /bands (that's what `currentURL()` asserts). However, our app doesn't serve the /bands path, as evidenced by the failing test:

The screenshot shows the Rarwe Tests interface. At the top, there are several filter options: Hide passed tests (unchecked), Check for Globals (unchecked), No try-catch (unchecked), Hide container (checked), Disable Linting (checked), Dock container (unchecked), and Development mode (unchecked). Below these are fields for Filter, Go, and Module: All modules. A status bar at the bottom indicates: QUnit 2.14.0; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_16_0) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/80.0.3987.0 Safari/537.36. It also says 2 tests completed in 169 milliseconds, with 1 failed, 0 skipped, and 0 todo. 1 assertion of 2 passed, 1 failed. The main area shows a red-highlighted test result for "Acceptance | bands: visiting /bands (1, 0, 1) Rerun". The error message is "Promise rejected during "visiting /bands": /bands" and the source code leading to the error is shown:

```
Promise rejected during "visiting /bands": /bands
Source: UnrecognizedURLError: /bands
    at URLTransitionIntent.applyToState (http://localhost:4200/assets/vendor.js)
    at PrivateRouter.getTransitionByIntent (http://localhost:4200/assets/vendor.js)
    at PrivateRouter.transitionByIntent (http://localhost:4200/assets/vendor.js)
    at PrivateRouter.doTransition (http://localhost:4200/assets/vendor.js)
    at PrivateRouter.handleURL (http://localhost:4200/assets/vendor.js)
    at Router._doURLTransition (http://localhost:4200/assets/vendor.js)
    at Router.handleURL (http://localhost:4200/assets/vendor.js)
    at Class.visit (http://localhost:4200/assets/vendor.js)
    at http://localhost:4200/assets/test-support.js
    at async Object.<anonymous> (http://localhost:4200/assets/tests.js)
```

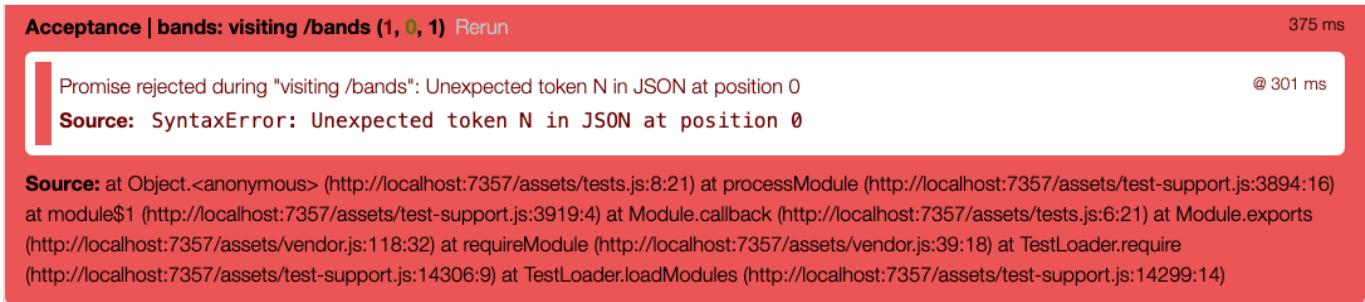
Let's fix that by going to the bands page of our app, /, instead:

```

1 // tests/acceptance/bands-test.js
2 import { module, test } from 'qunit';
3 import { visit, currentURL } from '@ember/test-helpers';
4 import { setupApplicationTest } from 'ember-qunit';
5
6 module('Acceptance | bands', function(hooks) {
7   setupApplicationTest(hooks);
8
9   test('visiting /bands', async function(assert) {
- 10     await visit('/bands');
+ 11     await visit('/');
12
- 13     assert.equal(currentURL(), '/bands');
+ 14     assert.equal(currentURL(), '/');
15   });
16 })

```

The app sends a request to the back-end to /bands to fetch data from. However, we didn't define a back-end to contact (like we did by passing the --proxy option in [the Advanced Routing chapter](#) so that fails:



Since we decided to write acceptance tests with stubbed out network requests, let's resolve the above error by stubbing out this request. ember-cli-mirage, the outstanding add-on we'll use, also allows us to create test fixtures and will serve as a one-stop shop to our testing needs.

Making the first test pass

As with all add-ons, the first step is to use `ember install` to add them to the project:

```
$ ember install ember-cli-mirage
```

This creates an empty config file under `mirage/config.js` to which we need to add the route handlers. These route handlers should replicate the behavior of the corresponding API endpoint.

Let's delete the comments in the file and add a couple of handlers for band-related requests:

```
1 // mirage/config.js
2 export default function () {
3   this.get('/bands');
4   this.get('/bands/:id');
5 }
```

These two lines will mock out the network requests the app makes against the back-end. They are called "shorthand handlers" because you don't have to implement what response should Mirage return for them – it is smart enough to figure that out.

Since we use Mirage for this, we'll need to set up Mirage in our acceptance test:

```

1 // tests/acceptance/bands-test.js
2 import { module, test } from 'qunit';
3 import { visit, currentURL } from '@ember/test-helpers';
4 import { setupApplicationTest } from 'ember-qunit';
+ 5 import { setupMirage } from 'ember-cli-mirage/test-support';
6
7 module('Acceptance | bands', function(hooks) {
8   setupApplicationTest(hooks);
+ 9   setupMirage(hooks);
10
11   test('visiting /bands', async function(assert) {
12     await visit('/');
13
14     assert.equal(currentURL(), '/');
15   });
16 });

```

The only `assert` line verifies that if the app transitions to / then the URL becomes /. Acceptance tests should be written from the end user's point of view: they don't care about URLs that much (unless they also happen to be web developers), but they do want to see the bands on the page. So let's tweak the test as follows:

```

1 // tests/acceptance/bands-test.js
2 import { module, test } from 'qunit';
3 import { visit, currentURL } from '@ember/test-helpers';
4 + import { visit } from '@ember/test-helpers';
5 import { setupApplicationTest } from 'ember-qunit';
6 import { setupMirage } from 'ember-cli-mirage/test-support';
7 + import { getPageTitle } from 'ember-page-title/test-support';
8
9 - module('Acceptance | bands', function(hooks) {
10 + module('Acceptance | Bands', function (hooks) {
11     setupApplicationTest(hooks);
12     setupMirage(hooks);
13
14     test('visiting /bands', async function(assert) {
15         test('List bands', async function (assert) {
16             this.server.create('band', { name: 'Radiohead' });
17             this.server.create('band', { name: 'Long Distance Calling' });
18
19             await visit('/');
20             assert.equal(getPageTitle(), 'Bands | Rock & Roll with Octane');
21
22             assert.equal(currentURL(), '/');
23             let bandLinks = document.querySelectorAll('.mb-2 > a');
24             assert.equal(bandLinks.length, 2, 'All band links are rendered');
25             assert.ok(
26                 bandLinks[0].textContent.includes('Radiohead'),
27                 'First band link contains the band name'
28             );
29             assert.ok(
30                 bandLinks[1].textContent.includes('Long Distance Calling'),
31                 'The other band link contains the band name'
32             );
33         });
34     });

```

`this.server` is how we access Mirage. We tell it to create two bands, passing an object with the properties to create the object with.

The new assertions now verify the page from the user's perspective: are there two band links, the first of which is Radiohead, the second Long Distance Calling?

All `assert` calls take as last parameter an optional message that will be printed for the assertion when it passes.

However, before our test can get to the point where it makes the request it fails with the following error:

Error while processing route: bands.index json.data is not iterable
TypeError: json.data is not iterable

Looking at the console, we find the actual reason is that Mirage fails with a 500 error when serving `/bands`:

Mirage: Error: You called `server.create('band')` but no model or factory was found. Make sure you're passing in the singularized version of the model or factory name.

Mirage models

Mirage needs to know what types of mock data we want to have, so we have to tell it by creating mirage models. Unsurprisingly, there's a blueprint for that:

```
1 $ ember g mirage-model band
2 installing mirage-model
3     create /mirage/models/band.js
```

When we look at the generated file, we see a strange syntax to define the class (it's only strange if you haven't worked on Ember apps in the pre-Octane times). It uses `Model.extend({ ... })` instead of the ES6 class syntax. It's useful to know that Ember has had its own class system which originates in the times when there wasn't widespread browser support for ES6 classes (or transpilers, for that matter). Being aware of this helps with reading code of not fully Octane Ember apps, or add-ons. This older syntax is mostly referred to as "classic" Ember class syntax.

```
1 // mirage/models/band.js
- 2 import { Model } from 'ember-cli-mirage';
+ 3 import { Model,hasMany } from 'ember-cli-mirage';
4
5 export default Model.extend({
+ 6   songs:hasMany(),
7 });
```

Simple properties don't need to be defined, but defining relationships gives us a few methods on the relationship property we can use to set up our data hierarchy.

While we're at it, let's create a Mirage model for our Song class, too:

```
1 $ ember g mirage-model song
2 installing mirage-model
3 create /mirage/models/song.js
```

And add the `band` relationship to it:

```
1 // mirage/models/song.js
- 2 import { Model } from 'ember-cli-mirage';
+ 3 import { Model, belongsTo } from 'ember-cli-mirage';
4
5 export default Model.extend({
+ 6   band:belongsTo(),
7 });
```

We don't strictly need the `songs` property though, as our test now passes with flying colors:

Rarwe Tests

Hide passed tests Check for Globals No try-catch Hide container Disable Linting Dock container Development mode Mirage logging

Filter: Go Module: All modules ▾

QUnit 2.14.0; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_16_0) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/80.0.3987.0 Safari/537.36

2 tests completed in 178 milliseconds, with 0 failed, 0 skipped, and 0 todo.
5 assertions of 5 passed, 0 failed.

Test Case	Time (ms)
Acceptance Bands: List bands (4) Rerun	82 ms
ember-qunit: Ember.onerror validation: Ember.onerror is functioning properly (1) Rerun	96 ms

(If the project has Ember Data models, Mirage auto-detects them and manually creating Mirage models is not necessary.)

Test creating a band

We now have the main elements for writing acceptance tests in place, so let's add a couple more scenarios in the same module to prevent features from breaking in the future.

The first should test creating a band:

```

1 // tests/acceptance/bands-test.js
2 import { module, test } from 'qunit';
- 3 import { visit } from '@ember/test-helpers';
+ 4 import { visit, click, fillIn } from '@ember/test-helpers';
5 import { setupApplicationTest } from 'ember-qunit';
6 import { setupMirage } from 'ember-cli-mirage/test-support';
7 import { getPageTitle } from 'ember-page-title/test-support';
8
9 module('Acceptance | Bands', function(hooks) {
10   setupApplicationTest(hooks);
11   setupMirage(hooks);
12
13   test('List bands', async function(assert) {
14     this.server.create('band', { name: 'Radiohead' });
15     this.server.create('band', { name: 'Long Distance Calling' });
16
17     await visit('/');
18     assert.equal(getPageTitle(), 'Bands | Rock & Roll with Octane');
19
20     let bandLinks = document.querySelectorAll('.mb-2 > a');
21     assert.equal(bandLinks.length, 2, 'All band links are rendered');
22     assert.ok(
23       bandLinks[0].textContent.includes('Radiohead'),
24       'First band link contains the band name'
25     );
26     assert.ok(
27       bandLinks[1].textContent.includes('Long Distance Calling'),
28       'The other band link contains the band name'
29     );
30   });
31
+ 32   test('Create a band', async function(assert) {
+ 33     this.server.create('band', { name: 'Royal Blood' });
+ 34
+ 35     await visit('/');
+ 36     await click('a[href="/bands/new"]');
+ 37     await fillIn('input', 'Caspian');

```

```

+ 38     await click('button');
+
+ 40     let bandLinks = document.querySelectorAll('.mb-2 > a');
+ 41     assert.equal(
+ 42         bandLinks.length,
+ 43         2,
+ 44         'All band links are rendered',
+ 45         'A new band link is rendered'
+ 46     );
+ 47     assert.ok(
+ 48         bandLinks[1].textContent.includes('Caspian'),
+ 49         'The new band link is rendered as the last item'
+ 50     );
+ 51     assert.ok(
+ 52         document
+ 53             .querySelector('.border-b-4.border-purple-400')
+ 54             .textContent.includes('Songs'),
+ 55             'The Songs tab is active'
+ 56     );
+ 57 });
58 });

```

We'll have to overcome all kinds of problems to make this new test pass – luckily none of them is existential.

First off, the Mirage server (the mock back-end) needs to handle the `POST /bands` request as it doesn't currently. We can do that with another magic shorthand handler:

```

1 // mirage/config.js
2 export default function () {
3     this.get('/bands');
4     this.get('/bands/:id');
+
5     this.post('/bands');
6 }
7 }
```

That doesn't get us far, though. The next error message on the console is pretty scary:

Mirage: Error: Your app tried to GET 'undefined', but there was no route defined to handle this request.

What rogue request would want to GET `undefined`?

To get more insight into what's going on, we can add `this.server.logging = true` to the test case, or just check the `Mirage` logging checkbox in newer versions. That makes Mirage print all requests and responses to the console – it's the equivalent's of Chrome's Network tab. Doing this we realize that the band items in the response to `GET /bands` don't contain the `relationships` property that we then use to fetch the related songs.

Mirage serializers

Serializers in Mirage are responsible for transforming the models which are returned from the route handlers – so they are partly responsible for us being able to use shorthand handlers. In this case, the serializer doesn't return the `relationships`, so we have to tweak it.

```
1 // mirage/serializers/application.js
2 import { JSONAPISerializer } from 'ember-cli-mirage';
3
4 export default JSONAPISerializer.extend({
+ 5   links(resource) {
+ 6     let { id, modelName } = resource;
+ 7     if (modelName === 'band') {
+ 8       return {
+ 9         songs: {
+10           related: `/bands/${id}/songs`,
+11           self: `/bands/${id}/relationships/songs`,
+12         },
+13       };
+14     }
+15   },
16 });


```

That change makes the Mirage serializer return a response very similar to what our real back-end does. Each

band item in the response will have a `relationships.songs.links.related` key with the URL to fetch the songs from as its value.

The next error is thrown when the app tries to fetch the songs:

Mirage: Error: Your app tried to GET '/bands/2/songs', but there was no route defined to handle this request.

With this request, we have to hold Mirage's hand a little bit. It can't figure out what we want to return for that URL, so we have to implement it ourselves. In other words, we can't use a shorthand handler, we'll have to use a function. Function handlers receive the schema as the first argument and the request as the second one:

```
1 // mirage/config.js
2 export default function () {
3     this.get('/bands');
4     this.get('/bands/:id');
+ 5     this.get('/bands/:id/songs', function (schema, request) {
+ 6         let id = request.params.id;
+ 7         return schema.songs.where({ bandId: id });
+ 8     });
9
10    this.post('/bands');
11 }
```

For each model that we define, a corresponding collection can be accessed on the schema. The bands collection is found on `schema.bands`, songs are on `schema.songs`. With the above, we return all songs whose `bandId` matches the `id` in the request.

We're making huge strides but our test still fails:

Rarwe Tests

Hide passed tests Check for Globals No try-catch Hide container Disable Linting Dock container Development mode Mirage logging

Filter: Go Module: All modules ▾

QUnit 2.14.0; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_16_0) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/80.0.3987.0 Safari/537.36

3 tests completed in 262 milliseconds, with 1 failed, 0 skipped, and 0 todo.
5 assertions of 7 passed, 2 failed.

Acceptance | Bands: List bands (4) Rerun 82 ms

Acceptance | Bands: Create a band (2, 0, 2) Rerun 96 ms

All band links are rendered @ 51 ms

Expected: 2
Result: 1
Diff: -1
Source: at Object.<anonymous> (<http://localhost:4200/assets/tests.js>)

Promise rejected during "Create a band": Cannot read property 'textContent' of undefined @ 55 ms

Source: TypeError: Cannot read property 'textContent' of undefined
at Object.<anonymous> (<http://localhost:4200/assets/tests.js>)

Source: at Object.<anonymous> (<http://localhost:4200/assets/tests.js:23:21>)

It turns out that by the time the test arrives at checking if there are two band links, the new band link hasn't rendered yet which explains the test failure. To better understand why that is, let's take a quick detour.

Settledness in tests

We have used a few of the test helpers, like `visit`, `click`, `fillIn`. There are way more "DOM" helpers we haven't used that but what they have in common is that they all wait for all async operations to finish before they resolve. That's why we have to put `await` in front of them, so that we can make sure all effects of that action (like clicking a button) have been fully reflected in the DOM before we move on.

So what can those async operations that the DOM helpers wait for be? Examples include route transitions, ajax requests, timers, and a few more that I encourage you check out in [the source of the settled helper](#).

Most of the time, we get away with using the DOM test helpers (with `await`) without worrying too much about async actions they trigger in the background. However, in some cases, we need to manually tell the test to wait until a certain condition fulfills. That's why Ember provides lower-level, non-DOM test helpers, in the same package. The one that catches our attention is called `waitFor` – it will wait for a DOM element to be rendered on screen. It also has a timeout so that it doesn't hang the test.

One element we can target that way is the text which is displayed when a band has no songs yet. That way, we know the empty songs page has been rendered for the new band and we expect the band to appear in the band list by that time:

```

1 // tests/acceptance/bands-test.js
2 import { module, test } from 'qunit';
3 import { visit, click, fillIn } from '@ember/test-helpers';
+ 4 import { visit, click, fillIn, waitFor } from '@ember/test-helpers';
5 import { setupApplicationTest } from 'ember-qunit';
6 import { setupMirage } from 'ember-cli-mirage/test-support';
7 import { getPageTitle } from 'ember-page-title/test-support';
8
9 module('Acceptance | Bands', function(hooks) {
10   setupApplicationTest(hooks);
11   setupMirage(hooks);
12
13   test('List bands', async function(assert) {
14     this.server.create('band', { name: 'Radiohead' });
15     this.server.create('band', { name: 'Long Distance Calling' });
16
17     await visit('/');
18     assert.equal(getPageTitle(), 'Bands | Rock & Roll with Octane');
19
20     let bandLinks = document.querySelectorAll('.mb-2 > a');
21     assert.equal(bandLinks.length, 2, 'All band links are rendered');
22     assert.ok(
23       bandLinks[0].textContent.includes('Radiohead'),
24       'First band link contains the band name'
25     );
26     assert.ok(
27       bandLinks[1].textContent.includes('Long Distance Calling'),
28       'The other band link contains the band name'
29     );
30   });
31
32   test('Create a band', async function(assert) {
33     this.server.create('band', { name: 'Royal Blood' });
34
35     await visit('/');
36     await click('a[href="/bands/new"]');
37     await fillIn('input', 'Caspian');

```

```

38     await click('button');
+ 39     await waitFor('p.text-center');

40
41     let bandLinks = document.querySelectorAll('.mb-2 > a');
42     assert.equal(
43         bandLinks.length,
44         2,
45         'All band links are rendered',
46         'A new band link is rendered'
47     );
48     assert.ok(
49         bandLinks[1].textContent.includes('Caspian'),
50         'The new band link is rendered as the last item'
51     );
52     assert.ok(
53         document
54             .querySelector('.border-b-4.border-purple-400')
55             .textContent.includes('Songs'),
56         'The Songs tab is active'
57     );
58 });
59 });

```

Rarwe Tests

Hide passed tests Check for Globals No try-catch Hide container Disable Linting Dock container Development mode Mirage logging

Filter: Go Module: All modules ▾

QUnit 2.14.0; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_16_0) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/80.0.3987.0 Safari/537.36

3 tests completed in 272 milliseconds, with 0 failed, 0 skipped, and 0 todo.
8 assertions of 8 passed, 0 failed.

