Ember.js IN {{ACTION}}

Joachim Haagen Skeie

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Ember.js in Action
version 10

For more information on this and other Manning titles go to
www.manning.com

brief contents

0 Preface

0.1 Preface

Since 2006, I have worked with the development of web applications in one form or another. I started out writing a web application for Norway's largest retailer, which was based on JavaServer Pages, and later migrated towards JavaServer Faces. At the time these technologies where great, and they served the purpose that they were intended to solve. Back then, before AJAX became widely used, the request-response cycle of the HTTP protocol demanded that most of the logic was put on the server-side and that the server would deliver complete markup, scripts and stylesheets to the browser on every request.

Even though these server-side approaches to writing web application did their job, there were always issues involved whenever state was a concern. Because the server was required to do the bookkeeping for all of the logged in users, managing state quickly becomes an issue. How do you deal with users opening multiple tabs with your application and switching between them, or how do you persist your session data when you want to scale the service across multiple (virtual) machines?

As I started working on the open source project Montric (then named EurekaJ), I quickly decided that in order to be able to scale the application out horizontally without requiring a separate session-cache, I needed to invest in learning one of the JavaScript frameworks that had started to gain momentum and popularity.

I assessed multiple frameworks that were available at the time, and built prototypes in both Cappuccino (http://www.cappuccino-project.org) and SproutCore (http://sproutcore.com). Even though Cappuccino had more complete tooling and promised a very detailed and good looking user interface for my application, my choice fell on SproutCore, as it was easier to leverage my existing skills, while also promising an easier integration with third party libraries. SproutCore also offered powerful views that acts as components that can be combined into a fully functional web application.

Coming from a component-based server-side framework world, these features made me feel at home with SproutCore. But this initial delight had its drawbacks eventually as I quickly found out that integrating SproutCore with third party libraries was not as easy as I had first through it would be.

Then, as the team behind SproutCore was acquired, and the momentum of the framework slowed down, a shift within the SproutCore community started. Work on SproutCore 2.0 had been well underway, but the gap between the original SproutCore and what was going to become SproutCore 2.0 grew.

As a result, Ember.js was born as a fork of SproutCore. Ember.js promised a framework that was thoroughly rooted in the technologies that drive our web experience, enabling you – as a developer – to use the skillset that you already have with developing JavaScript applications. Ember.js does not abstract or hide away the details of your JavaScript, HTML or CSS code, and instead embraces these technologies in order to lift them up into the 21st century.

Needless to say, I followed along with Ember.js and decided to rewrite EurekaJ's frontent to Ember.js, in the process also renaming the project to Montric. I have been on the Ember.js rollercoaster ever since. Pre 1.0.0, being part of the Ember.js community had its highs and its lows. With constant changing APIs and concepts being rethought and revisited on what felt like a weekly basis, the lows have all mostly been iron out, only leaving the highs. Deciding to write a comprehensive book about Ember.js in its pre 1.0.0 days have undoubtedly exaggerated the lows for my part.

As I am writing this Ember.js 1.2.0 is out, the API have stabilized completely and the project is healthy and growing. Ember.js really have become a truly awesome framework that lets you build applications that push the envelope on what you are able to build for the web.

Crafting and writing this book have been an extremely humbling, difficult and educational experience. Through the process I have tried to document as much about Ember.js as possible, while trying to keep the examples as real and down-to-earth as possible. I chose to base the second and third part of the book on the source code from Montric. This led to the occasional difficulty in isolating good examples, but I also think that this gives the book a bit more depth while it also allows you to see how Ember.js has grown from its 1.0.0 version, as the code that the books examples are built on will undoubtedly have been updated and changed somewhat.

0.2 Acknowledgements

This book is the most comprehensive and cohesive writing I have done to date, and it has been an experience with a high learning curve. I wish to thank the Manning team for being willing to write a book about Ember.js, and for pushing to get the book realized as the finished story that you're about to embark on. I would especially like to thank Susanna Kline, the books development editor for putting up with my many missed deadlines and tireless Skype questions, always focusing on giving great feedback on what needed improvements along the way. I also would like to thank the team of copy editors, Rosalie Donlon, Llianna Vlasiuk and

Teresa Wilson who caught and fixed an embarrassing amount of spelling and grammar mistakes throughout the text.

The reviewers also made sure that the book kept on point and relevant during the various stages of the book, and I'd like to extend my gratitude for their work along the way.

I wish to extend a special huge thank you to my beautiful wife, Lene, and my two amazing children, Nicolas and Aurora. Lene's support and understanding have been incredibly important during the writing of the book, especially during times where the book consumed a lot of my spare time in evenings and weekends. Knowing that the family life is safe, secure and happy is of utmost importance when deciding to prioritize your spare time elsewhere.

0.3 About the Book

Ember.js is the most ambitious web application framework for JavaScript. With the release of the final 1.0.0 version, after just under two years of development, the APIs have stabilized and the project has moved on steadily quickly reaching version 1.1.0 and 1.2.0.

Ember.js exists because writing large ambitious web application is hard work and Ember.js' creators wanted to build a framework that simplifies and standardize the way we write applications for the web. This book aims to go through the features and highlights of the framework while keeping the text driven by real world examples.

0.3.1 Roadmap

The book is divided into three major parts

- Part 1 – The Ember.js Fundamentals
- Part 2 – Building Ambitious Web applications for the real works
- Part 3 - Advanced Ember.js Topics

Part 1 starts off with simple, self contained examples that introduce you to Ember.js, its core features and what Ember.js expects from your application.

- Chapter 1 introduces you to Ember.js, and explains where the project comes from as well as where it fits into the world of web applications. Here, you will learn the fundamental concepts and terminology used in Ember.js.
- Chapter 2 builds from what we have learned in chapter one, but takes more time in explaining the core features of Ember.js. In this chapter you will learn about bindings, computed properties, observers and the Ember.js object model.
- Chapter 3 is dedicated to Ember Router – the backbone and glue that holds your whole application together.
- Chapter 4 talks about Handlebars.js, the template library of choice for Ember.js applications. Here you will learn the features Handlebars.js brings to the table for Ember.js applications, as well as the Ember.js specific addons that Ember.js adds to Handlebars.js.

Part 2 starts off by introducing the case study, Montric, that will be used for the majority of remaining chapters. This part delves into the harder parts of your web application

development; how you can interact with the server side efficiently both with and without Ember Data, writing custom components and testing your Ember.js applications.

- Chapter 5 dives right into how you can leverage the third-party Ember Data library in order to communicate with the server side. Based on Ember Data Beta 2, this chapter goes through what Ember Data expects from your server-side API and your Ember.js application, before showing how Ember Data can be customized to fit your existing server-side API.
- Chapter 6 shows you how you can interact with the server-side without relying on a framework to help you. This chapter shows you how you can build a complete CRUD data-layer from scratch.
- Chapter 7 is all about custom components, a feature added late to Ember.js. Using Ember.js Components you are able to build atomic, self-contained components that can be reused either on their own, or as building blocks of more complex components
- Chapter 8 is dedicated to showing you how you can test your Ember.js application. It leverages QUnit and PhantomJS in order to build up a viable testing solution.

Part 3 moves into the more obscure parts of Ember.js, and talks about leveraging other services and tools to facilitate your application development and increase your understanding of Ember.js

- Chapter 9 goes through how you can build in authentication and authorization using a third party authentication system. This chapter specifically implements Mozilla Persona, which is an open source solution from Mozilla.
- Chapter 10 eases you into the Ember run loop, called backburner.js, the background motor that drives your Ember.js application and ensures that your applications views are always kept up-to-date while keeping the performance high.
- Chapter 11 teaches you how you can structure your Ember.js application as your source code grows, and how you can build, assemble and package your application, making it ready for deployment.