Acceptance Bands: List bands (4)	Rerun	82 ms
Acceptance Bands: Create a band (3)	Rerun	106 ms
ember-qunit: Ember.onerror validation: Ember.onerror is functioning properly (1)	Rerun	84 ms

Improving tests

We have now written a couple of tests that guard against breaking existing functionality. They are, however, in the "just works" phase and there are various things we can do to make them more resilient – and more pleasant to write.

Using data attributes for designating elements in testing

If we take a look at our second test, the elements we trigger events on and assert against are specified either by tag name (`button`, `input`) or the CSS class (`p.text-center`). The problem with this is two-fold: they are not precise enough (there can be multiple buttons or inputs) and, since they are not there for testing purposes, they might change, causing false negative test failures.

Using data attributes specifically designated for identifying elements for our tests is a great way to solve this problem. Let's assign some to the elements we need and then rewrite the tests to use them.

To be able to bind the data attributes to components (like `LinkTo`), we use a great little add-on called `ember-test-selectors`:

```
$ ember install ember-test-selectors
```

Not only does this enable the `data-test` attributes to be bound on all components, it also strips these attributes in production where it'd only add clutter to the DOM elements.

Without further ceremony, let's tag our templates with such attributes:

```
1  {{!-- app/components/band-list.hbs --}}
2  <ul class="pl-2 pr-8">
3    {{#each this.bands as |item|}}
- 4      <li class="mb-2">
+ 5      <li class="mb-2" data-test-rr="band-list-item">
6        <LinkTo
7          class={{if item.isActive "border-purple-400 border-l-4 pl-2"}}
8            @route="bands.band"
9            @model={{item.band.id}}
+ 10           data-test-rr="band-link"
11          >
12            {{item.band.name}}
13          </LinkTo>
14        </li>
15      {{/each}}
16    </ul>
```

HBS

```

1  {{!-- app/components/band-navigation.hbs --}}
2  <nav
3    class="flex mb-8 text-lg"
4    role="navigation"
5  >
6    <div class="pb-1 {{if this.isActive.details "border-b-4 border-
purple-400" }}">
7      <div
8        class="pb-1 {{if this.isActive.details "border-b-4 border-
purple-400" }}"
9          data-test-rr="details-nav-item"
10         >
11           <LinkTo @route="bands.band.details">
12             Details
13           </LinkTo>
14         </div>
15
16    <div class="ml-4 pb-1 {{if this.isActive.songs "border-b-4 border-
purple-400" }}">
17      <div
18        class="ml-4 pb-1 {{if this.isActive.songs "border-b-4 border-
purple-400" }}"
19          data-test-rr="songs-nav-item"
20         >
21           <LinkTo @route="bands.band.songs">
22             Songs
23           </LinkTo>
24         </div>
25   </nav>

```

HBS

```
1 {{!!-- app/templates/bands.hbs --}}
2 {{pageTitle "Bands"}}
3
4 <div class="w-1/3 px-2">
5   <BandList @bands={{@model}} />
6   <LinkTo
7     @route="bands.new"
8     class="inline-block ml-2 mt-2 p-2 rounded bg-purple-600 shadow-md
9       hover:shadow-lg hover:text-white hover:bg-purple-500 focus:outline-
10      none"
11      data-test-rr="new-band-button"
12    >
13      Add band
14    </LinkTo>
15  </div>
16  <div class="w-2/3 px-2">
17    {{outlet}}
18  </div>
```

HBS

```

1  {{!!-- app/templates/bands/band/songs.hbs --}}
2  {{pageTitle @model.name " songs | Rock & Roll with Octane"
replace=true}}
3
4  {{#if @model.songs.length}}
5    <ul>
6      {{#each @model.songs as |song|}}
7        <li class="mb-2">
8          {{song.title}}
9          <span class="float-right">
10            <StarRating
11              @rating={{song.rating}}
12              @onUpdate={{fn this.updateRating song}}
13            />
14          </span>
15        </li>
16      {{/each}}
17    </ul>
18  {{else}}
- 19    <p class="text-center">
+ 20    <p class="text-center" data-test-rr="no-songs-text">
21      The band has no songs yet.
22    </p>
23  {{/if}}
24  {{#if this.showAddSong}}
25    <div class="flex justify-center mt-2">
26      <button
27        type="button"
28        class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-lg
hover:text-white hover:bg-purple-500 focus:outline-none"
29        {{on "click" (set this "showAddSong" false)}}
30      >
31        Add song
32      </button>
33    </div>
34  {{else}}
35    <div class="mt-6 flex">

```

HBS

```
36     <input
37         type="text"
38         class="text-gray-800 bg-white rounded-md py-2 px-4"
39         placeholder="Title"
40         value={{this.title}}
41         {{on "input" this.updateTitle}}
42     />
43     <button
44         type="button"
45         class="ml-4 px-4 py-2 rounded bg-purple-600 shadow-md
46         hover:shadow-lg hover:text-white"
47         {{on "click" this.saveSong}}
48     >
49         Save
50     </button>
51     <button
52         type="button"
53         class="ml-2 px-4 p-2 rounded bg-white border border-bg-
54         purple-600 shadow-md text-purple-600 hover:shadow-lg"
55         {{on "click" this.cancel}}
56     >
57         Cancel
58     </button>
59   </div>
60 {{/if}}
```

```

1 {{!!-- app/templates/bands/new.hbs --}}
2 {{pageTitle "New band"}}
3
4 <h3 class="text-lg leading-6 font-medium text-gray-100">
5   New band
6 </h3>
7 <div class="mt-6 grid grid-cols-1 row-gap-6 col-gap-4 sm:grid-cols-6">
8   <div class="sm:col-span-4">
9     <label for="name" class="block text-sm font-medium
leading-5">Name</label>
10    <div class="mt-1 relative rounded-md shadow-sm">
11      <input
12        id="name"
13        class="w-full text-gray-800 bg-white border rounded-md py-2
px-4"
14        value={{this.name}}
+ 15        data-test-rr="new-band-name"
16        {{on "input" this.updateName}}
17      />
18    </div>
19  </div>
20  <div class="sm:col-span-4">
21    <button
22      type="button"
23      class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-lg
hover:text-white"
+ 24    data-test-rr="save-band-button"
25    {{on "click" this.saveBand}}
26    >
27      Save
28    </button>
29  </div>
30 </div>

```

HBS

As the add-on establishes attribute bindings for data attributes that start with `data-test-` (note the final hyphen), just writing `data-test` is not enough: that's why I used the `rr` suffix in all attribute names.

We can now replace the brittle DOM selectors in our tests:

```

1 // tests/acceptance/bands-test.js
2 import { module, test } from 'qunit';
3 import { visit, click, fillIn, waitFor } from '@ember/test-helpers';
4 import { setupApplicationTest } from 'ember-qunit';
5 import { setupMirage } from 'ember-cli-mirage/test-support';
6 import { getPageTitle } from 'ember-page-title/test-support';
7
8 module('Acceptance | Bands', function(hooks) {
9   setupApplicationTest(hooks);
10  setupMirage(hooks);
11
12  test('List bands', async function(assert) {
13    this.server.create('band', { name: 'Radiohead' });
14    this.server.create('band', { name: 'Long Distance Calling' });
15
16    await visit('/');
17    assert.equal(getPageTitle(), 'Bands | Rock & Roll with Octane');
18
19    let bandLinks = document.querySelectorAll('.mb-2 > a');
20    assert.equal(bandLinks.length, 2, 'All band links are rendered');
21    assert.ok(
22      bandLinks[0].textContent.includes('Radiohead'),
23      'First band link contains the band name'
24    );
25    assert.ok(
26      bandLinks[1].textContent.includes('Long Distance Calling'),
27      'The other band link contains the band name'
28    );
29  });
30
31  test('Create a band', async function(assert) {
32    this.server.create('band', { name: 'Royal Blood' });
33
34    await visit('/');
35    await click('a[href="/bands/new"]');
36    await fillIn('input', 'Caspian');
37    await click('button');

```

```

- 38     await waitFor('p.text-center');
+ 39     await click('[data-test-rr="new-band-button"]');
+ 40     await fillIn('[data-test-rr="new-band-name"]', 'Caspian');
+ 41     await click('[data-test-rr="save-band-button"]');
+ 42     await waitFor('[data-test-rr="no-songs-text"]');
43
- 44     let bandLinks = document.querySelectorAll('.mb-2 > a');
+ 45     let bandLinks = document.querySelectorAll('[data-test-rr="band-link"]');
46     assert.equal(
47         bandLinks.length,
48         2,
49         'All band links are rendered',
50         'A new band link is rendered'
51     );
52     assert.ok(
53         bandLinks[1].textContent.includes('Caspian'),
54         'The new band link is rendered as the last item'
55     );
56     assert.ok(
57         document
- 58         .querySelector('.border-b-4.border-purple-400')
+ 59         .querySelector('[data-test-rr="songs-nav-item"]')
60         .textContent.includes('Songs'),
61         'The Songs tab is active'
62     );
63 });
64 });

```

Using more descriptive (and shorter) assertions

If we take a look at our assertions, we realize they are very verbose but not very informative. QUnit is a general purpose JavaScript test framework, so it only provides a handful of basic assertions (like `assert.ok`, `assert.equal`, `assert.throws`, and so on). It'd be nice if we could make those assertions more DOM specific.

Luckily for us, there is a package that does just that, called `qunit-dom`, that's installed by default in all Ember.js applications.

Armed with this great tool, we can rewrite our assertions:

```

1 // tests/acceptance/bands-test.js
2 import { module, test } from 'qunit';
3 import { visit, click, fillIn, waitFor } from '@ember/test-helpers';
4 import { setupApplicationTest } from 'ember-qunit';
5 import { setupMirage } from 'ember-cli-mirage/test-support';
6 import { getPageTitle } from 'ember-page-title/test-support';
7
8 module('Acceptance | Bands', function(hooks) {
9   setupApplicationTest(hooks);
10  setupMirage(hooks);
11
12  test('List bands', async function(assert) {
13    this.server.create('band', { name: 'Radiohead' });
14    this.server.create('band', { name: 'Long Distance Calling' });
15
16    await visit('/');
17    assert.equal(getPageTitle(), 'Bands | Rock & Roll with Octane');
18
- 19    let bandLinks = document.querySelectorAll('.mb-2 > a');
- 20    assert.equal(bandLinks.length, 2, 'All band links are rendered');
- 21    assert.ok(
- 22      bandLinks[0].textContent.includes('Radiohead'),
- 23      'First band link contains the band name'
- 24    );
- 25    assert.ok(
- 26      bandLinks[1].textContent.includes('Long Distance Calling'),
- 27      'The other band link contains the band name'
- 28    );
+ 29    assert
+ 30      .dom('[data-test-rr="band-link"]')
+ 31      .exists({ count: 2 }, 'All band links are rendered');
+ 32    assert
+ 33      .dom('[data-test-rr="band-list-item"]:first-child')
+ 34      .hasText('Radiohead', 'The first band link contains the band
name');
+ 35    assert
+ 36      .dom('[data-test-rr="band-list-item"]:last-child')

```

```

+ 37         .hasText(
+ 38             'Long Distance Calling',
+ 39             'The other band link contains the band name'
+ 40         );
41     });
42
43     test('Create a band', async function (assert) {
44         this.server.create('band', { name: 'Royal Blood' });
45
46         await visit('/');
47         await click('[data-test-rr="new-band-button"]');
48         await fillIn('[data-test-rr="new-band-name"]', 'Caspian');
49         await click('[data-test-rr="save-band-button"]');
50         await waitFor('[data-test-rr="no-songs-text"]');
51
- 52         let bandLinks = document.querySelectorAll('[data-test-rr="band-link"]');
- 53         assert.equal(
- 54             bandLinks.length,
- 55             2,
- 56             'All band links are rendered',
- 57             'A new band link is rendered'
- 58         );
- 59         assert.ok(
- 60             bandLinks[1].textContent.includes('Caspian'),
- 61             'The new band link is rendered as the last item'
- 62         );
- 63         assert.ok(
- 64             document
- 65                 .querySelector('[data-test-rr="songs-nav-item"]')
- 66                 .textContent.includes('Songs'),
- 67                 'The Songs tab is active'
- 68         );
+ 69         assert
+ 70             .dom('[data-test-rr="band-list-item"])
+ 71             .exists({ count: 2 }, 'A new band link is rendered');
+ 72         assert
+ 73             .dom('[data-test-rr="band-list-item"]:last-child')

```

```
+ 74     .hasText('Caspian', 'The new band link is rendered as the last
+ 75     item');
+ 76     assert(
+ 77       .dom('[data-test-rr="songs-nav-item"] > .active')
+ 77       .exists('The Songs tab is active'));
78   });
79 };
```

Writing a custom test helper

Wouldn't it be cool if we could express the necessary steps taken in our tests in the language of the application? If instead of saying, "go here, click this then fill out that," we could say "create a band with this name"?

If you weren't sure, that was a rhetorical question. We can do this in Ember.js quite easily, so let's do it for the creation of a band.

Let's open a new file for our custom test helpers:

```
1 // tests/helpers/custom-helpers.js
2 import { click, fillIn } from '@ember/test-helpers';
3
4 export async function createBand(name) {
5   await click('[data-test-rr="new-band-button"]');
6   await fillIn('[data-test-rr="new-band-name"]', name);
7   return click('[data-test-rr="save-band-button"]');
8 }
```

There is nothing special about the test helper. It's just a JavaScript module with an exported function. To be able to use it, we have to import it in our tests (just as we do with the "built-in" Ember test helpers):

```

1 // tests/acceptance/bands-test.js
2 import { module, test } from 'qunit';
3 import { visit, click, fillIn, waitFor } from '@ember/test-helpers';
4 + import { visit, waitFor } from '@ember/test-helpers';
5 import { setupApplicationTest } from 'ember-qunit';
6 import { setupMirage } from 'ember-cli-mirage/test-support';
7 import { getPageTitle } from 'ember-page-title/test-support';
8 + import { createBand } from 'rarwe/tests/helpers/custom-helpers';

9
10 module('Acceptance | Bands', function(hooks) {
11     setupApplicationTest(hooks);
12     setupMirage(hooks);
13
14     test('List bands', async function(assert) {
15         this.server.create('band', { name: 'Radiohead' });
16         this.server.create('band', { name: 'Long Distance Calling' });
17
18         await visit('/');
19         assert.equal(getPageTitle(), 'Bands | Rock & Roll with Octane');
20
21         assert
22             .dom('[data-test-rr="band-link"]')
23             .exists({ count: 2 }, 'All band links are rendered');
24         assert
25             .dom('[data-test-rr="band-list-item"]:first-child')
26             .hasText('Radiohead', 'The first band link contains the band name');
27         assert
28             .dom('[data-test-rr="band-list-item"]:last-child')
29             .hasText(
30                 'Long Distance Calling',
31                 'The other band link contains the band name'
32             );
33     });
34
35     test('Create a band', async function(assert) {
36         this.server.create('band', { name: 'Royal Blood' });

```

```

37
38     await visit('/');
- 39     await click('[data-test-rr="new-band-button"]');
- 40     await fillIn('[data-test-rr="new-band-name"]', 'Caspian');
- 41     await click('[data-test-rr="save-band-button"]');
+ 42     await createBand('Caspian');
43     await waitFor('[data-test-rr="no-songs-text"]');
44
45     assert
46         .dom('[data-test-rr="band-list-item"]')
47         .exists({ count: 2 }, 'A new band link is rendered');
48     assert
49         .dom('[data-test-rr="band-list-item"]:last-child')
50         .hasText('Caspian', 'The new band link is rendered as the last
item');
51     assert
52         .dom('[data-test-rr="songs-nav-item"] > .active')
53         .exists('The Songs tab is active');
54     });
55 });

```

Our tests are now both descriptive and robust. Writing a third test scenario for creating a song is left as an exercise for the reader.

Tips for faster debugging

Let me finish the section on acceptance testing with a couple of tips that have served me extremely well in identifying problems in tests or the scenarios they verify. I can guarantee they will help you in your debugging adventures, too.

`pauseTest` and `resumeTest`

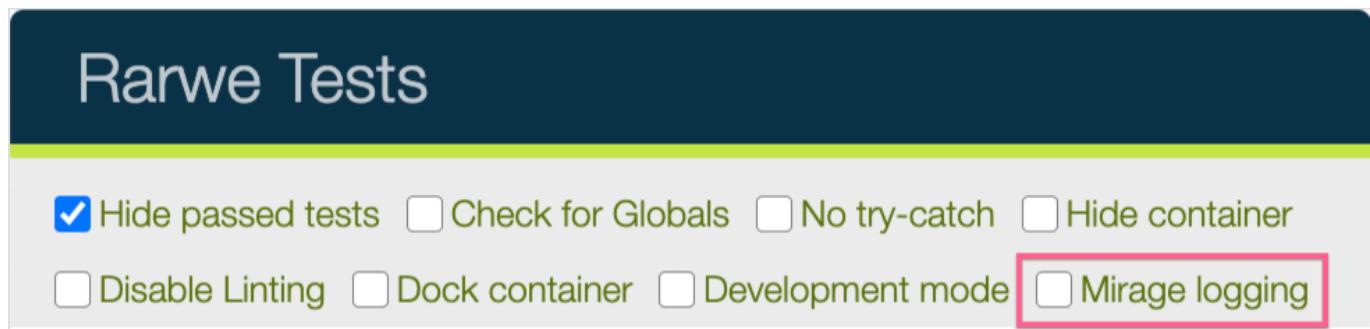
As acceptance tests are run, you can see the app under test being exercised in a small window. When a test fails, however, the app is torn down so you don't see what was on screen at that moment. The solution to this aching problem is a test helper called `pauseTest`. You can insert it anywhere in your tests by calling

`await this.pauseTest()`, and you can then observe the app at will.

Once you saw what you needed to see, you can then resume the test with `resumeTest`, imported from the same module, `@ember/test-helpers`. You can even use `pauseTest` in rendering and unit tests (which we'll see shortly).