WHO SHOULD READ THIS BOOK?

This book is written to help you become familiar and productive with Ember.js. Depending on your background, developing JavaScript applications with frameworks like Ember.js have a steep learning curve. This book will help you learn the concepts of Ember.js quickly and make you familiar with the Ember.js terminology and application structure. This book is both approachable to newcomers to Ember.js, as well as developers with previous Ember.js experience.

As a prerequisite, this book expects you to be familiar with JavaScript as a language, as well as some familiarity with jQuery.

CODE CONVENTIONS AND DOWNLOADS

This book includes a large amount of code snippets and source code. The source code for the first part of the book is available via the books GitHub Pages, or as a downloadable .zip file

from the books website at Manning.com/skeie. For part 2 and 3 of the book, the source code is based upon read world projects, and as such the code is available on GitHub. As these are living projects, the source code will invariably have changed since this text was written. To account for this, the links below are direct links that will bring you to the source code as it was during the writing of this book.

- Chapters 1 and 2: https://github.com/joachimhs/Ember.js-in-Action-Source/tree/master/chapter1/notes
- Chapter 3: https://github.com/joachimhs/Ember.js-in-Action-Source/tree/master/chapter3/blog
- Chapters 5, 7, 8, 9 and 11: https://github.com/joachimhs/Montric/tree/Ember.js-in-Action-Branch
- Chapter 6: https://github.com/joachimhs/EmberFestWebsite/tree/Ember.js-in-Action-branch

0.3.2 Author Online

The purchase of *Ember.js in Action* includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to http://www.manning.com/skeie. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The Author Online forum and the archives of previous discussions will be accessible from the publisher's web site as long as the book is in print.

0.3.3 About the Author

Joachim Haagen Skeie is self-employed though his company Haagen Software AS. He spends his time working as a freelance consultant, a course instructor teaching Ember.js and RaspberryPi, while working on launching the products Montric – and open source application performance monitor - and Contentice – an open source CMS API used to host rich web applicaitons. Both Montric and Contentice is based on Ember.js on the front end, with Java on the backed.

Joachim has been working on the development of web applications, big and small, since 2006, primarily focusing on Java and Ember.js. Joachim lives in Oslo, Norway with his wife and children.

0 Part Openers

0.1 Part 1

Ember.js is a JavaScript MVC framework that helps you organize and structure the source code for large web applications. In comparison to other popular JavaScript application frameworks it delivers a more complete MVC pattern, while also including features that will help you build applications for the web of tomorrow. It is also one of the more opinionated frameworks available, relying heavily on conventions over configuration to help you structure your application.

The large amount of features coupled with the conventions that Ember.js expects from your application, means that there is a rather steep learning curve. The first part of the book, comprising the first four chapters, are meant to ease our into the mindset of Ember.js application development, while ensuring that we build something useful right from the get go.

The first two chapters focus their attention on the core features that Ember.js brings to the table, while chapter 3 focus its attention on Ember Router and chapter 4 focus on the template library of choice for Ember.js developers – Handlebars.js.

0.2 Part 2

In part 1, we guided you through the core Ember.js features and functionality, while focusing on getting familiar over the conventions that is used throughout Ember.js applications. In part 2, we will shift our focus slightly and explore how we can make our Ember.js applications come alive.

Part 2 will introduce the case study that will form the basis for most of the books sample code, Montric. Montric is an open source application performance-monitoring tool, where the frontend is written in Ember.js. The backend is Java-based, running on top of a horizontal scalable database, Riak.

We start out by going through how we can integrate server-side communication via Ember Router, first with Ember Data beta 2 in chapter 5, before we move on to rolling our own model-layer in chapter 6.

Once we have sorted out the server side communication options, we move on to another important and core feature of Ember.js, custom components. Although self-contained components were added late in the development towards Ember.js 1.0.0, it is a much needed and very powerful feature. Chapter 7 goes through Ember.js' approach to self-contained components by first introducing a few simple components before combining these components into new, more complex components.

Before we leave part 2 of the book, we will take a deep dive and look at how we can test out Ember.js application. Chapter 8 shows how we can utilize QUnit and Phantom.js in order to build up a complete testing strategy.

0.3 Part 3

The final part of Ember.js in Action will show you some of Ember.js' more advanced features. We start out by showing how Montric is integrated with Mozilla Persona, for user authentication. Mozilla Persona is a complete, open source and independent user authentication and authorization platform. Even though you wont be using Mozilla Persona, you will still learn how you can integrate Ember.js with your third party authentication service of choice.

Underneath the surface, Ember.js hides an incredibly powerful engine that ensures that your applications views and templates are always updated whenever the data within your application change. This engine, often referred to as the Ember Run Loop, is called Backburner.js. While you can develop Ember.js application without ever having to understand backburner.js, knowing the features that it provides and how you can utilize its functionality within your own application will enable you to build an application with better performance and more maintainable code.

The final chapter, chapter 11, takes you through a complete assembly process for your Ember.js application. This is the final step that you need to overcome in order to transition your application from development to production. Implementing a sane directory structure and a proper oils build tool chain will also help you immensely throughout the development process itself.

1

Powering your next ambitious web application

This chapter covers

- A brief history of the single-page web application (SPA)
- An introduction to Ember.js
- What Ember.js can provide for you, as a web developer
- Your first Ember.js application

This chapter introduces the Ember.js application framework and touches on many of the features and the technologies in the Ember.js ecosystem. Most of these topics are covered in more detail in the book. This chapter gives a quick overview of what an Ember.js application might look like and what strengths you get from basing your application on Ember.js.

It also includes an overview of the building blocks of an Ember.js application and briefly touches on the different aspects that form the Ember.js framework. If you initially find any code presented here confusing or otherwise hard to understand, don't worry! All aspects of the source code development are explored in detail, every step of the way, in the book.

Ember.js comes with a steep learning curve if you're used to writing server-side generated web applications. The code examples presented in this chapter and the Notes application go through the many different concepts of structuring an Ember.js application.

The structure of Ember.js itself is based on a set of microlibraries. Each chapter of this book starts with a diagram that shows these microlibraries and highlights the ones that the current chapter discusses. We'll touch briefly on many of the microlibraries in Ember.js in this chapter as shown in figure 1.1.

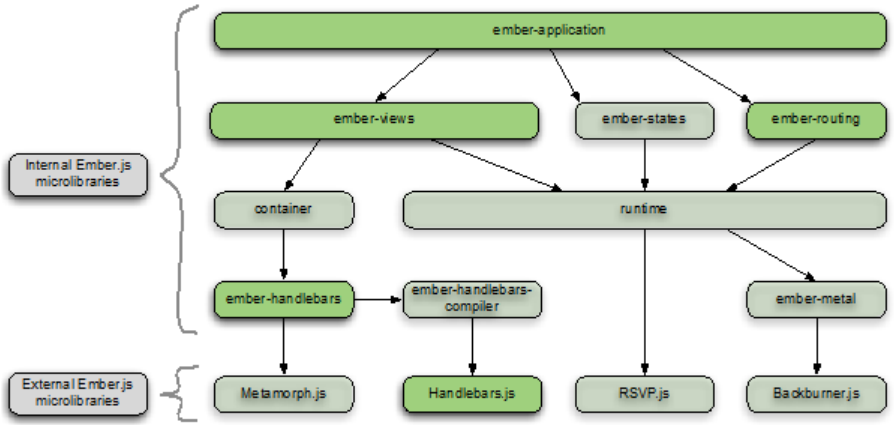


Figure 1.1 The internal structure of Ember.js

Depending on what you want to build, Ember.js may be the framework for you.

1.1 Who is Ember.js for?

Websites that serve content based on the traditional HTTP request-response lifecycle, such as the websites for the *New York Times* or Apple Inc., render most of the HTML, CSS, and JavaScript on the server. As shown in figure 1.2 (at left), for each request, the server generates a new, complete copy of the website's markup.

At the other end of the spectrum are rich Internet applications (RIAs), such as Google Maps, Trello, and GitHub. These websites aim to define new application types and rival native installed applications, and they render much of their content at the client side. As shown in figure 1.2 (at right), in response to the first request, the server sends a complete application (HTML, CSS, and JavaScript) only once. Subsequent requests return only the data required to display the next "page" in the application.

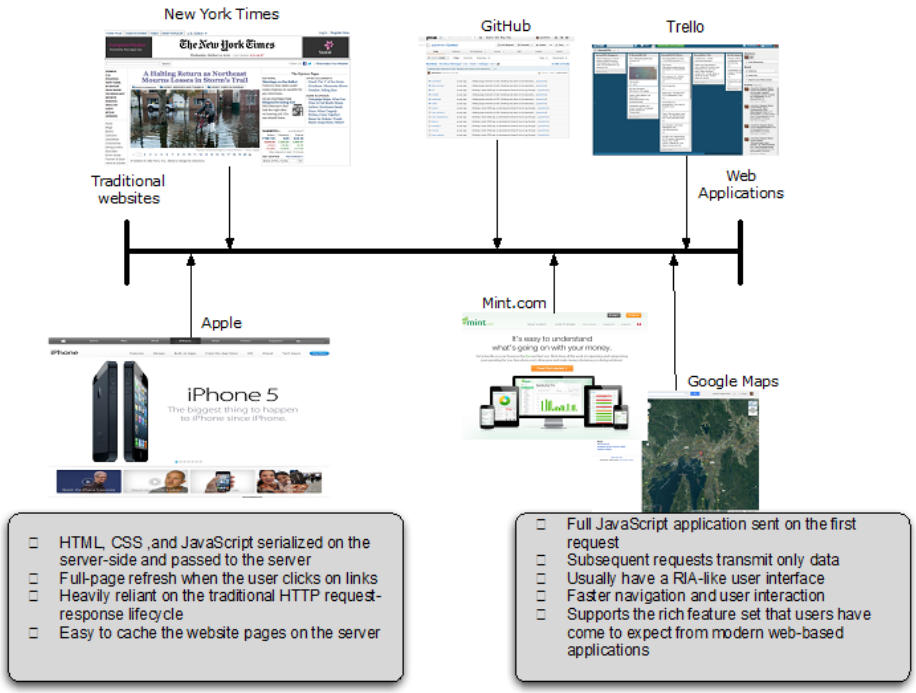


Figure 1.2 The Ember.js framework works with a variety of web applications.

Strengths and weaknesses are at both ends of the spectrum. The pages toward the left of the scale are easier to cache on the server, but they tend to rely on the request-response cycle and full-page refreshes in response to user actions.

The applications toward the other end of the scale typically have richer user interfaces, deliver a better user experience, and resemble and behave like familiar native applications, but they're more complex and require more of the browser software in terms of computing power, features, and stability.

Single-page applications (SPAs) have become more common. They're based on the server serving the complete application only once in response to the first request. Subsequent requests then transmit only updated data to receive any new or fresh data required to respond to user actions. RIAs, and SPAs in particular, feel more like native, installable applications because they have a more responsive user interface with few or no complete page refreshes. Within this domain, Ember.js aims to be a framework that provides the best solutions for web application developers and pushes the envelope of what's possible to develop for the web. As such, Ember.js fits well with applications that require long-lived user sessions, have rich user interfaces, and are based on standard web technologies.

If you build applications that towards the right in figure 1.2, then Ember.js is for you. On top of that, Ember.js makes you stop and think about how you want to structure your

application. It provides powerful tools for you to build rich web-based applications that stretch the limits of what's possible today while providing a rich set of features that enables you to build truly ambitious web applications.

Before we get started with developing an Ember.js application, it's useful to discuss why we have frameworks like Ember.js in the first place, in order to explain the problems that Ember.js promise to solve.

1.2 From static pages to Ajax to full-featured web apps

From the introduction of the World Wide Web (WWW or W3) in the mid-1990s, up until Ajax arrived in the mid-2000s, most websites were static in nature. The server responded to any HTTP page requests with a single HTTP response, which contained the complete HTML, CSS, and JavaScript required to display a complete page, as depicted in figure 1.3 (at left).

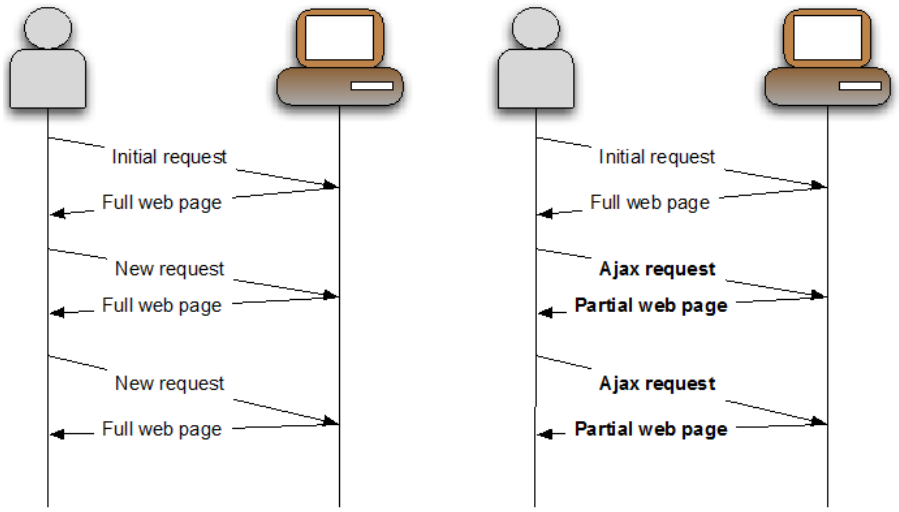


Figure 1.3 The structure of the early web (left) versus the promise of Ajax (right)

While a large amount of websites still rely on the full page refresh approach shown to the left in figure 1.3, more and more web sites are building dynamic content into their web sites. Today though, users are exepcting web sites to act and feel like applications with no page refreshes occuring.

1.2.1 The rise of asynchronous web applications

Along with the introduction of the asynchronous call came the ability to send specific parts of the website for each response. Dedicated JavaScript code would receive this response on the client side and replace the contents of HTML elements throughout the website, as shown in figure 1.3 (at right). As nice as this seems, this approach comes with a gigantic caveat.

It's trivial to implement a service on the server side that, given an element type, renders the new contents of that element and returns it back to the browser in an atomic manner. If that were representative of what rich web application users wanted, it would've solved the problem. The issue is that users rarely only want to update a single element at any one time.

For example, when you browse an online store, you navigate to or search for items to add to your shopping cart. When you add an item to the cart, you reasonably expect the item quantity and the shopping cart summary to update simultaneously. This lets you know the total number of items as well as the total price of the items in your shopping cart.

Because it is difficult to define a set of general rules that define which elements that the server will include in each of the AJAX responses, most server-side frameworks revert to sending the complete web page back to the client. The client, on the other hand, will know which elements to replace and swap out the correct HTML elements.

As you can guess, this approach is rather inefficient, while it also significantly increase the number of HTTP requests that the client sends to the server. This is where the power of Ember.js comes into play.

As a developer, you probably understand the issues with the model presented in figure 1.3, in which the server side returns the updated markup for single elements on the page. To update multiple elements, you need to take one of the following approaches:

- Require the browser to fire off additional Ajax requests, one for each element that updates on the website.
- Be aware of—on both the client and server sides—which elements must update for every action a user performs in your application

The first option multiplies the number of HTTP calls to your server; the second option requires you to maintain client state on *both* the client and the server.