Logging the response of the Mirage server

In some (a lot of) situations, it's crucial to be able to see what the mock server returns to see if that conforms to the data we expect (and, ideally, to the one the real back-end produces). To do this, we just have to insert a single line, `this.server.logging = true`, at the beginning of the test case we want the verbose logging in. In ember-cli-mirage versions 1.1.6 and above, there's a checkbox in the QUnit header that achieves the same (without having to modify the test):



Writing an integration test

Components in Ember are tested via integration (or rendering) tests. Let's type the following command:

```
1 $ ember g component-test star-rating
2 installing component-test
3     create tests/integration/components/star-rating-test.js
```

Ember CLI has generated the following test stub for us:

```

1 // tests/integration/components/star-rating-test.js
2 import { module, test } from 'qunit';
3 import { setupRenderingTest } from 'ember-qunit';
4 import { render } from '@ember/test-helpers';
5 import { hbs } from 'ember-cli-htmlbars';
6
7 module('Integration | Component | star-rating', function(hooks) {
8   setupRenderingTest(hooks);
9
10  test('it renders', async function(assert) {
11    // Set any properties with this.set('myProperty', 'value');
12    // Handle any actions with this.set('myAction', function(val) {
13    ... });
14
15    await render(hbs`<StarRating />`);
16
17    assert.equal(this.element.textContent.trim(), '');
18
19    // Template block usage:
20    await render(hbs`  

21      <StarRating>
22        template block text
23      </StarRating>
24    `);
25
26    assert.equal(this.element.textContent.trim(), 'template block
27      text');
28  });
29});
```

This looks really similar to the acceptance test stub. The first difference is that `setupRenderingTest` is imported instead of `setupApplicationTest` as a way to prepare the context for the appropriate test. The second difference is that `render` is imported, which can render a handlebars template within the test.

Other than that, it's the same method: `async` functions are used to wait for the result of asynchronous operations and we can use the same assertions as before.

Before we make the test meaningful, let's add a few data-test attributes to the component:

```
1 // app/components/star-rating.hbs
2 {{#each this.stars as |star|}}
- 3 <button type="button" {{on "click" (fn @onUpdate star.rating)}}>
+ 4 <button
+ 5   type="button"
+ 6   data-test-rr="star-rating-button"
+ 7   {{on "click" (fn @onUpdate star.rating)}}
+ 8 >
9   <FontAwesomeIcon
10    @icon="star"
11    @prefix={{if star.full "fas" "far"}}
+ 12    data-test-rr={{if star.full "full-star" "empty-star" }}
13  />
14 </button>
15 {{/each}}
16
```

Let's now fill up our test with assertions that are more relevant to our star-rating component. As almost all of the content is new, I'm showing the whole file, not just the diff:

```

1 // tests/integration/components/star-rating-test.js
2 import { module, test } from 'qunit';
3 import { setupRenderingTest } from 'ember-qunit';
4 import { render, click } from '@ember/test-helpers';
5 import { hbs } from 'ember-cli-htmlbars';
6
7 module('Integration | Component | star-rating', function(hooks) {
8   setupRenderingTest(hooks);
9
10  test('Renders the full and empty stars correctly', async function(assert) {
11    this.set('rating', 4);
12    this.set('updateRating', () => {});
13
14    await render(hbs``);
15  });
16
17  assert
18    .dom('[data-test-rr="full-star"]')
19    .exists({ count: 4 }, 'The right amount of full stars is rendered');
20
21  assert
22    .dom('[data-test-rr="empty-star"]')
23    .exists({ count: 1 }, 'The right amount of empty stars is rendered');
24
25  this.set('rating', 2);
26
27  assert
28    .dom('[data-test-rr="full-star"]')
29    .exists(
30      { count: 2 },
31      'The right amount of full stars is rendered after changing rating'
32
33

```

```
34      );
35      assert
36          .dom(' [data-test-rr="empty-star"] ')
37          .exists(
38              { count: 3 },
39              'The right amount of empty stars is rendered after changing
rating'
40      );
41  });
42});
```

The way Ember.js sets up these tests makes it very convenient to change properties in the context and see how the rendered component changes: we just have to set properties on `this`. We first verify that the correct number of full and empty stars are rendered initially and then change `this.rating` and check that the number of full and empty stars changes accordingly.

Another feature that is valuable to test is whether clicking one of the stars correctly calls the passed in action with the appropriate rating. We have the necessary test attributes, so let's go ahead:

```

1 // tests/integration/components/star-rating-test.js
2 import { module, test } from 'qunit';
3 import { setupRenderingTest } from 'ember-qunit';
4 import { render, click } from '@ember/test-helpers';
5 import { hbs } from 'ember-cli-htmlbars';
6
7 module('Integration | Component | star-rating', function(hooks) {
8   setupRenderingTest(hooks);
9
10  test('Renders the full and empty stars correctly', async function(assert) {
11    this.set('rating', 4);
12    this.set('updateRating', () => {});
13
14    await render(hbs``);
15  });
16
17  assert
18    .dom('[data-test-rr="full-star"]')
19    .exists({ count: 4 }, 'The right amount of full stars is rendered');
20
21  assert
22    .dom('[data-test-rr="empty-star"]')
23    .exists({ count: 1 }, 'The right amount of empty stars is rendered');
24
25  this.set('rating', 2);
26
27  assert
28    .dom('[data-test-rr="full-star"]')
29    .exists(
30      { count: 2 },
31      'The right amount of full stars is rendered after changing rating'
32
33

```

```

34         );
35     assert
36         .dom(' [data-test-rr="empty-star"] ')
37         .exists(
38             { count: 3 },
39             'The right amount of empty stars is rendered after changing
rating'
40         );
41     });
+ 42
+ 43     test('Calls onUpdate with the correct value', async function
(assert) {
+ 44         this.set('rating', 2);
+ 45         this.set('updateRating', (rating) => {
+ 46             assert.step(`Updated to rating: ${rating}`);
+ 47         });
+ 48
+ 49         await render(hbs`

```

Note that setting actions (event handlers) happens the same way we set other properties: we don't have to use the `action` decorator to bind the context of our event handler, `updateRating`. As opposed to the "real" app, we run everything in the same context (the context of the test), so there's no need to.

`assert.step` and `assert.verifySteps` is a nifty tool in QUnit.

Every time `assert.step` is called, it records the argument it was called with. We can then verify the order and equality of these arguments by calling `assert.verifySteps` and passing the expected values. Here, we only use it for a single value but it's especially useful when we want to check the order or calls.

If the star with a rating of 4 is clicked, we expect `assert.step` to be called with `Updated to rating: 4` so that's what our assertion verifies.

Writing unit tests

A unit test sits at the lowest level of the testing pyramid: it tests the entity (more informally: the thing) in isolation. Some of the pieces of Ember apps –that we have seen– suitable for unit testing are services, controllers, and models.

One of the small details I like about Ember's (kick-ass, in my opinion) testing story is that, as long as there's a blueprint for it, you don't have to know which type of test to write for a specific piece of your application: you just tell which piece you write the test for. So was the case for components, and so it is for services.

As we want to write a test for a service, we simply generate a service test:

```
1 $ ember g service-test catalog
2 installing service-test
3   create tests/unit/services/catalog-test.js
```

Next, we test the correct working of the `add` method, both for bands and songs:

```

1 // tests/unit/services/catalog-test.js
2 import { module, test } from 'qunit';
3 import { setupTest } from 'ember-qunit';
+ 4 import Band from 'rarwe/models/band';
+ 5 import Song from 'rarwe/models/song';
6
- 7 module('Unit | Service | catalog', function(hooks) {
+ 8 module('Unit | Service | catalog', function (hooks) {
9   setupTest(hooks);
10
- 11   // TODO: Replace this with your real tests.
- 12   test('it exists', function(assert) {
- 13     let service = this.owner.lookup('service:catalog');
- 14     assert.ok(service);
+ 15   test('it can store and retrieve bands', function (assert) {
+ 16     let catalog = this.owner.lookup('service:catalog');
+ 17     catalog.add('band', new Band({ id: 1, name: 'Led Zeppelin' }));
+ 18
+ 19     assert.equal(catalog.bands.length, 1);
+ 20     assert.equal(catalog.bands[0].name, 'Led Zeppelin');
+ 21   });
+ 22
+ 23   test('it can store and retrieve songs', function (assert) {
+ 24     let catalog = this.owner.lookup('service:catalog');
+ 25     catalog.add(
+ 26       'song',
+ 27       new Song({
+ 28         id: 1,
+ 29         title: 'Achilles Last Stand',
+ 30         rating: 5,
+ 31       })
+ 32     );
+ 33     assert.equal(catalog.songs.length, 1);
+ 34     assert.equal(catalog.songs[0].title, 'Achilles Last Stand');
35   });
36 });

```

The interesting bit is `this.owner`, which gives access to the application instance running in the test container. Its `lookup` method retrieves the referred entity from the registry. `service:catalog`, `controller:bands/new`, and `component:star-rating` are valid values, the first part denoting the "type" of the entity we want to fetch.

Another thing I'd test is the `load` method as it many code paths run through it. No new knowledge is needed to test it so I'll leave it to you to decide if you want to write that test yourself.

Ready?

Here's my version without further comments.

```

1 // tests/unit/services/catalog-test.js
2 import { module, test } from 'qunit';
3 import { setupTest } from 'ember-qunit';
4 import Band from 'rarwe/models/band';
5 import Song from 'rarwe/models/song';
6
7 module('Unit | Service | catalog', function (hooks) {
8   setupTest(hooks);
9
10  test('it can store and retrieve bands', function (assert) {
11    let catalog = this.owner.lookup('service:catalog');
12    catalog.add('band', new Band({ id: 1, name: 'Led Zeppelin' }));
13
14    assert.equal(catalog.bands.length, 1);
15    assert.equal(catalog.bands[0].name, 'Led Zeppelin');
16  });
17
18  test('it can store and retrieve songs', function (assert) {
19    let catalog = this.owner.lookup('service:catalog');
20    catalog.add(
21      'song',
22      new Song({
23        id: 1,
24        title: 'Achilles Last Stand',
25        rating: 5,
26      })
27    );
28    assert.equal(catalog.songs.length, 1);
29    assert.equal(catalog.songs[0].title, 'Achilles Last Stand');
30  });
31
32  test('it can load a record from a JSON:API response', function (
33    assert) {
34    let catalog = this.owner.lookup('service:catalog');
35    catalog.load({
36      data: {
37        type: 'bands',

```

```

+ 37         id: '1',
+ 38         attributes: {
+ 39             name: 'TOOL',
+ 40         },
+ 41         relationships: {
+ 42             songs: {
+ 43                 links: {
+ 44                     related: '/bands/1/songs',
+ 45                 },
+ 46             },
+ 47         },
+ 48     },
+ 49 });
+ 50     let band = catalog.bands[0];
+ 51     assert.equal(band.name, 'TOOL');
+ 52     assert.equal(band.relationships.songs, '/bands/1/songs');
+ 53 });
+ 54 });

```

Related hits

- Modern Ember Testing
- ember-cli-mirage
- ember-test-helpers
- settled ember-test helper – source code
- ember-test-selectors
- qunit-dom
- ember-mocha

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

Query params

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.

What are 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 after which post these 25 should 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 into the framework.

There is a one-to-one mapping between the query parameter in the URL and the corresponding property on the controller. A binding is automatically set up between these two that keeps them in sync.

Because each route has its corresponding controller, the list of query parameters can also be defined on the route. Because I think the concept of query parameters primarily relates to routing, I prefer to define them on the route.

We'll see how to do that in a moment, so let's add the ability to sort songs.

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 first create a separate file for testing the songs page, as we only have one for bands:

```
$ ember g acceptance-test songs
```

Following TDD principles, let's write a test that we expect to fail. We don't have to include everything in the

test we wish to have implemented by the time we'll have finished the feature – we move little by little and possibly learn a thing or two along the way.

So let's begin by just making sure the songs are sorted alphabetically when first arriving at the page:

```

1 // tests/acceptance/songs-test.js
2 import { module, test } from 'qunit';
- 3 import { visit, currentURL } from '@ember/test-helpers';
+ 4 import { visit, currentURL, click } from '@ember/test-helpers';
5 import { setupApplicationTest } from 'ember-qunit';
+ 6 import { setupMirage } from 'ember-cli-mirage/test-support';
7
- 8 module('Acceptance | songs', function(hooks) {
+ 9 module('Acceptance | Songs', function (hooks) {
10   setupApplicationTest(hooks);
+ 11   setupMirage(hooks);
12
- 13   test('visiting /songs', async function(assert) {
- 14     await visit('/songs');
+ 15     test('Sort songs in various ways', async function (assert) {
+ 16       let band = this.server.create('band', { name: 'Them Crooked
Vultures' });
+ 17       this.server.create('song', {
+ 18         title: 'Mind Eraser, No Chaser',
+ 19         rating: 2,
+ 20         band,
+ 21       });
+ 22       this.server.create('song', { title: 'Elephants', rating: 4, band
});
+ 23       this.server.create('song', {
+ 24         title: 'Spinning in Daffodils',
+ 25         rating: 5,
+ 26         band,
+ 27       });
+ 28       this.server.create('song', { title: 'New Fang', rating: 3, band
});
29
- 30       assert.equal(currentURL(), '/songs');
+ 31       await visit('/');
+ 32       await click('[data-test-rr=band-link]');
+ 33
+ 34       assert

```

```
+ 35     .dom('[data-test-rr=song-list-item]:first-child')
+ 36     .hasText(
+ 37         'Elephants',
+ 38         'The first song is the one that comes first in the alphabet'
+ 39     );
+ 40     assert
+ 41     .dom('[data-test-rr=song-list-item]:last-child')
+ 42     .hasText(
+ 43         'Spinning in Daffodils',
+ 44         'The last song is the one that comes last in the alphabet'
+ 45     );
46     });
47 }) ;
```

For that to have a chance to pass, we need to first add the `data-test-rr` attribute we refer in the test to the items of the song list:

```

1  {{!!-- app/templates/bands/band/songs.hbs --}}
2  {{pageTitle @model.name " songs | Rock & Roll with Octane"
replace=true}}
3
4  {{#if @model.songs.length}}
5    <ul>
6      {{#each @model.songs as |song|}}
- 7      <li class="mb-2">
+ 8      <li class="mb-2" data-test-rr="song-list-item">
9        {{song.title}}
10       <span class="float-right">
11         <StarRating
12           @rating={{song.rating}}
13           @onUpdate={{fn this.updateRating song}}
14         />
15       </span>
16     </li>
17   {{/each}}
18 </ul>
19 {{else}}
20 <p class="text-center" data-test-rr="no-songs-text">
21   The band has no songs yet.
22 </p>
23 {{/if}}
24 {{#if this.showAddSong}}
25 <div class="flex justify-center mt-2">
26   <button
27     type="button"
28     class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-lg
hover:text-white hover:bg-purple-500 focus:outline-none"
29     {{on "click" (set this "showAddSong" false)}}
30   >
31   Add song
32   </button>
33 </div>
34 {{else}}
35 <div class="mt-6 flex">
```

HBS

```

36   <input
37     type="text"
38     class="text-gray-800 bg-white rounded-md py-2 px-4"
39     placeholder="Title"
40     value={{this.title}}
41     {{on "input" this.updateTitle}}
42   />
43   <button
44     type="button"
45     class="ml-4 px-4 py-2 rounded bg-purple-600 shadow-md
46       hover:shadow-lg hover:text-white"
47     {{on "click" this.saveSong}}
48   >
49     Save
50   </button>
51   <button
52     type="button"
53     class="ml-2 px-4 p-2 rounded bg-white border border-bg-
54       purple-600 shadow-md text-purple-600 hover:shadow-lg"
55     {{on "click" this.cancel}}
56   >
57     Cancel
58   </button>
59 </div>
60 {{/if}}

```

We create a band (Them Crooked Vultures; if you haven't seen them, check out their [live performance of Elephants](#), it's absolutely outstanding!) and a few related songs on which we can test the various sorting options. So far so good, when we run that test, it fails because the songs appear in the order we added them, not in alphabetical order.

We have a failing test, so in true TDD-style we can start working on the implementation now. We establish the following plan of attack:

1. List the songs in ascending order of their title
2. Allow for different sorting criteria
3. Add buttons to the UI that change the sorting criteria
4. Create a query parameter that records the sorting criteria

Listing the songs in ascending order of their title

Let's define a `sortedSongs` property that will hold the sorted list of songs, derived from `this.model.songs` (the model of this route is the band object).

```

1 // app/controllers/bands/band/songs.js
2 import Controller from '@ember/controller';
3 import { tracked } from '@glimmer/tracking';
4 import { action } from '@ember/object';
5 import { inject as service } from '@ember/service';
6
7 export default class BandsBandSongsController extends Controller {
8   @tracked showAddSong = true;
9   @tracked title = '';
10
11   @service catalog;
12
+ 13   get sortedSongs() {
+ 14     return [...this.model.songs].sort((song1, song2) => {
+ 15       if (song1.title < song2.title) {
+ 16         return -1;
+ 17       }
+ 18       if (song1.title > song2.title) {
+ 19         return 1;
+ 20       }
+ 21       return 0;
+ 22     });
+ 23   }
+ 24
25   @action
26   async updateRating(song, rating) {
27     song.rating = rating;
28     this.catalog.update('song', song, { rating });
29   }
30
31   @action
32   updateTitle(event) {
33     this.title = event.target.value;
34   }
35
36   @action
37   async saveSong() {

```

```

38     let song = await this.catalog.create(
39         'song',
40         { title: this.title },
41         { band: { data: { id: this.model.id, type: 'bands' } } }
42     );
43     this.model.songs = [...this.model.songs, song];
44     this.title = '';
45     this.showAddSong = true;
46 }
47
48 @action
49 cancel() {
50     this.title = '';
51     this.showAddSong = true;
52 }
53 }
```

The `sort` function in JavaScript takes a "compare" function. If the first item is smaller, it needs to return a negative value. If the second, a positive value. If they are equal, zero. Also, `sort` mutates the array it's called on, so to be safe, we make a copy. (It also returns the sorted array.)