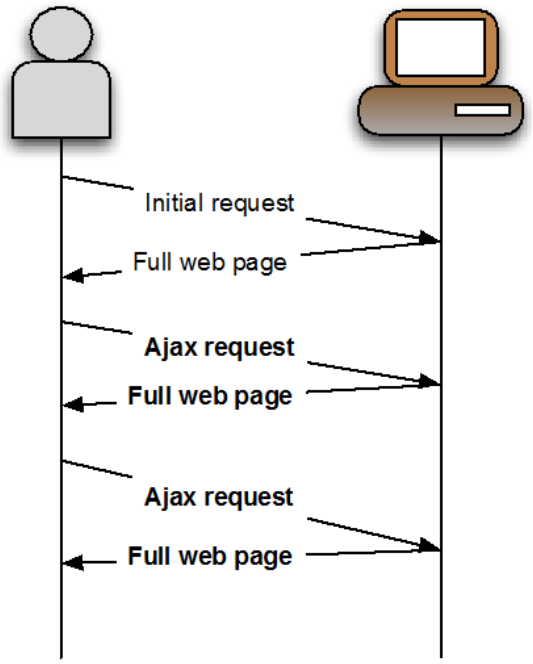


Figure 1.4 The structure of a server-side framework

As you can see, we have now significantly increased the number of HTTP requests that the client issues against the server. But at the same time, we have not decreased the amount of work that the server needs to do for each of these requests. Don't get me wrong, they did support partial page updates by way of replacing elements based on the element identifiers and cherry-picking these elements from the complete markup returned from the server. If you're thinking this is a waste of both server- and client-side resources, you're absolutely right. Figure 1.4 shows this structure.

Ideally, what we want to do, is to server the application only once. After the full application is loaded, we only want to submit request for data from the client. This brings us to the model that Ember.js employs.

1.2.2 Moving toward the Ember.js model

These days, websites rely less on passing markup between the server and the client, and more on passing data. It's in this realm that Ember.js comes into play as shown in figure 1.5.

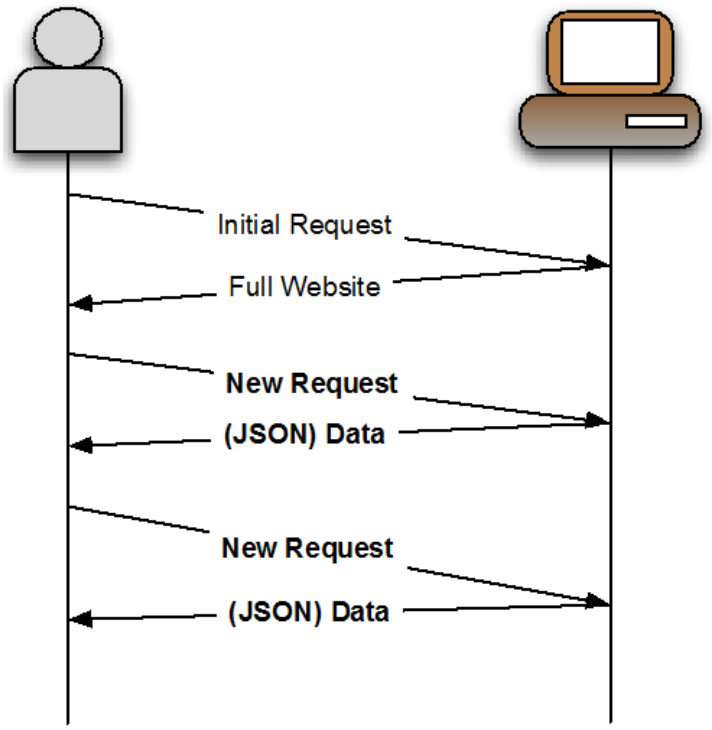


Figure 1.5 A modern web application model

In figure 1.5, the user receives the full website exactly once, upon the initial request. This leads to two things: increased initial load time but significantly improved performance for each subsequent user action.

In fact, the model presented in figure 1.5 is similar to the traditional client/server model dating back to the 1970s but with two important distinctions: the initial request serves as a highly viable and customizable distribution channel for the client application, while also ensuring that all clients adhere to a common set of web standards (HTML, CSS, JavaScript, and others).

Along with the client/server model, the business logic involving user interaction, the GUI, as well as performance logic has shifted off the server and onto the client. This shift might pose a security issue for specialized deployments, but generally, as long as the server controls who has access to the data being requested, the security concerns can be delegated to the server where they belong. It also means that the client and the server can get back to doing what they do best—serving the user interface and the data, respectively.

Now that we have an understanding of which types of web applications Ember.js is created to build, lets delve into the details of Ember.js.

1.3 Overview of Ember.js

Ember.js started its life as the second version of the SproutCore framework. But while working on version 2.0 of SproutCore, it became clear to the SproutCore team members that the underlying structure of the framework needed a radical change if they were to meet their goal of building an easy-to-use framework that applied to a wide range of target web applications but was still small.

> **What Is SproutCore?**
>
> If you're not familiar with SproutCore, it's a framework developed with a highly component-oriented programming model. SproutCore borrowed most of its concepts from Apple's Cocoa, and Apple has written some of its web applications (MobileMe and iCloud) on top of SproutCore. Apple also contributed a large chunk of code back to the SproutCore project. In November 2011, Facebook acquired the team responsible for maintaining SproutCore.

In the end, part of the core team decided to make these changes in a new framework separate from SproutCore's origins.

That being said, Ember.js does borrow much of its underlying structure and design from SproutCore. But where SproutCore tries to be an end-to-end solution for building desktop-like applications by hiding most of the implementation details from its users, Ember.js does what it can to make it clear to users that HTML and CSS are at the core of its development stack.

Ember.js' strengths lie in its ability to enable you to structure your JavaScript source code in a consistent and reliable pattern while keeping the HTML and CSS easily visible. In addition, not having to rely on specific build tools to develop, build, and assemble your application gives you more options and control when it comes to how you want to structure your development. And when the time comes to assemble and package your application, many reliable tools are available. In chapter 11, you'll learn about a few of the available packaging options.

You must be eager to get started with Ember.js by now, so let's quickly describe what Ember.js is and what parts make up and Ember.js application, before moving on to creating our first Ember.js application.

1.3.1 What is Ember.js?

According to the Ember.js website,[1] Ember.js is a framework that enables you to build "ambitious" web applications. That term "ambitious" can mean different things to different people, but as a general rule, Ember.js aims to help you push the envelope of what you're able to develop for the web, while ensuring that your application source code remains structured and sane.

[1] http://emberjs.com/

Ember.js achieves this goal by structuring your application into logical abstraction layers and forcing the development model to be as object oriented as possible. At its core Ember.js has built-in support for the following features:

- *Bindings*—Enables the changes to one variable to propagate into another variable and vice versa
- *Computed properties*—Enables you to mark functions as properties that automatically update along with the properties they rely on
- *Automatic updated templates*—Ensures that your GUI stays up to date whenever changes occur in the underlying data

Combine these features with a strong and well-planned Model-View-Controller (MVC) architecture and you've got a framework that delivers on its promise.

1.3.2 The parts that make up an Ember.js application

If you've spent most of your time developing web applications with server-side generated markup and JavaScript, Ember.js—and, indeed, most of the new JavaScript frameworks—has a completely different structure from what you're used to.

Ember.js includes a complete MVC implementation, which enriches both the controller and the view layers. We will discuss more about the MVC implementation as we progress through the chapters.

- *Controller layer*—Built with a combination of routes and controllers
- *View layer*—Built with a combination of templates and views

> **NOTE** Ember Data, which you'll learn about in chapter 5, enriches the model layer of Ember.js.

When you build an Ember.js application, you separate the concerns of your application in a consistent and structured manner. You also spend a decent amount of time thinking about where to best place your application logic. Even though this approach does take some careful consideration before you delve into the code, your end product is better structured and, as a result, easier to maintain.

Most likely, you'll opt to follow the guidelines and standard conventions of Ember.js, but in some cases you may need to spend some time off the beaten track to implement the more intricate features of your application.

As you can see in figure 1.6, Ember.js introduces extra concepts at each of the layers in the standard MVC model. These concepts are explored in detail in the first five chapters of this book.

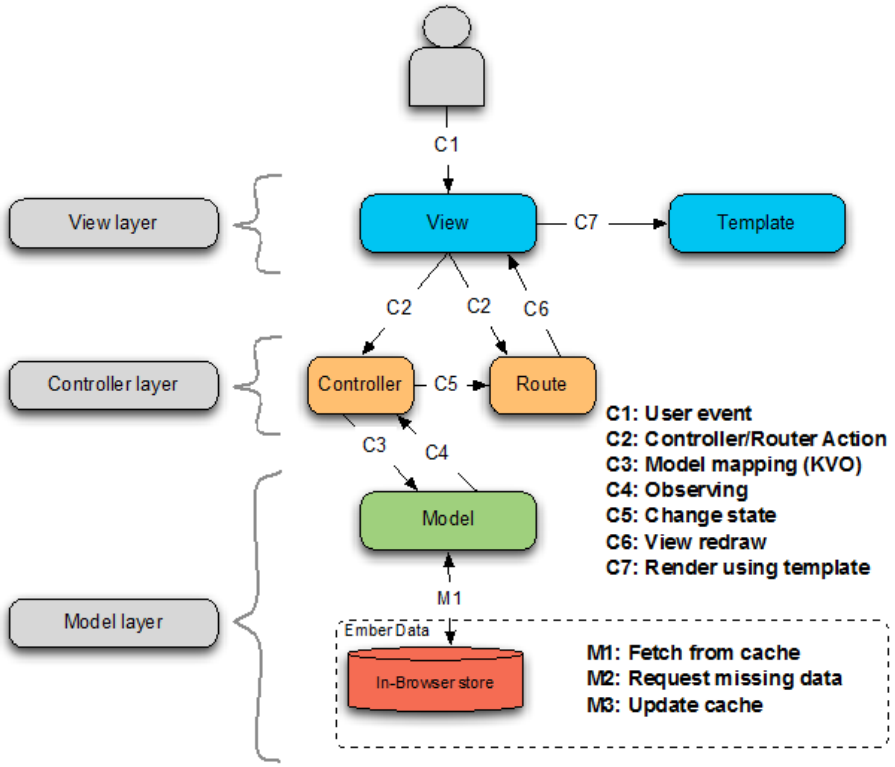


Figure 1.6 The parts that make up Ember.js and how they fit in with the MVC pattern

With that figure in mind, let's take a closer look at each of the MVC components in detail.

MODELS AND EMBER DATA

At the bottom of the stack, Ember.js uses Ember Data to simplify the application and provide it with the rich data-model features that you need to build truly rich web-based applications. Ember Data represents one possible implementation that you can employ when it comes to communicating with the server. Other libraries do exist for this functionality and you can also write or bring your own client-to-server communication layer. Ember Data is discussed in detail in chapter 5, and a discussion on how to roll your own data layer is covered in chapter 6.

The model layer holds the data for the application, which is typically specified clearly through a semistrict schema. The Model layer is responsible for any server-side communication, as well as model-specific tasks such as data formatting. The view binds the GUI components against properties on the model objects via a controller.

Ember Data lives in the model layer, and you use it to define your model objects, your client-to-server API, as well as the transport protocol between your Ember.js application and the server (jQuery, XHR, WebSockets, and others).

CONTROLLERS AND EMBER ROUTER

Above the model layer is the controller layer. The controller acts mainly as a link between the models and the views. Ember.js ships with a couple of custom controllers, most notably the `Ember.ObjectController` and the `Ember.ArrayController`. Generally, you use the `ObjectController` if your controller represents a single object (like a selected `note`); you use the `ArrayController` if your controller represents an array of items (like a list of all `notes` available for the current user).

On top of this, Ember.js uses Ember Router to split your application into clearly defined logical states. Each route can have a number of subroutes, and you can use the router to navigate between the states in your application.

The Ember Router is also the mechanism that Ember.js uses to update your application's URL and listen for URL changes. When using Ember Router you model all your application's states in a hierarchical structure that resembles a state chart. Ember Router is discussed in detail in chapter 3.

VIEWS AND HANDLEBARS.JS

The view layer is responsible for drawing its elements on the screen. The views generally hold no permanent state of their own, with few exceptions. By default, each view in Ember.js has one controller as its context. It uses this controller to fetch its data and, by default, uses this controller as the target for any user actions that occur on the view.

Also by default, Ember.js uses Handlebars.js as its templating engine. Therefore, most Ember.js applications define their user interfaces via Handlebars.js templates. Each view uses one template to render its view. Handlebars.js and templates are discussed in chapter 4.

Handlebars.js

Handlebars.js is based upon Mustache, which is a logic-less template library that exists for a number of programming languages, including JavaScript. Handlebars.js adds basic logic expressions (if, if-else, each, etc.) on top of Mustache. This, along with the ability to bind your templates to properties on your views and controllers, lets you build templates that are well structured, specific and hand-tailored to your Ember.js application.

Ember.js ships with default views for some of the basic HTML5 elements, and it's generally a good practice to use these views when you're in need of simple elements. You can easily create your own custom views that either extend or combine the standard Ember.js views to build complex elements in your web application.

Now that we have a clear understanding of the parts that make up an Ember.js application, its time to start writing our very first app.

1.4 Your first Ember.js application: Notes

The source code for the Notes application weighs in at about 200 lines of code and 130 lines of CSS, including the templates and JavaScript source code. You should be able to develop and run this application on any Windows-, Mac- or Linux-based platform using only a text editor.

> **TIP** I generally use WebStorm from JetBrains to write JavaScript-based applications, but this is not a requirement.

To get an idea of what to expect from an Ember.js application, let's dive in and write a simple web application that manages notes. The application has the following features:

- *Add a new note*—The application has an area that allows users to add notes to the system.
- *Select, view, and edit a note from the system*—Available notes appear in a list at the left. Users select one note at a time to view and edit its content in a window at the right.
- *Delete an existing note*—Users can delete the selected note from the system.

A rough design of what you're building is shown in figure 1.7.