In the template, we simply have to loop through the sorted songs:

```

1  {{!-- app/templates/bands/band/songs.hbs --}}
2  {{pageTitle @model.name " songs | Rock & Roll with Octane"
replace=true}}
3
4  {{#if @model.songs.length}}
5  {{#if this.sortedSongs.length}}
6    <ul>
7      {{#each @model.songs as |song|}}
8        {{#each this.sortedSongs as |song|}}
9          <li class="mb-2" data-test-rr="song-list-item">
10            {{song.title}}
11            <span class="float-right">
12              <StarRating
13                @rating={{song.rating}}
14                @onUpdate={{fn this.updateRating song}}
15              />
16            </span>
17          </li>
18        {{/each}}
19      </ul>
20    {{else}}
21      <p class="text-center" data-test-rr="no-songs-text">
22        The band has no songs yet.
23      </p>
24    {{/if}}
25  {{#if this.showAddSong}}
26    <div class="flex justify-center mt-2">
27      <button
28        type="button"
29        class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-lg
hover:text-white hover:bg-purple-500 focus:outline-none"
30        {{on "click" (set this "showAddSong" false)}}
31      >
32        Add song
33      </button>
34    </div>
35  {{else}}

```

```

36  <div class="mt-6 flex">
37    <input
38      type="text"
39      class="text-gray-800 bg-white rounded-md py-2 px-4"
40      placeholder="Title"
41      value={{this.title}}
42      {{on "input" this.updateTitle}}
43    />
44    <button
45      type="button"
46      class="ml-4 px-4 py-2 rounded bg-purple-600 shadow-md
47      hover:shadow-lg hover:text-white"
48      {{on "click" this.saveSong}}
49    >
50      Save
51    </button>
52    <button
53      type="button"
54      class="ml-2 px-4 p-2 rounded bg-white border border-bg-
55      purple-600 shadow-md text-purple-600 hover:shadow-lg"
56      {{on "click" this.cancel}}
57    >
58      Cancel
59    </button>
59  {{/if}}

```

Our test verifies if the songs are sorted in the order we just implemented, so it now passes:

The screenshot shows a test runner interface with the title "Rarwe Tests". At the top, there are several checkboxes for configuration: "Hide passed tests" (unchecked), "Check for Globals" (unchecked), "No try-catch" (unchecked), "Hide container" (checked), "Disable Linting" (unchecked), "Dock container" (unchecked), "Development mode" (unchecked), and "Mirage logging" (unchecked). Below these are buttons for "Filter: sort", "Go", and "Module: All modules". A dropdown arrow is visible on the right. The main area displays the test results: "QUnit 2.14.0; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_16_0) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/80.0.3987.0 Safari/537.36". It shows "1 tests completed in 74 milliseconds, with 0 failed, 0 skipped, and 0 todo." and "2 assertions of 2 passed, 0 failed." Below this, a link "Acceptance | Songs: Sort songs in various ways (2)" is followed by "Rerun" and "74 ms".

As my guitar teacher used to say, "easy as playing a solo in a pentatonic scale". Yeah, I know – let's move on.

Allowing for different sorting criteria

Adhering to TDD, let's write a test first that assumes the UI has buttons to change the sorting property and our app actually sorts songs based on that property.

```

1 // tests/acceptance/songs-test.js
2 import { module, test } from 'qunit';
3 import { visit, currentURL, click } from '@ember/test-helpers';
4 import { setupApplicationTest } from 'ember-qunit';
5 import { setupMirage } from 'ember-cli-mirage/test-support';
6
7 module('Acceptance | Songs', function(hooks) {
8   setupApplicationTest(hooks);
9   setupMirage(hooks);
10
11   test('Sort songs in various ways', async function(assert) {
12     let band = this.server.create('band', { name: 'Them Crooked
Vultures' });
13     this.server.create('song', {
14       title: 'Mind Eraser, No Chaser',
15       rating: 2,
16       band,
17     });
18     this.server.create('song', { title: 'Elephants', rating: 4, band
});
19     this.server.create('song', {
20       title: 'Spinning in Daffodils',
21       rating: 5,
22       band,
23     });
24     this.server.create('song', { title: 'New Fang', rating: 3, band });
25
26     await visit('/');
27     await click('[data-test-rr=band-link]');
28
29     assert
30       .dom('[data-test-rr=song-list-item]:first-child')
31       .hasText(
32         'Elephants',
33         'The first song is the one that comes first in the alphabet'
34       );
35     assert

```

```

36     .dom('[data-test-rr=song-list-item]:last-child')
37     .hasText(
38         'Spinning in Daffodils',
39         'The last song is the one that comes last in the alphabet'
40     );
+ 41
+ 42     await click('[data-test-rr=sort-by-title-desc]');
+ 43     assert
+ 44     .dom('[data-test-rr=song-list-item]:first-child')
+ 45     .hasText(
+ 46         'Spinning in Daffodils',
+ 47         'The first song is the one that comes last in the alphabet'
+ 48     );
+ 49     assert
+ 50     .dom('[data-test-rr=song-list-item]:last-child')
+ 51     .hasText(
+ 52         'Elephants',
+ 53         'The last song is the one that comes first in the alphabet'
+ 54     );
+ 55
+ 56     await click('[data-test-rr=sort-by-rating-asc]');
+ 57     assert
+ 58     .dom('[data-test-rr=song-list-item]:first-child')
+ 59     .hasText('Mind Eraser, No Chaser', 'The first song is the
lowest rated');
+ 60     assert
+ 61     .dom('[data-test-rr=song-list-item]:last-child')
+ 62     .hasText('Spinning in Daffodils', 'The last song is the highest
rated');
+ 63
+ 64     await click('[data-test-rr=sort-by-rating-desc]');
+ 65     assert
+ 66     .dom('[data-test-rr=song-list-item]:first-child')
+ 67     .hasText('Spinning in Daffodils', 'The first song is the
highest rated');
+ 68     assert
+ 69     .dom('[data-test-rr=song-list-item]:last-child')
+ 70     .hasText('Mind Eraser, No Chaser', 'The last song is the lowest')

```

```
    rated');
71      });
72  });


```

Sorry, test, there are no buttons to push so it's understandable that you're angry.

Let's calm it down by adding those sorting buttons to the songs template – and then making them work.

```

1  {{!-- app/templates/bands/band/songs.hbs --}}
2  {{pageTitle @model.name " songs | Rock & Roll with Octane"
replace=true}}
3
4  {{#if this.sortedSongs.length}}
+ 5   <div class="mb-8">
+ 6     <span class="relative z-0 inline-flex shadow-sm">
+ 7       <button
+ 8         type="button"
+ 9         class="rounded-l-md inline-flex items-center px-4 py-2 border
border-gray-500 bg-purple-600 leading-5 font-medium text-gray-100
hover:text-white hover:bg-purple-500"
+ 10        data-test-rr="sort-by-title-asc"
+ 11        {{on "click" (set this "sortBy" "title")}}>
+ 12      >
+ 13      Title
+ 14      <FaIcon
+ 15        class="ml-2"
+ 16        @icon="angle-up"
+ 17        @prefix="fas"
+ 18      />
+ 19    </button>
+ 20    <button
+ 21      type="button"
+ 22      class="-ml-px inline-flex items-center px-4 py-2 border
border-gray-500 bg-purple-600 leading-5 font-medium text-gray-100
hover:text-white hover:bg-purple-500"
+ 23      data-test-rr="sort-by-title-desc"
+ 24      {{on "click" (set this "sortBy" "-title")}}>
+ 25      >
+ 26      Title
+ 27      <FaIcon
+ 28        class="ml-2"
+ 29        @icon="angle-down"
+ 30        @prefix="fas"
+ 31      />
+ 32    </button>

```

```

+ 33   <button
+ 34     class="-ml-px inline-flex items-center px-4 py-2 border
+       border-gray-500 bg-purple-600 leading-5 font-medium text-gray-100
+       hover:text-white hover:bg-purple-500"
+ 35   type="button"
+ 36   data-test-rr="sort-by-rating-asc"
+ 37   {{on "click" (set this "sortBy" "rating")}}
+ 38   >
+ 39   Rating
+ 40   <FaIcon
+ 41     class="ml-2"
+ 42     @icon="angle-up"
+ 43     @prefix="fas"
+ 44   />
+ 45   </button>
+ 46   <button
+ 47     type="button"
+ 48     class="rounded-r-md -ml-px inline-flex items-center px-4 py-2
+       border border-gray-500 bg-purple-600 leading-5 font-medium text-
+       gray-100 hover:bg-purple-500 hover:text-white"
+ 49     data-test-rr="sort-by-rating-desc"
+ 50     {{on "click" (set this "sortBy" "-rating")}}
+ 51   >
+ 52   Rating
+ 53   <FaIcon
+ 54     class="ml-2"
+ 55     @icon="angle-down"
+ 56     @prefix="fas"
+ 57   />
+ 58   </button>
+ 59   </span>
+ 60   </div>
61   <ul>
62   {{#each this.sortedSongs as |song|}}
63   <li class="mb-2" data-test-rr="song-list-item">
64     {{song.title}}
65     <span class="float-right">
66       <StarRating

```

```

67          @rating={{song.rating}}
68          @onUpdate={{fn this.updateRating song}}
69      />
70      </span>
71  </li>
72 {{/each}}
73 </ul>
74 {{else}}
75 <p class="text-center" data-test-rr="no-songs-text">
76     The band has no songs yet.
77 </p>
78 {{/if}}
79 {{#if this.showAddSong}}
80 <div class="flex justify-center mt-2">
81     <button
82         type="button"
83         class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-lg
hover:text-white hover:bg-purple-500 focus:outline-none"
84         {{on "click" (set this "showAddSong" false)}}
85     >
86         Add song
87     </button>
88 </div>
89 {{else}}
90 <div class="mt-6 flex">
91     <input
92         type="text"
93         class="text-gray-800 bg-white rounded-md py-2 px-4"
94         placeholder="Title"
95         value={{this.title}}
96         {{on "input" this.updateTitle}}>
97     </>
98     <button
99         type="button"
100        class="ml-4 px-4 py-2 rounded bg-purple-600 shadow-md
shadow-lg hover:text-white"
101        {{on "click" this.saveSong}}>
102    </>

```

```
103     Save
104   </button>
105   <button
106     type="button"
107     class="ml-2 px-4 p-2 rounded bg-white border border-bg-
purple-600 shadow-md text-purple-600 hover:shadow-lg"
108     {{on "click" this.cancel}}
109   >
110   Cancel
111   </button>
112 </div>
113 {{/if}}
```

That's a lot of lines for a few nifty purple buttons (and they cry out to be enveloped in slick `UIButton` components).

The main functional piece is the `click` event handler we attach to the buttons. Our good friend, the `set` helper, makes sure to set the `sortBy` property on the controller to the desired value. A leading dash (-) is used to indicate descending sorting.

Our test now fails for the reason we'd expect: the `sortBy` property is not taken into account when calculating `sortedSongs`. Let's remedy that:

```

1 // app/controllers/bands/band/songs.js
2 import Controller from '@ember/controller';
3 import { tracked } from '@glimmer/tracking';
4 import { action } from '@ember/object';
5 import { inject as service } from '@ember/service';
6
7 export default class BandsBandSongsController extends Controller {
8   @tracked showAddSong = true;
9   @tracked title = '';
+ 10  @tracked sortBy = 'title';
11
12  @service catalog;
13
14  get sortedSongs() {
+ 15    let sortBy = this.sortBy;
+ 16    let isDescendingSort = false;
+ 17    if (sortBy.charAt(0) === '-') {
+ 18      sortBy = this.sortBy.slice(1);
+ 19      isDescendingSort = true;
+ 20    }
21    return [...this.model.songs].sort((song1, song2) => {
- 22      if (song1.title < song2.title) {
- 23        return -1;
+ 24        if (song1[sortBy] < song2[sortBy]) {
+ 25          return isDescendingSort ? 1 : -1;
26        }
- 27        if (song1.title > song2.title) {
- 28          return 1;
+ 29        if (song1[sortBy] > song2[sortBy]) {
+ 30          return isDescendingSort ? -1 : 1;
21      }
22      return 0;
23    });
24  }
25
26  @action
27  async updateRating(song, rating) {

```

```

38     song.rating = rating;
39     this.catalog.update('song', song, { rating });
40 }
41
42 @action
43 updateTitle(event) {
44     this.title = event.target.value;
45 }
46
47 @action
48 async saveSong() {
49     let song = await this.catalog.create(
50         'song',
51         { title: this.title },
52         { band: { data: { id: this.model.id, type: 'bands' } } }
53     );
54     this.model.songs = [...this.model.songs, song];
55     this.title = '';
56     this.showAddSong = true;
57 }
58
59 @action
60 cancel() {
61     this.title = '';
62     this.showAddSong = true;
63 }
64 }
```

With both the UI and the adjusted sorting code in place, the test has now turned green:

Creating a query parameter for sorting

Now, we get to finally do what we came for. We can now add a query parameter for sorting.

Let's first extend the tests to check that the param has appeared in the URL.

```

1 // tests/acceptance/songs-test.js
2 import { module, test } from 'qunit';
3 import { visit, currentURL, click } from '@ember/test-helpers';
4 import { setupApplicationTest } from 'ember-qunit';
5 import { setupMirage } from 'ember-cli-mirage/test-support';
6
7 module('Acceptance | Songs', function(hooks) {
8   setupApplicationTest(hooks);
9   setupMirage(hooks);
10
11   test('Sort songs in various ways', async function(assert) {
12     let band = this.server.create('band', { name: 'Them Crooked
Vultures' });
13     this.server.create('song', {
14       title: 'Mind Eraser, No Chaser',
15       rating: 2,
16       band,
17     });
18     this.server.create('song', { title: 'Elephants', rating: 4, band
});
19     this.server.create('song', {
20       title: 'Spinning in Daffodils',
21       rating: 5,
22       band,
23     });
24     this.server.create('song', { title: 'New Fang', rating: 3, band });
25
26     await visit('/');
27     await click('[data-test-rr=band-link]');
28
29     assert
30       .dom('[data-test-rr=song-list-item]:first-child')
31       .hasText(
32         'Elephants',
33         'The first song is the one that comes first in the alphabet'
34       );
35     assert

```

```

36     .dom('[data-test-rr=song-list-item]:last-child')
37     .hasText(
38       'Spinning in Daffodils',
39       'The last song is the one that comes last in the alphabet'
40     );
41
42     await click('[data-test-rr=sort-by-title-desc]');
43     assert
44     .dom('[data-test-rr=song-list-item]:first-child')
45     .hasText(
46       'Spinning in Daffodils',
47       'The first song is the one that comes last in the alphabet'
48     );
49     assert
50     .dom('[data-test-rr=song-list-item]:last-child')
51     .hasText(
52       'Elephants',
53       'The last song is the one that comes first in the alphabet'
54     );
+ 55     assert.ok(
+ 56       currentURL().includes('s=-title'),
+ 57       'The sort query param appears in the URL with the correct value'
+ 58     );
59
60     await click('[data-test-rr=sort-by-rating-asc]');
61     assert
62     .dom('[data-test-rr=song-list-item]:first-child')
63     .hasText('Mind Eraser, No Chaser', 'The first song is the lowest
rated');
64     assert
65     .dom('[data-test-rr=song-list-item]:last-child')
66     .hasText('Spinning in Daffodils', 'The last song is the highest
rated');
+ 67     assert.ok(
+ 68       currentURL().includes('s=rating'),
+ 69       'The sort query param appears in the URL with the correct value'
+ 70     );
71

```

```

72     await click('[data-test-rr=sort-by-rating-desc]');
73     assert
74         .dom('[data-test-rr=song-list-item]:first-child')
75             .hasText('Spinning in Daffodils', 'The first song is the highest
76             rated');
77         assert
78             .dom('[data-test-rr=song-list-item]:last-child')
79                 .hasText('Mind Eraser, No Chaser', 'The last song is the lowest
80             rated');
+ 81             assert.ok(
+ 82                 currentURL().includes('s=-rating'),
+ 83                     'The sort query param appears in the URL with the correct value'
+ 84             );
85         });
86     });

```

The test reveals that we want the query param for sorting to appear as `s` in the URL. It's also worth noting we don't test the `s=title` case. When the value of the query param is its default value (as seen on the controller where we set `sortBy`), Ember doesn't add the QP to the URL.

Now comes the best part. All it takes to define that query parameter –and to make the test pass– is the following tiny change:

```

1 // app/routes/bands/band/songs.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
4
5 export default class BandsBandSongsRoute extends Route {
6   @service catalog;
7
+ 8   queryParams = {
+  9     sortBy: {
+ 10       as: 's',
+ 11     },
+ 12   };
+ 13
14   async model() {
15     let band = this.modelFor('bands.band');
16     await this.catalog.fetchRelated(band, 'songs');
17     return band;
18   }
19
20   resetController(controller) {
21     controller.title = '';
22     controller.showAddSong = true;
23   }
24 }
```

That's it, three extra lines. This is as easy as adding a dash of salt at the end of cooking a dish. Ember makes JavaScript oh so tasty.

The screenshot shows a test results page titled "Rarwe Tests". At the top, there are several checkboxes for test configuration: "Hide passed tests" (unchecked), "Check for Globals" (unchecked), "No try-catch" (unchecked), "Hide container" (checked), "Disable Linting" (unchecked), "Dock container" (unchecked), "Development mode" (unchecked), and "Mirage logging" (unchecked). Below these are buttons for "Filter: sort" and "Go", and a dropdown menu set to "Module: All modules". A blue header bar displays the environment: "QUnit 2.14.0; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_16_0) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/80.0.3987.0 Safari/537.36". The main content area shows a summary: "1 tests completed in 110 milliseconds, with 0 failed, 0 skipped, and 0 todo." and "11 assertions of 11 passed, 0 failed.". Below this is a link to "Acceptance | Songs: Sort songs in various ways (11)" followed by a "Rerun" button and a timestamp of "110 ms".

Using links instead of buttons

We can improve the 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 this, only the template needs to change:

```

1  {{!-- app/templates/bands/band/songs.hbs --}}
2  {{pageTitle @model.name " songs | Rock & Roll with Octane"
replace=true}}
3
4  {{#if this.sortedSongs.length}}
5    <div class="mb-8">
6      <span class="relative z-0 inline-flex shadow-sm">
- 7        <button
- 8          type="button"
+ 9        <LinkTo
10          class="rounded-l-md inline-flex items-center px-4 py-2 border
border-gray-500 bg-purple-600 leading-5 font-medium text-gray-100
hover:text-white hover:bg-purple-500"
11          data-test-rr="sort-by-title-asc"
- 12          {{on "click" (set this "sortBy" "title")}}
+ 13          @query={{hash s="title"}}
14        >
15        Title
16        <FaIcon
17          class="ml-2"
18          @icon="angle-up"
19          @prefix="fas"
20        />
- 21        </button>
- 22        <button
- 23          type="button"
+ 24        </LinkTo>
+ 25        <LinkTo
26          class="-ml-px inline-flex items-center px-4 py-2 border border-
gray-500 bg-purple-600 leading-5 font-medium text-gray-100 hover:text-
white hover:bg-purple-500"
27          data-test-rr="sort-by-title-desc"
- 28          {{on "click" (set this "sortBy" "-title")}}
+ 29          @query={{hash s="-title"}}
30        >
31        Title
32        <FaIcon

```

```

33         class="ml-2"
34         @icon="angle-down"
35         @prefix="fas"
36     />
- 37     </button>
- 38     <button
+ 39     </LinkTo>
+ 40     <LinkTo
41         class="-ml-px inline-flex items-center px-4 py-2 border border-
gray-500 bg-purple-600 leading-5 font-medium text-gray-100 hover:text-
white hover:bg-purple-500"
42         type="button"
43         data-test-rr="sort-by-rating-asc"
- 44         { {on "click" (set this "sortBy" "rating") } }
+ 45         @query={{hash s="rating"}}
46     >
47     Rating
48     <FaIcon
49         class="ml-2"
50         @icon="angle-up"
51         @prefix="fas"
52     />
- 53     </button>
- 54     <button
+ 55     </LinkTo>
+ 56     <LinkTo
57         type="button"
58         class="rounded-r-md -ml-px inline-flex items-center px-4 py-2
border border-gray-500 bg-purple-600 leading-5 font-medium text-
gray-100 hover:bg-purple-500 hover:text-white"
59         data-test-rr="sort-by-rating-desc"
- 60         { {on "click" (set this "sortBy" "-rating") } }
+ 61         @query={{hash s="-rating"}}
62     >
63     Rating
64     <FaIcon
65         class="ml-2"
66         @icon="angle-down"

```

```

67             @prefix="fas"
68         />
- 69     </button>
+ 70     </LinkTo>
71     </span>
72   </div>
73   <ul>
74     {{#each this.sortedSongs as |song|}}
75       <li class="mb-2" data-test-rr="song-list-item">
76         {{song.title}}
77         <span class="float-right">
78           <StarRating
79             @rating={{song.rating}}
80             @onUpdate={{fn this.updateRating song}}
81           />
82         </span>
83       </li>
84     {{/each}}
85   </ul>
86 {{else}}
87   <p class="text-center" data-test-rr="no-songs-text">
88     The band has no songs yet.
89   </p>
90 {{/if}}
91 {{#if this.showAddSong}}
92   <div class="flex justify-center mt-2">
93     <button
94       type="button"
95       class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-lg
96       hover:text-white hover:bg-purple-500 focus:outline-none"
97       {{on "click" (set this "showAddSong" false)}}
98     >
99       Add song
100      </button>
101    </div>
102 {{else}}
103   <div class="mt-6 flex">
104     <input

```

```

104     type="text"
105     class="text-gray-800 bg-white rounded-md py-2 px-4"
106     placeholder="Title"
107     value={{this.title}}
108     {{on "input" this.updateTitle}}
109   />
110   <button
111     type="button"
112     class="ml-4 px-4 py-2 rounded bg-purple-600 shadow-md
113     hover:shadow-lg hover:text-white"
114     {{on "click" this.saveSong}}
115   >
116     Save
117   </button>
118   <button
119     type="button"
120     class="ml-2 px-4 p-2 rounded bg-white border border-bg-
121     purple-600 shadow-md text-purple-600 hover:shadow-lg"
122     {{on "click" this.cancel}}
123   >
124     Cancel
125   </button>
126 </div>
127 {{/if}}

```

Note the relevant parts: the `<LinkTo>` component can take a `@query` parameter that contains an object that the query parameter values should be set to.

Also note that we omitted the route name argument. Instead of writing this:

```
<LinkTo @route="bands.band.songs" @query={{hash s="rating"}}>
```

HBS

, we wrote the following:

```
<LinkTo @query={{hash s="rating"}}>
```

HBS

, 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 and that when one hovers over the buttons, the URL is displayed just as with "ordinary" links.

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:

```

1 // tests/acceptance/songs-test.js
2 import { module, test } from 'qunit';
3 import { visit, currentURL, click } from '@ember/test-helpers';
+ 4 import { visit, currentURL, click, fillIn } from '@ember/test-helpers';
5 import { setupApplicationTest } from 'ember-qunit';
6 import { setupMirage } from 'ember-cli-mirage/test-support';
7
8 module('Acceptance | Songs', function(hooks) {
9   setupApplicationTest(hooks);
10  setupMirage(hooks);
11
12  test('Sort songs in various ways', async function(assert) {
13    let band = this.server.create('band', { name: 'Them Crooked
Vultures' });
14    this.server.create('song', {
15      title: 'Mind Eraser, No Chaser',
16      rating: 2,
17      band,
18    });
19    this.server.create('song', { title: 'Elephants', rating: 4, band
});
20    this.server.create('song', {
21      title: 'Spinning in Daffodils',
22      rating: 5,
23      band,
24    });
25    this.server.create('song', { title: 'New Fang', rating: 3, band });
26
27    await visit('/');
28    await click('[data-test-rr=band-link]');
29
30    assert
31      .dom('[data-test-rr=song-list-item]:first-child')
32      .hasText(
33        'Elephants',
34        'The first song is the one that comes first in the alphabet'
35      );

```

```

36     assert
37         .dom('[data-test-rr=song-list-item]:last-child')
38         .hasText(
39             'Spinning in Daffodils',
40             'The last song is the one that comes last in the alphabet'
41         );
42
43     await click('[data-test-rr=sort-by-title-desc]');
44     assert
45         .dom('[data-test-rr=song-list-item]:first-child')
46         .hasText(
47             'Spinning in Daffodils',
48             'The first song is the one that comes last in the alphabet'
49         );
50     assert
51         .dom('[data-test-rr=song-list-item]:last-child')
52         .hasText(
53             'Elephants',
54             'The last song is the one that comes first in the alphabet'
55         );
56     assert.ok(
57         currentURL().includes('s==title'),
58         'The sort query param appears in the URL with the correct value'
59     );
60
61     await click('[data-test-rr=sort-by-rating-asc]');
62     assert
63         .dom('[data-test-rr=song-list-item]:first-child')
64         .hasText('Mind Eraser, No Chaser', 'The first song is the lowest
rated');
65     assert
66         .dom('[data-test-rr=song-list-item]:last-child')
67         .hasText('Spinning in Daffodils', 'The last song is the highest
rated');
68     assert.ok(
69         currentURL().includes('s=rating'),
70         'The sort query param appears in the URL with the correct value'
71     );

```

```

72
73     await click('[data-test-rr=sort-by-rating-desc]');
74     assert
75         .dom('[data-test-rr=song-list-item]:first-child')
76         .hasText('Spinning in Daffodils', 'The first song is the highest
rated');
77     assert
78         .dom('[data-test-rr=song-list-item]:last-child')
79         .hasText('Mind Eraser, No Chaser', 'The last song is the lowest
rated');
80     assert.ok(
81         currentURL().includes('s=-rating'),
82         'The sort query param appears in the URL with the correct value'
83     );
84 });
+ 85
+ 86 test('Search songs', async function (assert) {
+ 87     let band = this.server.create('band', { name: 'Them Crooked
Vultures' });
+ 88     this.server.create('song', {
+ 89         title: 'Mind Eraser, No Chaser',
+ 90         rating: 2,
+ 91         band,
+ 92     });
+ 93     this.server.create('song', { title: 'Elephants', rating: 4, band
});
+ 94     this.server.create('song', {
+ 95         title: 'Spinning in Daffodils',
+ 96         rating: 5,
+ 97         band,
+ 98     });
+ 99     this.server.create('song', { title: 'New Fang', rating: 3, band
});
+100    this.server.create('song', {
+101        title: 'No One Loves Me & Neither Do I',
+102        rating: 4,
+103        band,
+104    });

```

```

+105
+106    await visit('/');
+107    await click('[data-test-rr=band-link]');
+108    await fillIn('[data-test-rr=search-box]', 'no');
+109
+110    assert
+111        .dom('[data-test-rr=song-list-item]')
+112            .exists({ count: 2 }, 'The songs matching the search term are
+113            displayed');
+114
+115    await click('[data-test-rr=sort-by-title-desc]');
+116    assert
+117        .dom('[data-test-rr=song-list-item]:first-child')
+118            .hasText(
+119                'No One Loves Me & Neither Do I',
+120                'A matching song that comes later in the alphabet appears on
+121                top');
+122    assert
+123        .dom('[data-test-rr=song-list-item]:last-child')
+124            .hasText(
+125                'Mind Eraser, No Chaser',
+126                'A matching song that comes sooner in the alphabet appears at
+127                the bottom');
+128 });

```

The test prescribes a field with a data attribute of `data-test-rr=search-box` 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):

```

1 // app/controllers/bands/band/songs.js
2 import Controller from '@ember/controller';
3 import { tracked } from '@glimmer/tracking';
4 import { action } from '@ember/object';
5 import { inject as service } from '@ember/service';
6
7 export default class BandsBandSongsController extends Controller {
8     @tracked showAddSong = true;
9     @tracked title = '';
10    @tracked sortBy = 'title';
11    @tracked searchTerm = '';
12
13    @service catalog;
14
15    get matchingSongs() {
16        let searchTerm = this.searchTerm.toLowerCase();
17        return this.model.songs.filter((song) => {
18            return song.title.toLowerCase().includes(searchTerm);
19        });
20    }
21
22    get sortedSongs() {
23        let sortBy = this.sortBy;
24        let isDescendingSort = false;
25        if (sortBy.charAt(0) === '-') {
26            sortBy = this.sortBy.slice(1);
27            isDescendingSort = true;
28        }
29        return [...this.model.songs].sort((song1, song2) => {
30            if (song1[sortBy] < song2[sortBy]) {
31                return isDescendingSort ? 1 : -1;
32            }
33            if (song1[sortBy] > song2[sortBy]) {
34                return isDescendingSort ? -1 : 1;
35            }
36            return 0;
37        });
}

```

```

38     }
39
40     @action
41     async updateRating(song, rating) {
42         song.rating = rating;
43         this.catalog.update('song', song, { rating });
44     }
45
46     @action
47     updateTitle(event) {
48         this.title = event.target.value;
49     }
50
51     @action
52     async saveSong() {
53         let song = await this.catalog.create(
54             'song',
55             { title: this.title },
56             { band: { data: { id: this.model.id, type: 'bands' } } }
57         );
58         this.model.songs = [...this.model.songs, song];
59         this.title = '';
60         this.showAddSong = true;
61     }
62
63     @action
64     cancel() {
65         this.title = '';
66         this.showAddSong = true;
67     }
68 }
```

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` (instead of `this.model.songs`):

```
1 // app/controllers/bands/band/songs.js
2 import Controller from '@ember/controller';
3 import { tracked } from '@glimmer/tracking';
4 import { action } from '@ember/object';
5 import { inject as service } from '@ember/service';
6
7 export default class BandsBandSongsController extends Controller {
8     @tracked showAddSong = true;
9     @tracked title = '';
10    @tracked sortBy = 'title';
11    @tracked searchTerm = '';
12
13    @service catalog;
14
15    get matchingSongs() {
16        let searchTerm = this.searchTerm.toLowerCase();
17        return this.model.songs.filter((song) => {
18            return song.title.toLowerCase().includes(searchTerm);
19        });
20    }
21
22    get sortedSongs() {
23        let sortBy = this.sortBy;
24        let isDescendingSort = false;
25        if (sortBy.charAt(0) === '-') {
26            sortBy = this.sortBy.slice(1);
27            isDescendingSort = true;
28        }
29        return [...this.model.songs].sort((song1, song2) => {
30            return this.matchingSongs.sort((song1, song2) => {
31                if (song1[sortBy] < song2[sortBy]) {
32                    return isDescendingSort ? 1 : -1;
33                }
34                if (song1[sortBy] > song2[sortBy]) {
35                    return isDescendingSort ? -1 : 1;
36                }
37            return 0;
38        });
39    }
40}
```

```

38     });
39 }
40
41 @action
42 async updateRating(song, rating) {
43     song.rating = rating;
44     this.catalog.update('song', song, { rating });
45 }
46
47 @action
48 updateTitle(event) {
49     this.title = event.target.value;
50
51
52 @action
53 async saveSong() {
54     let song = await this.catalog.create(
55         'song',
56         { title: this.title },
57         { band: { data: { id: this.model.id, type: 'bands' } } }
58     );
59     this.model.songs = [...this.model.songs, song];
60
61     this.title = '';
62     this.showAddSong = true;
63 }
64 @action
65 cancel() {
66     this.title = '';
67     this.showAddSong = true;
68 }
69 }
```

The only reason for the template to change is to incorporate the search panel:

```

1  {{!!-- app/templates/bands/band/songs.hbs --}}
2  {{pageTitle @model.name " songs | Rock & Roll with Octane"
replace=true}}
3
- 4 {{#if this.sortedSongs.length}}
- 5   <div class="mb-8">
+ 6   <div class="mb-8 flex">
+ 7     {{#if this.sortedSongs.length}}
8       <span class="relative z-0 inline-flex shadow-sm">
9         <LinkTo
10           class="rounded-l-md inline-flex items-center px-4 py-2 border
border-gray-500 bg-purple-600 leading-5 font-medium text-gray-100
hover:text-white hover:bg-purple-500"
11             data-test-rr="sort-by-title-asc"
12             @query={{hash s="title"}}
13           >
14             Title
15             <FaIcon
16               class="ml-2"
17               @icon="angle-up"
18               @prefix="fas"
19             />
20           </LinkTo>
21           <LinkTo
22             class="-ml-px inline-flex items-center px-4 py-2 border border-
gray-500 bg-purple-600 leading-5 font-medium text-gray-100 hover:text-
white hover:bg-purple-500"
23             data-test-rr="sort-by-title-desc"
24             @query={{hash s="-title"}}
25           >
26             Title
27             <FaIcon
28               class="ml-2"
29               @icon="angle-down"
30               @prefix="fas"
31             />
32           </LinkTo>

```

HBS

```

33     <LinkTo
34         class="-ml-px inline-flex items-center px-4 py-2 border border-
35             gray-500 bg-purple-600 leading-5 font-medium text-gray-100 hover:text-
36             white hover:bg-purple-500"
37             type="button"
38             data-test-rr="sort-by-rating-asc"
39             @query={{hash s="rating"}}
40         >
41             Rating
42             <FaIcon
43                 class="ml-2"
44                 @icon="angle-up"
45                 @prefix="fas"
46             />
47         </LinkTo>
48         <LinkTo
49             type="button"
50             class="rounded-r-md -ml-px inline-flex items-center px-4 py-2
51             border border-gray-500 bg-purple-600 leading-5 font-medium text-
52             gray-100 hover:bg-purple-500 hover:text-white"
53             data-test-rr="sort-by-rating-desc"
54             @query={{hash s="-rating"}}
55         >
56             Rating
57             <FaIcon
58                 class="ml-2"
59                 @icon="angle-down"
60                 @prefix="fas"
61             />
62         </LinkTo>
63     </span>
+ 60 {{/if}}
+ 61 <div class="relative ml-auto rounded-md shadow-sm">
+ 62   <input
+ 63       class="border rounded-md py-2 px-3 block w-full pr-10 text-
+ 64       gray-800 sm:text-sm sm:leading-5"
+ 65       data-test-rr="search-box"
+ 66       value={{this.searchTerm}}>

```

```

+ 66         {{on "input" this.updateSearchTerm}}
+ 67     />
+ 68     <div class="absolute inset-y-0 right-0 pr-3 flex items-center
pointer-events-none">
+ 69         <FaIcon
+ 70             class="h-5 w-5 text-gray-400"
+ 71                 @icon="search"
+ 72                 @prefix="fas"
+ 73             />
+ 74         </div>
75     </div>
+ 76 </div>
+ 77 {{#if this.sortedSongs.length}}
78     <ul>
79         {{#each this.sortedSongs as |song|}}
80             <li class="mb-2" data-test-rr="song-list-item">
81                 {{song.title}}
82                 <span class="float-right">
83                     <StarRating
84                         @rating={{song.rating}}
85                         @onUpdate={{fn this.updateRating song}}
86                     />
87                 </span>
88             </li>
89         {{/each}}
90     </ul>
91 {{else}}
92     <p class="text-center" data-test-rr="no-songs-text">
93         The band has no songs yet.
94     </p>
95 {{/if}}
96 {{#if this.showAddSong}}
97     <div class="flex justify-center mt-2">
98         <button
99             type="button"
100             class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-lg
hover:text-white hover:bg-purple-500 focus:outline-none"
101             {{on "click" (set this "showAddSong" false)}}>
```

```

102      >
103          Add song
104      </button>
105  </div>
106 {{else}}
107  <div class="mt-6 flex">
108      <input
109          type="text"
110          class="text-gray-800 bg-white rounded-md py-2 px-4"
111          placeholder="Title"
112          value={{this.title}}
113          {{on "input" this.updateTitle}}>
114      />
115      <button
116          type="button"
117          class="ml-4 px-4 py-2 rounded bg-purple-600 shadow-md
118          hover:shadow-lg hover:text-white"
119          {{on "click" this.saveSong}}>
120          Save
121      </button>
122      <button
123          type="button"
124          class="ml-2 px-4 p-2 rounded bg-white border border-bg-
125          purple-600 shadow-md text-purple-600 hover:shadow-lg"
126          {{on "click" this.cancel}}>
127          Cancel
128      </button>
129  </div>
130 {{/if}}

```

The `updateSearchTerm` action doesn't exist yet, so let's quickly define it on the controller:

```

1 // app/controllers/bands/band/songs.js
2 import Controller from '@ember/controller';
3 import { tracked } from '@glimmer/tracking';
4 import { action } from '@ember/object';
5 import { inject as service } from '@ember/service';
6
7 export default class BandsBandSongsController extends Controller {
8     @tracked showAddSong = true;
9     @tracked title = '';
10    @tracked sortBy = 'title';
11    @tracked searchTerm = '';
12
13    @service catalog;
14
15    get matchingSongs() {
16        let searchTerm = this.searchTerm.toLowerCase();
17        return this.model.songs.filter((song) => {
18            return song.title.toLowerCase().includes(searchTerm);
19        });
20    }
21
22    get sortedSongs() {
23        let sortBy = this.sortBy;
24        let isDescendingSort = false;
25        if (sortBy.charAt(0) === '-') {
26            sortBy = this.sortBy.slice(1);
27            isDescendingSort = true;
28        }
29        return this.matchingSongs.sort((song1, song2) => {
30            if (song1[sortBy] < song2[sortBy]) {
31                return isDescendingSort ? 1 : -1;
32            }
33            if (song1[sortBy] > song2[sortBy]) {
34                return isDescendingSort ? -1 : 1;
35            }
36            return 0;
37        });
}

```

```

38     }
39
40     @action
41     async updateRating(song, rating) {
42         song.rating = rating;
43         this.catalog.update('song', song, { rating });
44     }
45
46     @action
47     updateTitle(event) {
48         this.title = event.target.value;
49     }
50
+ 51     @action
+ 52     updateSearchTerm(event) {
+ 53         this.searchTerm = event.target.value;
+ 54     }
+ 55
56     @action
57     async saveSong() {
58         let song = await this.catalog.create(
59             'song',
60             { title: this.title },
61             { band: { data: { id: this.model.id, type: 'bands' } } }
62         );
63         this.model.songs = [...this.model.songs, song];
64         this.title = '';
65         this.showAddSong = true;
66     }
67
68     @action
69     cancel() {
70         this.title = '';
71         this.showAddSong = true;
72     }
73 }
```

With that, our "Search songs" test now passes:

The screenshot shows a test runner interface with the following details:

- Test Title:** Rarwe Tests
- Test Status:** 1 tests completed in 82 milliseconds, with 0 failed, 0 skipped, and 0 todo.
- Assertions:** 3 assertions of 3 passed, 0 failed.
- Run Details:** Acceptance | Songs: Search songs (3) Rerun, 82 ms

Creating a query parameter for the search term

To round off this feature, let's add a query parameter for the search term. As it's almost universally denoted by `q`, let's call it that:

```

1 // app/routes/bands/band/songs.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
4
5 export default class BandsBandSongsRoute extends Route {
6   @service catalog;
7
8   queryParams = {
9     sortBy: {
10       as: 's',
11     },
12     searchTerm: {
13       as: 'q',
14     },
15   };
16
17   async model() {
18     let band = this.modelFor('bands.band');
19     await this.catalog.fetchRelated(band, 'songs');
20     return band;
21   }
22
23   resetController(controller) {
24     controller.title = '';
25     controller.showAddSong = true;
26   }
27 }

```

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:

The screenshot shows a dark-themed web application interface. At the top, the title "Rock & Roll with Octane" is displayed. Below it, a sidebar on the left lists several bands: Foo Fighters, Kaya Project, Led Zeppelin, Pearl Jam, Radiohead, and Red Hot Chili Peppers. To the right of the sidebar, there are two tabs: "Details" and "Songs", with "Songs" being the active tab. Under the "Songs" tab, there are four buttons for sorting: "Title ^", "Title ^", "Rating ^", and "Rating ^". A search bar with the placeholder "an" and a magnifying glass icon is positioned next to the sorting buttons. Below the search bar, two song entries are shown: "Animal" by Radiohead and "State of Love and Trust" by Red Hot Chili Peppers. Each entry has a five-star rating icon to its right. At the bottom left of the main area is a purple button labeled "Add band", and at the bottom right is a purple button labeled "Add song".