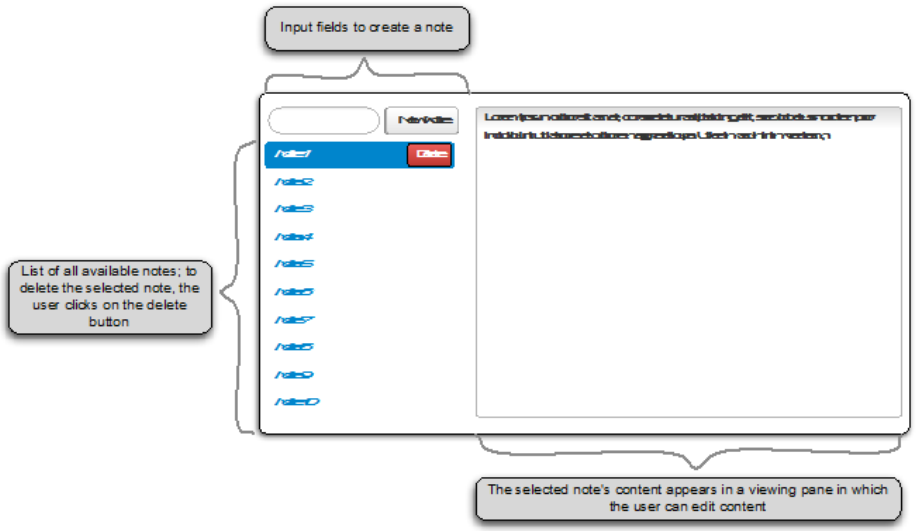


Figure 1.7 The design and layout of the Notes application

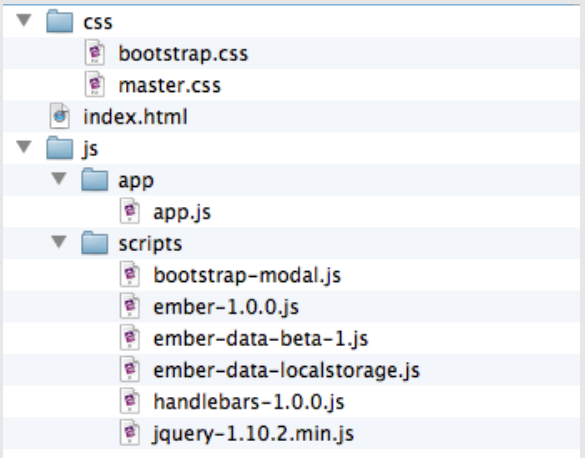
To get started, download the following libraries. The versions of each library might vary slightly, depending on the current version of Ember.js:

- Ember.js version 1.0.0

- Handlebars.js version 1.0.0
- jQuery version 1.1x
- Twitter Bootstrap CSS
- Twitter Bootstrap Modal
- Ember Data version 1.x Beta
- Ember Data Local Storage Adapter

Your options: start from scratch or get the code from GitHub

To start from scratch:
1. Create a directory on your hard drive to store all the application files.
2. Create the directory structure as shown:



To get the code from GitHub:
If you'd rather get the source code all packed up and ready to go, download or clone the Git source repository from GitHub: https://github.com/joachimhs/Ember.js-in-Action-Source/tree/master/chapter1.

After you're set up, open the index.html file.

1.4.1 Getting started with the Notes application

Wire the application files together inside your index.html file, as shown in the following listing.

Listing 1.1 Creating links in the index.html file

```
<!DOCTYPE html>                                                          #A

<html lang="en">                                                         #B
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <meta name="viewport" content="width=device-width,
        initial-scale=1.0, maximum-scale=1.0">

     <title>Ember.js Chapter 1 - Notes</title>
     <link rel="stylesheet" href="css/bootstrap.css"
        type="text/css" charset="utf-8">                                 #C
     <link rel="stylesheet" href="css/master.css"
        type="text/css" charset="utf-8">                                 #D

    <script src="js/scripts/jquery-1.10.2.min.js"
        type="text/javascript" charset="utf-8">
    </script>
    <script src="js/scripts/bootstrap-modal.js"
        type="text/javascript" charset="utf-8">
    </script>
    <script src="js/scripts/handlebars-1.0.0.js"
        type="text/javascript" charset="utf-8">
    </script>
    <script src="js/scripts/ember-1.0.0.js"
        type="text/javascript" charset="utf-8">
    </script>
    <script src="js/scripts/ember-data-beta-1.js"
        type="text/javascript" charset="utf-8">
    </script>
    <script src="js/scripts/ember-data-localstorage.js"
        type="text/javascript" charset="utf-8">
    </script>
    <script src="js/app/app.js" type="text/javascript"
        charset="utf-8">
    </script>                                                            #E

</head>
<body>

</body>
</html>
```

#A: Standard doctype declaration
#B: Standard elements to start an HTML document
#C: Links to Twitter Bootstrap CSS file
#D: Links to custom CSS file
#E: Adds application source code

The code in this listing is enough to get you started developing the Notes application.

NOTE For simplicity, you'll place all your application templates inside the index.html file. This simplifies your setup, and it's a convenient method to get started on a new Ember.js application. Once your application grows, you typically extract your templates into separate files and bring them in via build tools. Build tools will be discussed in chapter 11.

In most production-ready Ember.js applications, the code in listing 1.1 is all the code that will ever be inside the index.html file. This might be different from the web development that you're used to; it certainly was for me before I was introduced to Ember.js a few years back. Unless you specify anything else, by default the Ember.js application places its contents inside the `body` tag of your HTML document.

Nothing special is happening in the code. The document starts out by defining the `doctype` before starting the `HTML` element with the standard `HEAD` element. Inside the `HEAD` element you set the page's title, along with links to both the Twitter Bootstrap CSS as well as a separate CSS file where you'll put the custom CSS required by your Notes application. The `script` elements of the `HEAD` tag define links to the scripts that your application is dependent on; the last `script` tag links to the source code for the Notes application, which you'll develop throughout the rest of this chapter.

1.4.2 Creating a namespace and a router

In this section you'll build the first part of the Notes application with the basic web application layout in place.

> **NOTE** The source code for this section is available as app1.js in either your code source directory or online at GitHub: https://github.com/joachimhs/Ember.js-in-Action-Source/blob/master/chapter1/notes/js/app/app1.js.

The first thing any Ember.js application needs is a namespace that the application lives inside. For your Notes application this namespace is `Notes`.

After your namespace is created, you need to create a router that knows how your application is structured. Using the router is not a requirement, but as you'll see in this book, it greatly simplifies and manages the structure of your entire application. You can think of the router as the glue that holds your application in place and connects different parts of your application together.

The code required to get your Notes application up and running to serve a blank website is minimal:

```
var Notes = Ember.Application.create({                    #A
    LOG_TRANSITIONS: true,                                #B
});
```

#A: Creates a namespace for the application
#B: Enables LOG_TRANSITIONS, prints changes to console log

This code creates your Notes namespace on the first line via `Ember.Application.create()`. Any code that you write related to this application is contained in this `Notes` namespace. This keeps the code separate from any other code that you might bring in via third-party libraries or even inline inside your JavaScript file. But serving a completely blank website is rather boring. Let's get some content onto the screen.

Currently Ember.js has created four objects with default behavior, all four of which are related to the Notes application:

- An application route
- An application controller
- An application view
- An application template

You don't need to know what these four objects do at this point in time. What is important to know is that you can override these default objects to include customized behavior.

To write some text onto the page, let's override the default application template with custom markup.Add a `script` tag inside your `head` tag of index.html. The type of this script tag needs to be`""` and must include the name (`id`) of your template, as shown in this example.

Listing 1.3 Overriding the application template

```
<script type="text/x-handlebars" id="application">                    #A
        Hello Notes Application!                                      #B
</script>                                                             #C
```

#A: Creates a Handlebars.js template named application
#B: Contents of template will be written to the screen
#C: Remember to close your script tag!

Load your index.html file into your browser and you should see the text "Hello Notes Application!" as shown in figure 1.8.

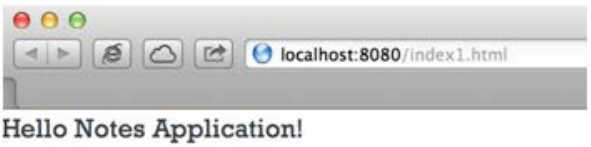


Figure 1.8 Rendering the application template

Running your application

Although you can simply run the Notes application by dragging the index.html file into the browser, I would recommend hosting the application in a proper webserver. You can use the webserver that you are most comfortable with. If you just want to start up a small lightweight webserver that will host the current directory you are in, you can use either the asdf Ruby Gem, or a simple Python script.

If you have Ruby installed, install the asdf Gem by typing `gem install asdf` into your terminal (Mac, Linux) or command prompt (windows). Once the gem is installed, you can host the current directory by executing `asdf –port 8080` in your terminal or command prompt. Once the gem is started your can navigate to http://localhost:8088/index.html in order to load your Notes application.