Next song

Cool, but what if the back-end 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 substates

Tuning

When the app "boots up," a request is sent to the back-end 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 back-end, 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.

Note: you can download the same spinner image I use from <https://s3-eu-west-1.amazonaws.com/rarwe-book/spinner.svg>. After you have downloaded it, place it in the `public/images` folder of your application.

Loading subroutes

Let's assume, just as we did in the [Nested routes chapter](#), that the router transitions from the top all the way to `bands.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:

```
1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
4
5 export default class BandsRoute extends Route {
6   @service catalog;
7
8   model() {
9     return this.catalog.fetchAll('bands');
10  }
11}
```

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:

```
1 $ ember g template loading
2 installing template
3      create app/templates/loading.hbs
```

And add some "loading content" into it:

```
1 {{!-- app/templates/loading.hbs --}}
2 <div class="flex flex-col mx-auto mt-4 items-center">
3   Loading data
4   
5 </div>
```

HBS

To simulate network latency, we can use the following function:

```
1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
4
+ 5 function wait(delay) {
+ 6   return new Promise(function(resolve) {
+ 7     setTimeout(resolve, delay);
+ 8   });
+ 9 }
+ 10
11 export default class BandsRoute extends Route {
12   @service catalog;
13
- 14   model() {
+ 15   async model() {
+ 16     await wait(3000);
17     return this.catalog.fetchAll('bands');
18   }
19 }
```

Indeed, when the bands route is entered, we see the loading screen as intended:

Rock & Roll with Octane

Loading data



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.

This implies that each level has its own loading subroutine. We can thus descend one level and create a loading subroutine for the band route. The beauty of this concept is that we only need the template to indicate that the app is loading data:

```
1 $ ember g template bands/band/loading
2 installing template
3     create app/templates/bands/band/loading.hbs
```

```
1 {{!-- app/templates/bands/band/loading.hbs --}}
2 <div class="flex flex-col mx-auto mt-4 items-center">
3   Loading data
4   
5 </div>
```

HBS

Before we add an artificial delay on the next level of routes too, let's extract the wait helper we used above.

```
1 // app/utils/wait.js
2 export default function wait(delay) {
3   return new Promise(function(resolve) {
4     setTimeout(resolve, delay);
5   });
6 }
```

We can now use our wait util function to delay the loading of songs in both `bands` and `bands.band.songs`:

```
1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
- 4
- 5 function wait(delay) {
- 6   return new Promise(function(resolve) {
- 7     setTimeout(resolve, delay);
- 8   });
- 9 }
+ 10 import wait from 'rarwe/utils/wait';
11
12 export default class BandsRoute extends Route {
13   @service catalog;
14
15   async model() {
16     await wait(3000);
17     return this.catalog.fetchAll('bands');
18   }
19 }
```

```

1 // app/routes/bands/band/songs.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
+ 4 import wait from 'rarwe/utils/wait';
5
6 export default class BandsBandSongsRoute extends Route {
7   @service catalog;
8
9   queryParams = {
10     sortBy: {
11       as: 's',
12     },
13     searchTerm: {
14       as: 'q',
15     },
16   };
17
18   async model() {
19     let band = this.modelFor('bands.band');
+ 20     await wait(3000);
20     await this.catalog.fetchRelated(band, 'songs');
21     return band;
22   }
23
24   resetController(controller) {
25     controller.title = '';
26     controller.showAddSong = true;
27   }
28 }
29 }
```

We will see our loading template instead of the songs:

Rock & Roll with Octane

Foo Fighters

Details Songs

Kaya Project

Led Zeppelin

Loading data

Pearl Jam



Radiohead

Red Hot Chili Peppers

Add band

The same loading subroutine 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:

```
1 // app/routes/bands.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
4 - import wait from 'rarwe/utils/wait';
5
6 export default class BandsRoute extends Route {
7   @service catalog;
8
9   async model() {
10     -   await wait(3000);
11     return this.catalog.fetchAll('bands');
12   }
13 }
```

```

1 // app/routes/bands/band/songs.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
- 4 import wait from 'rarwe/utils/wait';
5
6 export default class BandsBandSongsRoute extends Route {
7   @service catalog;
8
9   queryParams = {
10     sortBy: {
11       as: 's',
12     },
13     searchTerm: {
14       as: 'q',
15     },
16   };
17
18   async model() {
19     let band = this.modelFor('bands.band');
- 20     await wait(3000);
21     await this.catalog.fetchRelated(band, 'songs');
22     return band;
23   }
24
25   resetController(controller) {
26     controller.title = '';
27     controller.showAddSong = true;
28   }
29 }

```

Under normal circumstances, network latency happens on its own and so we can rest assured we'll see our pretty spinners.

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, show an alert box that the user has to dismiss (not saying this is the best option from a UX point of view, though):

```
1 import wait from 'rarwe/utils/wait';
2 import { action } from '@ember/object';
3
4 export default class SlowRoute extends Route {
5   // ...
6   @action
7   loading() {
8     window.alert("Loading the band's songs, ok?");
9   }
10 }
```

JS

Error subroutines

Error subroutines are activated when a model hook throws an error, either because the back-end returned an error response or because the application code itself ran into a problem.

Error subroutines work exactly the same way as loading subroutines:

- 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:

```
1 $ ember g template bands/band/error
2 installing template
3   create app/templates/bands/band/error.hbs
```

We can display a similar "panel" as we did for the loading template – with colors that indicate something slightly bad has happened:

```
1 {{!-- app/templates/bands/band/error.hbs --}}
2 <div class="flex items-center justify-center h-16 mx-auto mt-4 text-
  center text-red-700 bg-red-200" data-test-rr="band-loading-error">
3   Something went wrong while fetching band data, soorry.
4 </div>
```

HBS

We can test the displaying of the error template by making a rejecting promise in the model hook:

```

1 // app/routes/bands/band/songs.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
4
5 export default class BandsBandSongsRoute extends Route {
6   @service catalog;
7
8   queryParams = {
9     sortBy: {
10       as: 's',
11     },
12     searchTerm: {
13       as: 'q',
14     },
15   };
16
17   async model() {
18     let band = this.modelFor('bands.band');
19     await this.catalog.fetchRelated(band, 'songs');
20     return band;
21     return Promise.reject();
22   }
23
24   resetController(controller) {
25     controller.title = '';
26     controller.showAddSong = true;
27   }
28 }

```

And indeed the error message is rendered on screen:

Rock & Roll with Octane

Foo Fighters

[Details](#) [Songs](#)

Kaya Project

Led Zeppelin

Pearl Jam

Radiohead

Red Hot Chili Peppers

Add band

Something went wrong while fetching band data, soorry.

Let's not forget to clean up after ourselves and revert the model hook.

```

1 // app/routes/bands/band/songs.js
2 import Route from '@ember/routing/route';
3 import { inject as service } from '@ember/service';
4
5 export default class BandsBandSongsRoute extends Route {
6   @service catalog;
7
8   queryParams = {
9     sortBy: {
10       as: 's',
11     },
12     searchTerm: {
13       as: 'q',
14     },
15   };
16
17   async model() {
18     return Promise.reject();
19     let band = this.modelFor('bands.band');
20     await this.catalog.fetchRelated(band, 'songs');
21     return band;
22   }
23
24   resetController(controller) {
25     controller.title = '';
26     controller.showAddSong = true;
27   }
28 }

```

If the back-end, for whatever reason, can't load the data we're fetching in routes under `bands.band`, the user will see our nice, descriptive, helpful error message.

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 goal.

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 implement conditional logic.

We have used a great number of helpers already in our application, possibly even without being aware of it. `if`, `each`, `outlet`, and `fn` 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 markup or event handling provided by components. The following table summarizes a few typical use cases for helpers.

Category	Template content	Output
Timestamps	<code>{{ago message.createdAt}}</code>	24 minutes ago
Translation	<code>{{translate "butterfly"}}</code>	papillon
Localizations	<code>{{localize product.price}}</code>	\$4.99
Text transformations	<code>{{shout 'silence'}}</code>	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`:

```

1 $ ember g helper capitalize
2 installing helper
3   create app/helpers/capitalize.js
4 installing helper-test
5   create tests/integration/helpers/capitalize-test.js

```

That creates a `app/helpers/capitalize.js` file that we should take a look at:

```
1 // app/helpers/capitalize.js
2 import { helper } from '@ember/component/helper';
3
4 export default helper(function capitalize(params/*, hash*/) {
5   return params;
6 });
```

This creates a helper called `capitalize` we can use in our templates. The `helper` function imported from `@ember/component/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 changes, all instances which used the `capitalize` helper would rerun to produce a new output for the changed input.

Let's add the code that implements splitting apart several words on whitespace and then capitalizing the first character of each word:

```

1 // app/helpers/capitalize.js
2 import { helper } from '@ember/component/helper';
+ 3 import { capitalize as emberCapitalize } from '@ember/string';
4
- 5 export default helper(function capitalize(params/*, hash*/) {
- 6   return params;
- 7 });
+ 8 export function capitalize(input) {
+ 9   let words = input
+ 10  .toString()
+ 11  .split(/\s+/)
+ 12  .map((word) => {
+ 13    return emberCapitalize(word.charAt(0)) + word.slice(1);
+ 14  });
+ 15  return words.join(' ');
+ 16 }
+ 17
+ 18 export default helper(capitalize);

```

(We only call `capitalize` on the first character of each word because we want to allow songs that are in all caps, like MFC from Pearl Jam. If we called `capitalize` on each full word, users would not be able to create such a song title.)

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.

We used the `as` keyword in the `import` statement because the `@ember/string` module also has a `capitalize` `export` and we don't want it to clash with the new `capitalize` function.

The final step is to use the new helper in the templates, to capitalize band names:

```
1 {{!-- app/components/band-list.hbs --}}
2 <ul class="pl-2 pr-8">
3   {{#each this.bands as |item|}}
4     <li class="mb-2" data-test-rr="band-list-item">
5       <LinkTo
6         class={{if item.isActive "border-purple-400 border-l-4 pl-2"}}
7         @route="bands.band"
8         @model={{item.band.id}}
9         data-test-rr="band-link"
10      >
11        {{item.band.name}}
12        {{capitalize item.band.name}}
13      </LinkTo>
14    </li>
15  {{/each}}
16 </ul>
```

HBS

and song titles:

```

1 {{!!-- app/templates/bands/band/songs.hbs --}}
2 {{pageTitle @model.name " songs | Rock & Roll with Octane"
replace=true}}
3
4 <div class="mb-8 flex">
5   {{#if this.sortedSongs.length}}
6     <span class="relative z-0 inline-flex shadow-sm">
7       <LinkTo
8         class="rounded-l-md inline-flex items-center px-4 py-2 border
border-gray-500 bg-purple-600 leading-5 font-medium text-gray-100
hover:text-white hover:bg-purple-500"
9           data-test-rr="sort-by-title-asc"
10          @query={{hash s="title"}}>
11        >
12        Title
13        <FaIcon
14          class="ml-2"
15          @icon="angle-up"
16          @prefix="fas"
17        />
18      </LinkTo>
19      <LinkTo
20        class="-ml-px inline-flex items-center px-4 py-2 border border-
gray-500 bg-purple-600 leading-5 font-medium text-gray-100 hover:text-
white hover:bg-purple-500"
21          data-test-rr="sort-by-title-desc"
22          @query={{hash s="-title"}}>
23        >
24        Title
25        <FaIcon
26          class="ml-2"
27          @icon="angle-down"
28          @prefix="fas"
29        />
30      </LinkTo>
31      <LinkTo
32        class="-ml-px inline-flex items-center px-4 py-2 border border-

```

```

gray-500 bg-purple-600 leading-5 font-medium text-gray-100 hover:text-
white hover:bg-purple-500"
33         type="button"
34         data-test-rr="sort-by-rating-asc"
35         @query={{hash s="rating"}}
36     >
37         Rating
38         <FaIcon
39             class="ml-2"
40             @icon="angle-up"
41             @prefix="fas"
42         />
43     </LinkTo>
44     <LinkTo
45         type="button"
46         class="rounded-r-md -ml-px inline-flex items-center px-4 py-2
border border-gray-500 bg-purple-600 leading-5 font-medium text-
gray-100 hover:bg-purple-500 hover:text-white"
47         data-test-rr="sort-by-rating-desc"
48         @query={{hash s="-rating"}}
49     >
50         Rating
51         <FaIcon
52             class="ml-2"
53             @icon="angle-down"
54             @prefix="fas"
55         />
56     </LinkTo>
57   </span>
58 {{/if}}
59 <div class="relative ml-auto rounded-md shadow-sm">
60   <input
61       class="border rounded-md py-2 px-3 block w-full pr-10 text-
gray-800 sm:text-sm sm:leading-5"
62       data-test-rr="search-box"
63       value={{this.searchTerm}}
64       {{on "input" this.updateSearchTerm}}
65   />

```

```

66      <div class="absolute inset-y-0 right-0 pr-3 flex items-center
pointer-events-none">
67          <FaIcon
68              class="h-5 w-5 text-gray-400"
69                  @icon="search"
70                  @prefix="fas"
71          />
72      </div>
73  </div>
74 </div>
75 {{#if this.sortedSongs.length}}
76 <ul>
77     {{#each this.sortedSongs as |song|}}
78         <li class="mb-2" data-test-rr="song-list-item">
- 79             {{song.title}}
+ 80             {{capitalize song.title}}
81             <span class="float-right">
82                 <StarRating
83                     @rating={{song.rating}}
84                     @onUpdate={{fn this.updateRating song}}
85                 />
86             </span>
87         </li>
88     {{/each}}
89 </ul>
90 {{else}}
91 <p class="text-center" data-test-rr="no-songs-text">
92     The band has no songs yet.
93 </p>
94 {{/if}}
95 {{#if this.showAddSong}}
96 <div class="flex justify-center mt-2">
97     <button
98         type="button"
99             class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-lg
hover:text-white hover:bg-purple-500 focus:outline-none"
100             {{on "click" (set this "showAddSong" false)}}
101     >
```

```

102     Add song
103     </button>
104   </div>
105 {{else}}
106   <div class="mt-6 flex">
107     <input
108       type="text"
109       class="text-gray-800 bg-white rounded-md py-2 px-4"
110       placeholder="Title"
111       value={{this.title}}
112       {{on "input" this.updateTitle}}>
113     />
114     <button
115       type="button"
116       class="ml-4 px-4 py-2 rounded bg-purple-600 shadow-md
hover:shadow-lg hover:text-white"
117       {{on "click" this.saveSong}}>
118     >
119       Save
120     </button>
121     <button
122       type="button"
123       class="ml-2 px-4 p-2 rounded bg-white border border-bg-
purple-600 shadow-md text-purple-600 hover:shadow-lg"
124       {{on "click" this.cancel}}>
125     >
126       Cancel
127     </button>
128   </div>
129 {{/if}}

```

The main difference you can see between calling helpers and components from your templates is that helpers need to be invoked using curly braces – you can't use angle-brackets. Also, helpers usually take positional, and not named arguments.

Using it in templates, or elsewhere

It's reasonable to argue that our helper seems like a "utility" function. You know, one of those that you're never quite sure where to place in your perfectly compartmentalized file structure. It would be useful to have access to our capitalize function from JavaScript code, not just in our templates. It could be handy to apply the proper casing to band names in document titles, for example.

The solution is to restructure the JavaScript module of our helper. Let's see:

```
1 // app/helpers/capitalize.js
2 import { helper } from '@ember/component/helper';
3 import { capitalize as emberCapitalize } from '@ember/string';
4
5 export function capitalize(input) {
6   let words = input
7     .toString()
8     .split(/\s+/)
9     .map((word) => {
10       return emberCapitalize(word.charAt(0)) + word.slice(1);
11     });
12   return words.join(' ');
13 }
14
15 export default helper(capitalize);
```

I simply moved the `capitalize` function outside of the `helper` call and created a named export for it. This gives us the possibility to use that named export anywhere in or code.

We can use the helper function in JavaScript code to introduce a more telling placeholder for the new song text field:

```

1 {{!!-- app/templates/bands/band/songs.hbs --}}
2 {{pageTitle @model.name " songs | Rock & Roll with Octane"
replace=true}}
3
4 <div class="mb-8 flex">
5   {{#if this.sortedSongs.length}}
6     <span class="relative z-0 inline-flex shadow-sm">
7       <LinkTo
8         class="rounded-l-md inline-flex items-center px-4 py-2 border
border-gray-500 bg-purple-600 leading-5 font-medium text-gray-100
hover:text-white hover:bg-purple-500"
9         data-test-rr="sort-by-title-asc"
10        @query={{hash s="title"}}>
11      >
12      Title
13      <FaIcon
14        class="ml-2"
15        @icon="angle-up"
16        @prefix="fas"
17        />
18    </LinkTo>
19    <LinkTo
20      class="-ml-px inline-flex items-center px-4 py-2 border border-
gray-500 bg-purple-600 leading-5 font-medium text-gray-100 hover:text-
white hover:bg-purple-500"
21      data-test-rr="sort-by-title-desc"
22      @query={{hash s="-title"}}>
23    >
24    Title
25    <FaIcon
26      class="ml-2"
27      @icon="angle-down"
28      @prefix="fas"
29      />
30  </LinkTo>
31  <LinkTo
32    class="-ml-px inline-flex items-center px-4 py-2 border border-

```

```

gray-500 bg-purple-600 leading-5 font-medium text-gray-100 hover:text-
white hover:bg-purple-500"
33         type="button"
34         data-test-rr="sort-by-rating-asc"
35         @query={{hash s="rating"}}
36     >
37         Rating
38         <FaIcon
39             class="ml-2"
40             @icon="angle-up"
41             @prefix="fas"
42         />
43     </LinkTo>
44     <LinkTo
45         type="button"
46         class="rounded-r-md -ml-px inline-flex items-center px-4 py-2
border border-gray-500 bg-purple-600 leading-5 font-medium text-
gray-100 hover:bg-purple-500 hover:text-white"
47         data-test-rr="sort-by-rating-desc"
48         @query={{hash s="-rating"}}
49     >
50         Rating
51         <FaIcon
52             class="ml-2"
53             @icon="angle-down"
54             @prefix="fas"
55         />
56     </LinkTo>
57   </span>
58 {{/if}}
59 <div class="relative ml-auto rounded-md shadow-sm">
60   <input
61       class="border rounded-md py-2 px-3 block w-full pr-10 text-
gray-800 sm:text-sm sm:leading-5"
62       data-test-rr="search-box"
63       value={{this.searchTerm}}
64       {{on "input" this.updateSearchTerm}}
65   />

```

```

66      <div class="absolute inset-y-0 right-0 pr-3 flex items-center
pointer-events-none">
67          <FaIcon
68              class="h-5 w-5 text-gray-400"
69              @icon="search"
70              @prefix="fas"
71          />
72      </div>
73  </div>
74</div>
75 {{#if this.sortedSongs.length}}
76  <ul>
77      {{#each this.sortedSongs as |song|}}
78          <li class="mb-2" data-test-rr="song-list-item">
79              {{capitalize song.title}}
80              <span class="float-right">
81                  <StarRating
82                      @rating={{song.rating}}
83                      @onUpdate={{fn this.updateRating song}}
84                  />
85              </span>
86          </li>
87      {{/each}}
88  </ul>
89 {{else}}
90  <p class="text-center" data-test-rr="no-songs-text">
91      The band has no songs yet.
92  </p>
93 {{/if}}
94 {{#if this.showAddSong}}
95  <div class="flex justify-center mt-2">
96      <button
97          type="button"
98          class="px-4 py-2 rounded bg-purple-600 shadow-md hover:shadow-lg
hover:text-white hover:bg-purple-500 focus:outline-none"
99          {{on "click" (set this "showAddSong" false)}}
100     >
101        Add song

```

```

102      </button>
103    </div>
104  {{else}}
105    <div class="mt-6 flex">
106      <input
107        type="text"
108        class="text-gray-800 bg-white rounded-md py-2 px-4"
-109        placeholder="Title"
+110        placeholder={{this.newSongTitle}}
111        value={{this.title}}
112        {{on "input" this.updateTitle}}
113      />
114      <button
115        type="button"
116        class="ml-4 px-4 py-2 rounded bg-purple-600 shadow-md
hover:shadow-lg hover:text-white"
117        {{on "click" this.saveSong}}
118      >
119        Save
120      </button>
121      <button
122        type="button"
123        class="ml-2 px-4 p-2 rounded bg-white border border-bg-
purple-600 shadow-md text-purple-600 hover:shadow-lg"
124        {{on "click" this.cancel}}
125      >
126        Cancel
127      </button>
128    </div>
129  {{/if}}

```

Gaining access to the function is a matter of importing the named export:

```

1 // app/controllers/bands/band/songs.js
2 import Controller from '@ember/controller';
3 import { tracked } from '@glimmer/tracking';
4 import { action } from '@ember/object';
5 import { inject as service } from '@ember/service';
+ 6 import { capitalize } from 'rarwe/helpers/capitalize';
7
8 export default class BandsBandSongsController extends Controller {
9   @tracked showAddSong = true;
10  @tracked title = '';
11  @tracked sortBy = 'title';
12  @tracked searchTerm = '';
13
14  @service catalog;
15
16  get matchingSongs() {
17    let searchTerm = this.searchTerm.toLowerCase();
18    return this.model.songs.filter((song) => {
19      return song.title.toLowerCase().includes(searchTerm);
20    });
21  }
22
23  get sortedSongs() {
24    let sortBy = this.sortBy;
25    let isDescendingSort = false;
26    if (sortBy.charAt(0) === '-') {
27      sortBy = this.sortBy.slice(1);
28      isDescendingSort = true;
29    }
30    return this.matchingSongs.sort((song1, song2) => {
31      if (song1[sortBy] < song2[sortBy]) {
32        return isDescendingSort ? 1 : -1;
33      }
34      if (song1[sortBy] > song2[sortBy]) {
35        return isDescendingSort ? -1 : 1;
36      }
37      return 0;

```

```

38     });
39 }
40
+ 41     get newSongPlaceholder() {
+ 42         let bandName = this.model.name;
+ 43         return `New ${capitalize(bandName)} song`;
+ 44     }
+ 45

46     @action
47     async updateRating(song, rating) {
48         song.rating = rating;
49         this.catalog.update('song', song, { rating });
50     }

51
52     @action
53     updateTitle(event) {
54         this.title = event.target.value;
55     }

56
57     @action
58     updateSearchTerm(event) {
59         this.searchTerm = event.target.value;
60     }

61
62     @action
63     async saveSong() {
64         let song = await this.catalog.create(
65             'song',
66             { title: this.title },
67             { band: { data: { id: this.model.id, type: 'bands' } } }
68         );
69         this.model.songs = [...this.model.songs, song];
70         this.title = '';
71         this.showAddSong = true;
72     }

73
74     @action
75     cancel() {

```

```
76     this.title = '';
77     this.showAddSong = true;
78   }
79 }
```

Band names and song titles are now correctly capitalized everywhere they're displayed:

The screenshot shows a dark-themed web application interface. At the top, the title "Rock & Roll with Octane" is displayed. Below it, a sidebar lists several bands: Foo Fighters, Kaya Project, Led Zeppelin, Pearl Jam, Radiohead, and Red Hot Chili Peppers. A purple button labeled "Add band" is located at the bottom of this sidebar. To the right of the sidebar, there are two tabs: "Details" and "Songs", with "Songs" being the active tab. Under the "Songs" tab, there is a search bar with a magnifying glass icon. Below the search bar are four sorting options: "Title ^", "Title ^", "Rating ^", and "Rating ^". The main content area displays a list of songs with their corresponding ratings represented by star icons. The songs listed are Alive, Animal, Daughter, Inside Job, State Of Love And Trust, and Yellow Ledbetter. Each song has a five-star rating icon to its right.

Testing a helper

The helper generator also created a test file for the helper. Since helpers are primarily for the UI, it's an integration test with some boilerplate.

Let's make it meaningful for our use case:

```

1 // tests/integration/helpers/capitalize-test.js
2 import { module, test } from 'qunit';
3 import { setupRenderingTest } from 'ember-qunit';
4 import { render } from '@ember/test-helpers';
5 import hbs from 'htmlbars-inline-precompile';
6
7 module('Integration | Helper | capitalize', function (hooks) {
8   setupRenderingTest(hooks);
9
10  test('It capitalizes each word', async function (assert) {
11    this.set('title', 'seven nations army');
12    await render(hbs`{{capitalize title}}`);
13    assert.dom(this.element).hasText('Seven Nations Army');
14
15    this.set('title', 'MVC');
16    assert.dom(this.element).hasText('MVC');
17  });
18});

```

`this.element` refers to the top-level `#ember-testing` element – a sort of sandbox helpers are tested in.

Since we're running our tests, we realize that some of them are broken now. Specifically, some of the assertions in the `songs-test.js` assertion test fail because the song titles are not capitalized in the test. That's easy to rectify:

```

1 // tests/acceptance/songs-test.js
2 import { module, test } from 'qunit';
3 import { visit, currentURL, click, fillIn } from '@ember/test-helpers';
4 import { setupApplicationTest } from 'ember-qunit';
5 import { setupMirage } from 'ember-cli-mirage/test-support';
6
7 module('Acceptance | Songs', function(hooks) {
8   setupApplicationTest(hooks);
9   setupMirage(hooks);
10
11   test('Sort songs in various ways', async function(assert) {
12     let band = this.server.create('band', { name: 'Them Crooked
Vultures' });
13     this.server.create('song', {
14       title: 'Mind Eraser, No Chaser',
15       rating: 2,
16       band,
17     });
18     this.server.create('song', { title: 'Elephants', rating: 4, band
});
19     this.server.create('song', {
20       title: 'Spinning in Daffodils',
21       rating: 5,
22       band,
23     });
24     this.server.create('song', { title: 'New Fang', rating: 3, band });
25
26     await visit('/');
27     await click('[data-test-rr=band-link]');
28
29     assert
30       .dom('[data-test-rr=song-list-item]:first-child')
31       .hasText(
32         'Elephants',
33         'The first song is the one that comes first in the alphabet'
34       );
35     assert

```

```

36         .dom('[data-test-rr=song-list-item]:last-child')
37         .hasText(
- 38             'Spinning in Daffodils',
+ 39             'Spinning In Daffodils',
40             'The last song is the one that comes last in the alphabet'
41         );
42
43         await click('[data-test-rr=sort-by-title-desc]');
44         assert
45             .dom('[data-test-rr=song-list-item]:first-child')
46             .hasText(
- 47                 'Spinning in Daffodils',
+ 48                 'Spinning In Daffodils',
49                 'The first song is the one that comes last in the alphabet'
50             );
51         assert
52             .dom('[data-test-rr=song-list-item]:last-child')
53             .hasText(
54                 'Elephants',
55                 'The last song is the one that comes first in the alphabet'
56             );
57         assert.ok(
58             currentURL().includes('s=-title'),
59             'The sort query param appears in the URL with the correct value'
60         );
61
62         await click('[data-test-rr=sort-by-rating-asc]');
63         assert
64             .dom('[data-test-rr=song-list-item]:first-child')
65             .hasText('Mind Eraser, No Chaser', 'The first song is the lowest
rated');
66         assert
67             .dom('[data-test-rr=song-list-item]:last-child')
- 68             .hasText('Spinning in Daffodils', 'The last song is the highest
rated');
+ 69             .hasText('Spinning In Daffodils', 'The last song is the highest
rated');
70         assert.ok(

```

```

71     currentURL().includes('s=rating'),
72     'The sort query param appears in the URL with the correct value'
73   );
74
75   await click('[data-test-rr=sort-by-rating-desc]');
76   assert
77     .dom('[data-test-rr=song-list-item]:first-child')
- 78     .hasText('Spinning in Daffodils', 'The first song is the
highest rated');
+ 79     .hasText('Spinning In Daffodils', 'The first song is the
highest rated');
80   assert
81     .dom('[data-test-rr=song-list-item]:last-child')
82     .hasText('Mind Eraser, No Chaser', 'The last song is the lowest
rated');
83   assert.ok(
84     currentURL().includes('s==rating'),
85     'The sort query param appears in the URL with the correct value'
86   );
87 });
88
89 test('Search songs', async function (assert) {
90   let band = this.server.create('band', { name: 'Them Crooked
Vultures' });
91   this.server.create('song', {
92     title: 'Mind Eraser, No Chaser',
93     rating: 2,
94     band,
95   });
96   this.server.create('song', { title: 'Elephants', rating: 4, band
});
97   this.server.create('song', {
98     title: 'Spinning in Daffodils',
99     rating: 5,
100    band,
101  });
102  this.server.create('song', { title: 'New Fang', rating: 3, band });
103  this.server.create('song', {

```

```

104     title: 'No One Loves Me & Neither Do I',
105     rating: 4,
106     band,
107   });
108
109   await visit('/');
110   await click('[data-test-rr=band-link]');
111   await fillIn('[data-test-rr=search-box]', 'no');
112
113   assert
114     .dom('[data-test-rr=song-list-item]')
115       .exists({ count: 2 }, 'The songs matching the search term are
116         displayed');
117
118   await click('[data-test-rr=sort-by-title-desc]');
119   assert
120     .dom('[data-test-rr=song-list-item]:first-child')
121       .hasText(
122         'No One Loves Me & Neither Do I',
123         'A matching song that comes later in the alphabet appears on
124           top'
125       );
126   assert
127     .dom('[data-test-rr=song-list-item]:last-child')
128       .hasText(
129         'Mind Eraser, No Chaser',
130         'A matching song that comes sooner in the alphabet appears at
131           the bottom'
132       );
133   });
134 });

```

Next song

Our app has now outgrown its weekend hobby project status and we'd like to show it to our friends. We'll

take some time in the next chapter to learn how to deploy it.

CHAPTER 15

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 ways 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` add-on:

```
$ 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 add-on is installed, log in to surge from your project's directory:

```
$ surge login
```

The add-on 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`. Let's set up the `https://app.rockandrollwithemberjs.com` domain to point to surge, as described [in the docs](#).

We'll then need to modify the CNAME file to define the custom host:

```
- 1 rarwe.surge.sh
+ 2 https://app.rockandrollwithemberjs.com
```

One thing we have to watch out for before we expect our app to work in production is the network 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.

The best place to store non-sensitive configuration is the `config/environment.js` file as it also provides the ability to have different config values for each environment (by default, `development`, `test` and `production`). We only need to set our back-end API host for production:

```

1 // config/environment.js
2 'use strict';
3
4 module.exports = function (environment) {
5   let ENV = {
6     modulePrefix: 'rarwe',
7     environment,
8     rootURL: '/',
9     locationType: 'auto',
10    EmberENV: {
11      FEATURES: {
12        // Here you can enable experimental features on an ember
canary build
13        // e.g. EMBER_NATIVE_DECORATOR_SUPPORT: true
14      },
15      EXTEND_PROTOTYPES: {
16        // Prevent Ember Data from overriding Date.parse.
17        Date: false,
18      },
19    },
20
21    APP: {
22      // Here you can pass flags/options to your application instance
when it is created
23    },
24  },
25};
26
27 if (environment === 'development') {
28   // ENV.APP.LOG_RESOLVER = true;
29   // ENV.APP.LOG_ACTIVE_GENERATION = true;
30   // ENV.APP.LOG_TRANSITIONS = true;
31   // ENV.APP.LOG_TRANSITIONS_INTERNAL = true;
32   // ENV.APP.LOG_VIEW_LOOKUPS = true;
33 }
34
35 if (environment === 'test') {
36   // Testem prefers this...

```

```

37     ENV.locationType = 'none';
38
39     // keep test console output quieter
40     ENV.APP.LOG_ACTIVE_GENERATION = false;
41     ENV.APP.LOG_VIEW_LOOKUPS = false;
42
43     ENV.APP.rootElement = '#ember-testing';
44     ENV.APP.autoboot = false;
45 }
46
47 if (environment === 'production') {
- 48     // here you can enable a production-specific feature
+ 49     ENV.apiHost = 'https://json-api.rockandrollwithemberjs.com';
50 }
51
52 return ENV;
53 };

```

Being diligent about having all of our data operations in the catalog service now bears fruit. That's the only file we need to change to ensure requests are always sent to the correct URL:

```

1 // app/services/catalog.js
2 import Service from '@ember/service';
3 import Band from 'rarwe/models/band';
4 import Song from 'rarwe/models/song';
5 import { tracked } from 'tracked-built-ins';
6 import { isArray } from '@ember/array';
+ 7 import ENV from 'rarwe/config/environment';

8

9 function extractRelationships(object) {
10   let relationships = {};
11   for (let relationshipName in object) {
12     relationships[relationshipName] =
13       object[relationshipName].links.related;
14   }
15   return relationships;
16 }

17 export default class CatalogService extends Service {
18   storage = {};
19
20   constructor() {
21     super(...arguments);
22     this.storage.bands = tracked([]);
23     this.storage.songs = tracked([]);
24   }
25

+ 26   get bandsURL() {
+ 27     return `${ENV.apiHost || ''}/bands`;
+ 28   }
+ 29
+ 30   get songsURL() {
+ 31     return `${ENV.apiHost || ''}/songs`;
+ 32   }
+ 33

34   async fetchAll(type) {
35     if (type === 'bands') {
- 36       let response = await fetch('/bands');

```

```

+ 37     let response = await fetch(this.bandsURL);
38     let json = await response.json();
39     this.loadAll(json);
40     return this.bands;
41   }
42   if (type === 'songs') {
- 43     let response = await fetch('/songs');
+ 44     let response = await fetch(this.songsURL);
45     let json = await response.json();
46     this.loadAll(json);
47     return this.songs;
48   }
49 }
50
51 loadAll(json) {
52   let records = [];
53   for (let item of json.data) {
54     records.push(this._loadResource(item));
55   }
56   return records;
57 }
58
59 load(response) {
60   return this._loadResource(response.data);
61 }
62
63 _loadResource(data) {
64   let record;
65   let { id, type, attributes, relationships } = data;
66   if (type === 'bands') {
67     let rels = extractRelationships(relationships);
68     record = new Band({ id, ...attributes }, rels);
69     this.add('band', record);
70   }
71   if (type === 'songs') {
72     let rels = extractRelationships(relationships);
73     record = new Song({ id, ...attributes }, rels);
74     this.add('song', record);

```

```

75     }
76     return record;
77   }

78
79   async fetchRelated(record, relationship) {
80     let url = record.relationships[relationship];
81     let response = await fetch(url);
82     let json = await response.json();
83     if (isArray(json.data)) {
84       record[relationship] = this.loadAll(json);
85     } else {
86       record[relationship] = this.load(json);
87     }
88     return record[relationship];
89   }

90
91   async create(type, attributes, relationships = {}) {
92     let payload = {
93       data: {
94         type: type === 'band' ? 'bands' : 'songs',
95         attributes,
96         relationships,
97       },
98     };
99     let response = await fetch(type === 'band' ? '/bands' : '/songs',
100    {
101      method: 'POST',
102      headers: {
103        'Content-Type': 'application/vnd.api+json',
104      },
105      body: JSON.stringify(payload),
106    });
107    let response = await fetch(
108      type === 'band' ? this.bandsURL : this.songsURL,
109      {
110        method: 'POST',
111        headers: {
112          'Content-Type': 'application/vnd.api+json',

```

```

+112     },
+113     body: JSON.stringify(payload),
+114   }
+115 );
116   let json = await response.json();
117   return this.load(json);
118 }
119
120 async update(type, record, attributes) {
121   let payload = {
122     data: {
123       id: record.id,
124       type: type === 'band' ? 'bands' : 'songs',
125       attributes,
126     },
127   };
-128   let url = type === 'band' ? `/bands/${record.id}` :
`/songs/${record.id}`;
+129   let url =
+130     type === 'band'
+131     ? `${this.bandsURL}/${record.id}`
+132     : `${this.songsURL}/${record.id}`;
133   await fetch(url, {
134     method: 'PATCH',
135     headers: {
136       'Content-Type': 'application/vnd.api+json',
137     },
138     body: JSON.stringify(payload),
139   });
140 }
141
142 add(type, record) {
143   let collection = type === 'band' ? this.storage.bands :
this.storage.songs;
144   let recordIds = collection.map((record) => record.id);
145   if (!recordIds.includes(record.id)) {
146     collection.push(record);
147   }

```