If you have Python installed you can execute `python -m SimpleHTTPServer 8088` in you terminal or command prompt. Once Python is started, you can navigate to http://localhost:8088/index.html in order to load your Notes application.

Now that you've got some text on the screen, you can move on to defining the setup for the rest of the Notes application. To do this, you'll need to think about which states (routes) your application can be in.

But first, delete the application template we added in listing 1.3. For the rest of the chapter, you won't need to override the default application template, so go ahead and delete it.

1.4.3 Defining application routes

Looking back at figure 1.7, you can see that the Notes application can be one of two logical states: the list of notes at the left of the application window represents one state, and the selected note's content at the right represents the second state. In addition, the selected note state is dependent on the selection made in the list at the left. Based on this, you can split the application up into two routes. Name the initial route notes. Once a user selects a route, the application transitions to the second route, which you'll name notes.note.

The Ember Router and how routes work is thoroughly explained in chapter 3. For now, add the following route definition to your app.js file:

Listing 1.4 Defining the application's router

```
Notes.Router.map(function () {                                    #A
    this.resource('notes', {path: "/"}, function() {              #B
        this.route('note', {path: "/note/:note_id"});             #C
    });
});
```

#A: Defines application router
#B: Defines top-level route, notes, responding to URL "/"
#C: Defines subroute, notes.note, responding to URL "/note/:note_id"

This code creates a map of your application's routes inside the `Notes.Router` class. Your router has two routes: one is named notes andbelongs to the URL "/"; the other is named note and is a subroute of the notes route. A route that can have subroutes is referred to as a *resource* in Ember.js, whereas a leaf route is referred to as a *route*.

The leaf route derives its fully qualified name as a combination of its parent route name and its own name. For example, the `note` route in listing 1.4 is referred to as the notes.note route. This convention extends to controllers, views, and templates, too. Based on the router you defined, Ember.js creates the following default object implementations:

- `Notes.NotesRoute`
- `Notes.NotesController`
- `Notes.NotesView`
- `notes` template

- `Notes.NotesNoteRoute`
- `Notes.NotesNoteController`
- `Notes.NotesNoteView`
- `notes/note` template

In addition, each of your application's routes binds itself to a relative URL path for two-way access, meaning that it responds as expected to URL changes, while at the same time updates the URL when you transition between states programatically. The concepts of routes might seem confusing at first, but rest assured they're thoroughly explained in chapter 3.

> **NOTE** Even though Ember.js will create default implementations of each of the files listed above, we only need to override the ones that we want to change. This means that our Notes application won't have implementation for all of the classes listed.

Now that you've defined which routes your application has, you also need to tell your Notes application what data each of the two routes have available to them. The following listing shows the definition of both the notes and the `notes.note` routes.

Listing 1.5 Defining the application's routes

```
Notes.NotesRoute = Ember.Route.extend({                          #A
    model: function() {                                          #B
        return this.store.find('note');
    }
});

Notes.NotesNoteRoute = Ember.Route.extend({                      #C
    model: function(note) {                                      #B
    return this.store.find('note', note.note_id);
    }
})
```

#A: Defines the notes route by extending Ember.Route
#B: Defines the data available to this route
#C: Defines the notes.note route by extending Ember.Route

This code introduces a couple of new concepts. The most obvious is that each of your routes extends from `Ember.Route`. Next, you're using the `model()`function to tell each of your routes what data is valid within them. We won't discuss what the code inside the `model()`functions does in detail here.

Using Ember Data, you're telling your `notes` route to populate a `NotesController` with all the notes registered in your system. Similarily, you're telling the notes.note route to populate a `NotesNoteController` with only the selected note. In addition, you're using a Local Storage Adapter for Ember Data, which means that the notes that you create are stored locally in the browser and made available to the application across site refreshes. The concept of Ember Data might seem confusing to you at this point. Don't worry, though, we'll go through Ember Data in detail in chapter 5.

Now, let's add some real content to your application.

1.4.4 Creating and listing notes

Inside the `notes` route, you'll include an input text field and a button so that users can add new notes to the application. Beneath these itemsyou'll provide a list of all the notes that are registered in your application.

Because you already defined your routes, all you need to get started is to add a new template called `notes`. The following code shows the text field and the button added to index.html.

Listing 1.6 Adding a template, input field, and button

```
<script type="text/x-handlebars" id="notes">                    #A
    <div id="notes" class="azureBlueBackground
        azureBlueBorderThin">                                   #B
        {{input}}                                               #C
        <button class="btn btn-default btn-xs">                 #D
            New Note
        </button>
    </div>
</script>
```

#A: Defines template named notes
#B: Wraps template contents inside div with id notes
#C: Adds text field to template
#D: Adds button labeled "New Note"

You wrap the contents of your `notes` template in a `div` element with the `id` of notes to ensure that the correct CSS styling is applied to the list of notes. Inside this `div` element, you add the text field and the button. For now, they don't have any functionality because you haven't told Ember.js what to do with the text entered in the text field or what to do when the user clicks the button.

The end goal is to allow the user to enter the name of a new note in the text field, and then click the button to create the note and save it to the browser's local storage.

To achieve this, you need to bind the content of your text field to a variable on the `NotesController` and add an action that triggers on the `NotesController` when the button is clicked. Ember.js automatically created a default `NotesController` for you, but to implement your action you need to override it. The following listing shows the additions made to the app.js file.

Listing 1.7 Create the NotesController

```
Notes.NotesController = Ember.ArrayController.extend({             #A
    newNoteName: null,                                              #B

    actions: {                                                      #C
        createNewNote: function() {                                 #D
            var content = this.get('content');
            var newNoteName = this.get('newNoteName');
            var unique = newNoteName != null && newNoteName.length > 1;
```

```
            content.forEach(function(note) {                              #E
                if (newNoteName === note.get('name')) {
                    unique = false; return;
                }
            });

            if (unique) {                                                 #F
                var newNote = this.store.createRecord('note');            #F
                newNote.set('id', newNoteName);                           #F
                newNote.set('name', newNoteName);                         #F
                newNote.save();                                           #F

                this.set('newNoteName', null);                            #F
            } else {
                alert('Note must have a unique name of at
                    least 2 characters!');                                #G
            }
        }
    }
});
```

#A: Extends Ember.ArrayController
#B: Binds newNoteName property to text field
#C: Defines the actions on controller
#D: Defines createNewNote action
#E: Ensures that note name is unique
#F: If name is unique, creates note using Ember Data createRecord, persists note to browser's local storage, and resets contents of text field
#G: If note name isn't unique, alerts user

A lot is happening in the code in this example. First, you create a controller named `Notes.NotesController`. Because this controller contains a list of notes, you extend `Ember.ArrayController`.

Next, you define a `newNoteName` property on the controller. You'll bind this to your input text field. You could omit this declaration here, as Ember.js would've created it automatically for you the first time the user typed into the text field, but I like to be explicit with the properties that my templates are using. This is a personal preference, though, and your opinions might differ.

The contents of the `createNewNote` action are straightforward. You verify that the name of the new note contains at least two characters before ensuring that no other notes exist in the system with the same name. Once the new note name has been verified, you create a new note and persist it into the browser local storage.

To add notes via the application's user interface, you need to update the notes template. But first you need to initialize Ember Data. This listing shows the code added to app.js.

Listing 1.8 Initializing Ember Data

```
Notes.Store = DS.Store.extend({                                           #A
    adapter: DS.LSAdapter                                                 #B
});
```

```
Notes.Note = DS.Model.extend({                                    #C
    name: DS.attr('string'),                                      #D
    value: DS.attr('string')                                      #E
});
```

#A: Creates Notes.Store class extending Ember Data's DS.Store
#B: Specifies to use the Local Storage Adapter
#C: Creates definition for the Note model object
#D: Specifes name property of type string
#E: Specifes value property of type string

Now that your application is set up to use the browser's local storage through Ember Data, you can bind your text field value and your button action to the `Notes.NotesController`. The following listing shows the updated notes template in index.html.