```
148     }
149
150     get bands() {
151         return this.storage.bands;
152     }
153
154     get songs() {
155         return this.storage.songs;
156     }
157
158     find(type, filterFn) {
159         let collection = type === 'band' ? this.bands : this.songs;
160         return collection.find(filterFn);
161     }
162 }
```

We can now go ahead and deploy our application:

```
$ ember surge --environment production
```

Not so fast, says the `tracked-built-ins` add-on:

`tracked-built-ins` uses `Proxy`, which is not supported in IE 11 or other older browsers. You have included IE 11 in your targets, which is not supported. Consider using `tracked-maps-and-sets`, which does support IE 11.

This comes handy as it allows us to learn about another configuration file, `config/targets.js`. This is where we need to define which browsers we aim to support – and we can do this in a very natural language, thanks to [Browserslist](#) which this file is backed by.

Knowing which browsers the app runs in also helps with Babel.js transpilation – if a certain feature is natively included in all supported browsers, no transpilation is needed.

Happily, none of our numerous customers demands IE 11 support, so let's instead switch to Edge support in production (and CI):

```
1 // config/targets.js
2 'use strict';
3
4 const browsers = [
5   'last 1 Chrome versions',
6   'last 1 Firefox versions',
7   'last 1 Safari versions',
8 ];
9
10 const isCI = Boolean(process.env.CI);
11 const isProduction = process.env.EMBER_ENV === 'production';
12
13 if (isCI || isProduction) {
- 14   browsers.push('ie 11');
+ 15   browsers.push('last 1 Edge versions');
16 }
17
18 module.exports = {
19   browsers,
20 };
```

Now we can run the deploy command again:

```
$ ember surge --environment production
```

Our app is now live on the interwebs, at app.rockandrollwithemberjs.com.

Heroku

Deploying to [Heroku](#) is another option that is very easy to start with.

We have to install [the Heroku toolbelt](#) so that we have the `heroku` command. One also needs a Heroku account, but no further setup is needed.

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 Ember buildpack that will build our app during the deployment process:

```
1      $ heroku create --buildpack https://codon-
2          buildpacks.s3.amazonaws.com/buildpacks/heroku/emberjs.tgz
3          Creating app... done, ? salty-journey-31707
4          Setting buildpack to https://codon-buildpacks.s3.amazonaws.com/
5          buildpacks/heroku/emberjs.tgz... done
6          https://salty-journey-31707.herokuapp.com/ |
7          https://git.heroku.com/salty-journey-31707.git
```

Deploying to the Heroku platform is a matter of pushing to the master branch of the remote called `heroku`:

```

1      $ git push heroku master
2      remote: Compressing source files... done.
3      remote: Building source:
4      remote:
5      - 5      remote: > emberjs app detected
6      - 6      remote: > Setting NPM_CONFIG_PRODUCTION to false to install ember-
7      - 7      cli toolchain
8      - 8      remote: > Fetching buildpack heroku/nodejs-v98
9      remote:
10     - 10     remote: > Creating runtime environment
11    (...)

12     - 12     remote: > Installing binaries
13    (...)

14     - 14     remote: > Restoring cache
15     remote:       Skipping cache restore (new runtime signature)
16     remote:
17     - 17     remote: > Building dependencies
18     (...)

19     remote:       Done in 56.19s.

20     remote:

21     - 21     remote: > Caching build
22     (...)

23     - 23     remote: > Build succeeded!
24     (...)

25     - 25     remote: > Caching ember assets
26     remote: > Writing static.json
27     remote: > Fetching buildpack heroku/static
28     remote: > Static HTML detected

29     remote: % Total % Received % Xferd Average Speed   Time
30             Time   Current
31             Dload Upload Total
32             Spent Left Speed
33             remote: 100 838k 100 838k 0 0 12.2M 0 --:--:-- --
34             :--:-- --:--:-- 12.2M
35
36     - 36     remote: > Installed directory to /app/bin
37     - 37     remote: > Discovering process types

```

```

34      remote:          Procfile declares types      -> (none)
35      remote:          Default types for buildpack -> web
36      remote:
- 37      remote: > Compressing...
38      remote:          Done: 75.1M
- 39      remote: > Launching...
40      remote:          Released v3
41      remote:          https://salty-journey-31707.herokuapp.com/ deployed
      to Heroku
42      remote:
43      remote: Verifying deploy... done.

```

Our app is now live on Heroku:

As you can see from the above log, the installation of node, npm, and the building of the app happens during deployment, on the Heroku platform. Consequently, the whole process can take longer than the previous solution when assets were built locally and then pushed to Surge. Subsequent pushes are faster as the installed npm packages are cached for later deployments.

Albeit not the fastest, pushing to Heroku is extremely simple way to expose your app to the greater public and the stability of the platform and the amazing architecture (and add-ons) built around it are definite benefits.

Note that unlike ember-cli-surge and the ember-cli-deploy, the Heroku buildpack doesn't build the app

locally and upload the assets afterward – it does the build on the Heroku platform. That means that everything that needs to be included in the build has to be committed before pushing to Heroku. It all makes sense but I've lost a non-trivial amount of time forgetting about this.

ember-cli-deploy

[ember-cli-deploy](#) provides a plugin-based framework for deploying Ember apps.

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 ember-cli-deploy 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 is a great number 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 to S3

Let's start by installing the base package, `ember-cli-deploy`:

```
$ ember install ember-cli-deploy
```

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-cli-deploy-revision-data ember-  
cli-deploy-display-revisions ember-cli-deploy-s3 ember-cli-deploy-s3-index
```

Now, let's look inside the generated configuration file:

```

1 // config/deploy.js
2 /* eslint-env node */
3 'use strict';
4
5 module.exports = function(deployTarget) {
6     let ENV = {
7         build: {}
8         // include other plugin configuration that applies to all deploy
9         targets here
10    };
11
12    if (deployTarget === 'development') {
13        ENV.build.environment = 'development';
14        // configure other plugins for development deploy target here
15    }
16
17    if (deployTarget === 'staging') {
18        ENV.build.environment = 'production';
19        // configure other plugins for staging deploy target here
20    }
21
22    if (deployTarget === 'production') {
23        ENV.build.environment = 'production';
24        // configure other plugins for production deploy target here
25    }
26
27    // Note: if you need to build some configuration asynchronously, you
28    // can return
29    // a promise that resolves with the ENV object instead of returning
30    // the
31    // ENV object synchronously.
32    return ENV;
33}

```

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:

```

1 // config/deploy.js
2 /* eslint-env node */
3 'use strict';
4
- 5 module.exports = function(deployTarget) {
+ 6 module.exports = function (deployTarget) {
7     let ENV = {
- 8         build: {}
+ 9         build: {},
10        // include other plugin configuration that applies to all deploy
11        targets here
12    };
13
14    if (deployTarget === 'development') {
15        ENV.build.environment = 'development';
16        // configure other plugins for development deploy target here
17    }
18
19    if (deployTarget === 'staging') {
20        ENV.build.environment = 'production';
21        // configure other plugins for staging deploy target here
22    }
23
24    if (deployTarget === 'production') {
25        ENV.build.environment = 'production';
26        // configure other plugins for production deploy target here
+ 26        ENV.s3 = {
+ 27            accessKeyId:
process.env['AWS_PERSONAL_IAM_BALINT_ACCESS_KEY_ID'],
+ 28            secretAccessKey:
process.env['AWS_PERSONAL_IAM_BALINT_SECRET_ACCESS_KEY'],
+ 29            bucket: 'rarwe-production',
+ 30            region: 'us-east-1',
+ 31        };
+ 32        ENV['s3-index'] = {
+ 33            accessKeyId:
process.env['AWS_PERSONAL_IAM_BALINT_ACCESS_KEY_ID'],

```

```

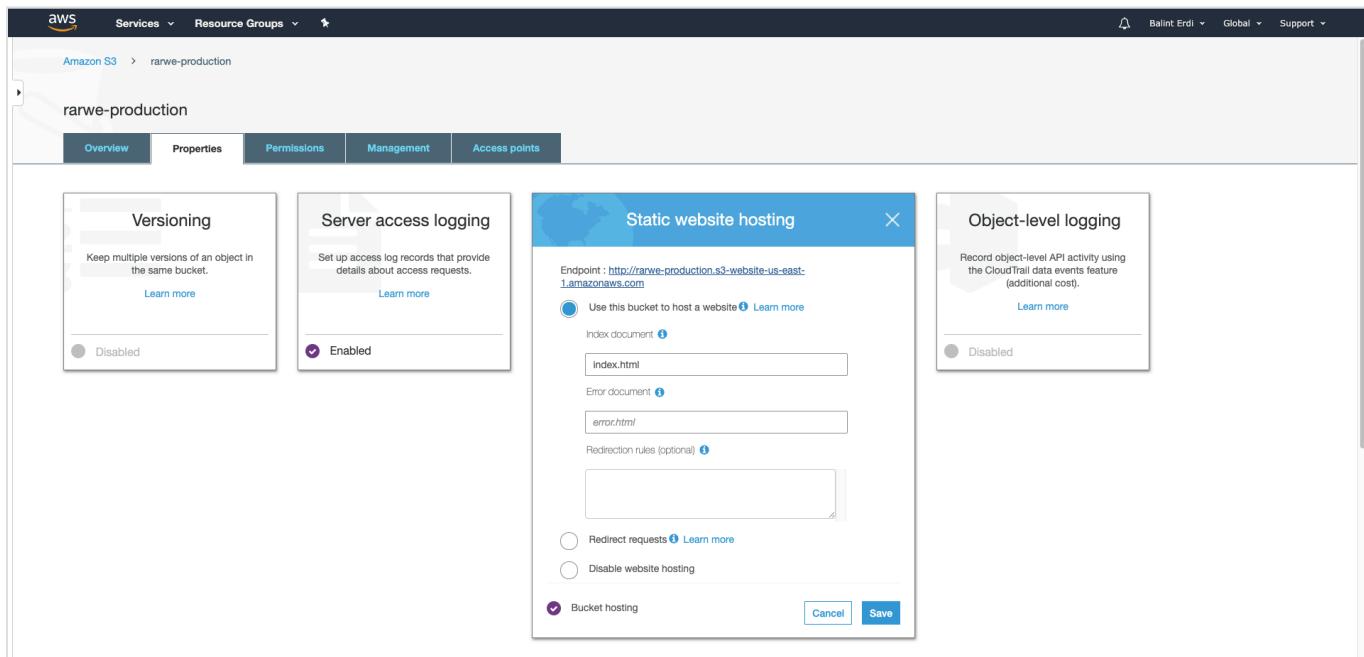
+ 34     secretAccessKey:
process.env['AWS_PERSONAL_IAM_BALINT_SECRET_ACCESS_KEY'],
+ 35     bucket: 'rarwe-production',
+ 36     region: 'us-east-1',
+ 37   };
38 }
39
40   // Note: if you need to build some configuration asynchronously, you
can return
41   // a promise that resolves with the ENV object instead of returning
the
42   // ENV object synchronously.
43   return ENV;
44 }

```

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.

The screenshot shows the AWS IAM User Summary page for a user named 'balint'. The left sidebar shows navigation options like Dashboard, Access management, Groups, Roles, Policies, Identity providers, Account settings, and more. The main area displays the user's ARN, Path, and Creation time. Below this is a 'Permissions' tab, which is selected, showing four policies applied: 'AmazonS3FullAccess' (highlighted with a red box), 'CloudFrontFullAccess', and three others that are partially visible. There are tabs for 'Groups (1)', 'Tags', 'Security credentials', and 'Access Advisor'.

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:



We now stand ready to deploy the app in production:

```
1 $ ember deploy production
2
3 Deploying [====? ] 100% [plugin: s3-index -> fetchRevisions]
4 ember deploy production 59.74s user 6.80s system 75% cpu 1:28.49 total
```

We are not done yet, as we've only deployed a new version but haven't activated it. The idea behind this is that the process of uploading assets and activating a certain version of the app is decoupled. This enables richer deployment scenarios – like A/B testing.

The `deploy:list` command lists the deployed versions:

```
1 $ ember deploy:list production
2      timestamp          | revision
3 =====
4 > 2020/06/05 16:28:56 | 00cb558a4e072db36c4d40fe607abf3e
5   2020/06/05 16:06:05 | a83fba9e6eaab651fbedd54f315bda00
```

Armed with the revision number, we can use `deploy:activate` to make the new version the active one:

```
1 $ ember deploy:activate production --revision  
00cb558a4e072db36c4d40fe607abf3e  
2 - ✓ index.html:00cb558a4e072db36c4d40fe607abf3e => index.html
```

Alternatively, we can do the deployment+activation phases in one go, passing `--activate` to the `deploy` command, as `ember deploy production --activate`.

With that, our app is running on Amazon's infrastructure.

Related hits

- [Surge](#)
- [Heroku](#)
- [Ember CLI Deploy](#)

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 are 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 first journey 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.

Wish you all the best, Balint

Appendix

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.

ES2015 stands for ECMAScript2015, a specification that introduced a whole slew of things that made JavaScript more powerful – and a lot more fun to code in. One of the included features in ES2015 is the module system.

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`:

```
1 // lib/fibonacci.js
2
3 export default function fibonacci(n) {
4 }
```

js

When another module needs what the fibonacci module exports, it accesses it via an import statement:

```
1 // tests/fibonacci-test.js
2 import fib from 'math/fibonacci';
3
4 console.log("Did I get this right? " + (fib(3) === fib(1) + fib(2)) ?
5   "YES!" :
6   "Nope");
```

js

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/fibonacci-test.js` file.

You can also export several things from the same module by defining 'named exports.' In this case, you omit the `default` designation and just use the name of the thing you want to export:

```
1 // lib/string-functions.js
2 export function capitalize(...) {
3 }
4
5 export function dasherize(...) {
6 }
```

js

Importing named exports is done via using `{}` and optionally reassigning the exported property name as a different variable by using the `as` keyword:

```
1 import { capitalize as cap, dasherize } from 'string-functions';
2
3 cap('them crooked vultures');
4 dasherize('Long Distance Calling');
```

js

Curly braces component invocation

For a long time, up until 3.4, components could not be invoked using the angle-bracket invocation syntax we used throughout the book (like `<StarRating ...>`). Instead, curly braces needed to be used with the component's name in its dasherized form.

Invoking the star-rating component with curly braces (aka. mustaches) would happen as follows:

```
{ {star-rating class="fr" rating=song.rating} }
```

HBS

Also, components that had to be callable with curly braces (so all components prior to 3.4) needed to have a dash in their name.

So why would someone use the curly braces invocation? I can think of two reasons:

1. You have an Ember app before 3.4.
2. You want to use positional parameters when invoking the component.

Positional parameters are passed as just values, not as a value assigned to a key. An example is the curly invocation of the `link-to` component we've been using throughout the app:

```
1 {{#link-to "bands.band.songs" band.id class="mt-4"} }  
2 (...)  
3 {{/link-to}}
```

HBS

Here, `"bands.band.songs"` and `band` are both positional parameters as they are not part of the params hash. Contrast this with

```
<LinkTo @route="bands.band.songs" @model={{band.id}} class="mt-4">
```

HBS

, where both parameters are named ones, `@route` and `@model`.

Angle-bracket invocation feels natural as it has the same syntax as html tags but it doesn't support positional parameters. I haven't personally found that to be a nuisance, though.

Presentational vs. Container components

Presentational components are dumb (and they are also named as such) in the good sense when talking about flexibility and reusability: they assume very little about the context they are used in and can thus be easily used in different contexts and even in different applications. They define "private" properties, they are configured via passed-in arguments and they can only change their surroundings via actions (in Ember). A widget to rate items by clicking on stars or a fancy drop-down menu are good examples for presentational components.

Container components are smart: they assume more about the context they are invoked in and can act as autonomous agents, changing the state of the application directly (for example, firing an ajax request to create a new band on the back-end). This reduces their reusability on one hand, but on the other hand they fit better in our application since they can be highly context specific. Using container components we write less and more readable code, with fewer levels of abstraction. A good example for when to reach for container components is a user form: we could pass in all the validators and save the user via an action that the user form calls, but it's way easier to just pass in the user object and have the form know about the user's properties and validations and save the user itself.

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, we get a serious advantage.

To see this, let's assume Ember did allow function calls in templates, and we had `{ {#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 always calling `stars()` and rendering the result in the DOM. If `stars()` is slow, we incur a performance hit, perhaps unnecessarily if the result is unchanged.

Contrast that with using properties (or helpers) and relying on Ember's tracking mechanism to detect changes and trigger re-renders. There is no guessing involved, and we get performant updates without

having to think about what changes when and which pieces need re-rendering.

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 [Components chapter](#) that the `StarRating` component could be used in two forms: block and non-block. When used in its block form, `yield` defines the slot where the calling context can render its content.

To give an example, a form component that focuses on the first text input field when rendered would just have a `{yield}` as its template and could be used something like this:

```
1 <FocusingForm @onSubmit=(action "register") {}>
2   <label for="email">Email</label>
3   <input type="text" name="email" value={{user.email}}>
4   {{!-- (...) --}}
5 </FocusingForm>
```

HBS

You can think of `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 use `yield` to account for the differences in markup.

async functions

Async functions were added to JavaScript as an ES2017 feature and have wide adoption in browsers (of the main browsers only IE 11 doesn't support them natively) and Node (since version 7.6).

They work on top of promises and provide many advantages over them, not the least of which is better readability and debugging. Most of the improved readability comes from using the `await` keyword used to

pause the execution of any async function until the async operation resolves.

They also wrap their return value in a promise so you don't have to do the wrapping yourself.

Let's see (an enhanced version of) the example of the `saveBand` action from the app. Note that the exact details of the `errorTracker` service and the `extractError` function are not important in this context; the point is to show how async functions provide a clearer way of writing code.

Using an async function:

```
1  async saveBand() {  
2      try {  
3          let band = await this.catalog.create('band', { name: this.name });  
4          this.confirmedLeave = true;  
5          this.router.transitionTo('bands.band.songs', band.id);  
6      } catch(e) {  
7          this.errorTracker.send(e);  
8          this.set('errorMessage', extractError(e));  
9      }  
10 }
```

js

Using promises:

```
1  saveBand() {  
2      this.catalog.create('band', { name: this.name }).then(band => {  
3          this.confirmedLeave = true;  
4          this.router.transitionTo('bands.band.songs', band.id);  
5      })  
6      .catch(e => {  
7          this.errorTracker.send(e);  
8          this.set('errorMessage', extractError(e));  
9      })  
10 }
```

js

Even in this simple example, I find the async function reads better, but imagine if you have a promise-chain with several levels, or promises needing each other's resolved values.