Listing 1.9 Adding a binding

```
<script type="text/x-handlebars" id="notes">
     <div id="notes" class="azureBlueBackground azureBlueBorderThin">
          {{input valueBinding="newNoteName"}}                          #A
          <button class="btn btn-default btn-xs" {{action "createNewNote"}}>New
Note</button>                                          #B
     </div>
</script>
```

#A: Binds value of text field input to newNoteName property
#B: Adds action to trigger createNewNote action on NotesController

Now that you can add new notes to the application, you also want the ability to list all notes in the Notes application. To implement this functionality, add the code shown in the next listing to the notes template.

Listing 1.10 Creating the list of notes

```
<script type="text/x-handlebars" id="notes">
    <div id="notes" class="azureBlueBackground azureBlueBorderThin">
        {{input valueBinding="newNoteName"}}
         <button class="btn btn-default btn-xs"
            {{action "createNewNote"}}>
            New Note
        </button>

        <div class="list-group" style="margin-top: 10px;">            #A
            {{#each controller                                        #B
                <div class="list-group-item">                        #C
                    {{name}}                                          #C
                </div>
            {{/each}}
        </div>
    </div>
</script>
```

#A: Adds Twitter Bootstrap list group to hold notes list
#B: Iterates over each note registered in NotesController
#C: Prints out the name of each note

The additions to the code are straightforward, if a little unfamiliar at this point. You use the `{{#each}}` Handlebars.js expression to iterate over each of the notes inside the `Notes.NotesController`. For each of the notes, you print out the name. You use Twitter Bootstrap to style your user interface. Figure 1.9 shows the results of loading the updated index.htm.

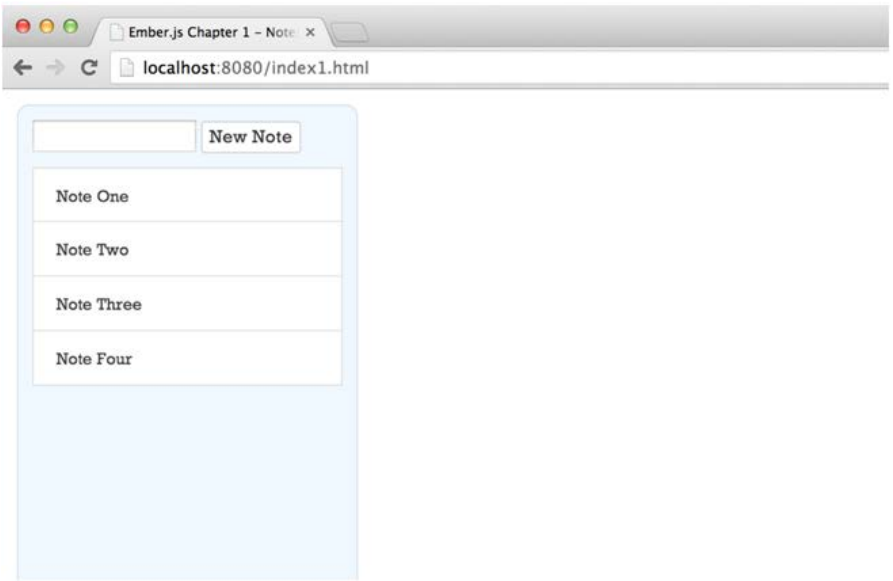


Figure 1.9 The updated Notes application after loading index.htm

At this point, you might think that you went through quite an ordeal to get a list of notes onto the screen, but as you'll soon discover, all that hard work will pay dividends.

Next up, you'll implement part two of the application: selecting a note in the list to transition to the `notes.note` route and view the contents of each note.

1.4.5 Selecting and viewing a note

What's a notes application without the ability to write text into the individual notes? By the end of this section, you'll have implemented this part of the application.

> **NOTE** The complete source code for this section is available as app2.js in either your source code directory or online via GitHub: https://github.com/joachimhs/Ember.js-in-Action-Source/blob/master/chapter1/notes/js/app/app2.js. This means that the completed source code for this section will be found within the index2.html and app2.js files.

The first thing you need to do is to link each of the individual notes to the `notes.note` route using the `{{#linkTo}}` expression. The following listing shows the updated notes template.

Listing 1.11 Linking each note to the notes.note route

```
<script type="text/x-handlebars" id="notes">
    <div id="notes" class="azureBlueBackground azureBlueBorderThin">
        {{input valueBinding="newNoteName"}}
        <button class="btn btn-default btn-xs"
            {{action "createNewNote"}}>
            New Note
        </button>

        <div class="list-group" style="margin-top: 10px;">
            {{#each controller}}
                <div class="list-group-item">
                    {{#linkTo "notes.note" this}}                #A
                        {{name}}                                 #A
                    {{/linkTo}}                                  #A
                </div>
            {{/each}}
        </div>
    </div>
</script>
```

#A: Wraps name of note in a linkTo expression

Wrapping the `{{name}}` expression inside a `{{linkTo}}` expression is the most common way to transition the user from one route to another as the user navigates your Ember.js application. The `{{linkTo}}` expression takes one or two attributes: the first is the name of the route to transition to; the second attribute specifies the context that the `{{linkTo}}` expression injects into the linked-to route.

In the case of your Notes application, you want to transition the user from the `NotesRoute` to the `NotesNoteRoute` whenever the user clicks the name of a note. In addition, you want to pass in the selected note to the `NotesNoteRoute`.

Refresh your application and select a note by clicking it. Each note name in the list is now an HTML hyperlink. When you click a note, notice that the application URL updates to reflect which note you're currently viewing (see figure 1.10).

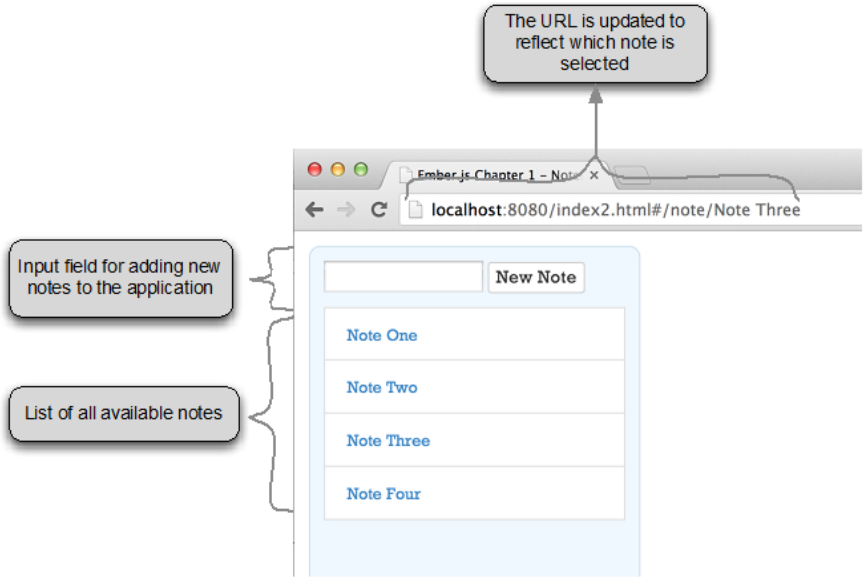


Figure 1.10 The notes in the list are HTML hyperlinks, and the browser URL updates when a note is selected.

Now that you can view and select your notes, you also want to be able to display the contents of the selected note at the right of the list.

To display the selected note, you need to create a `notes/note` template. But before you do so, you need to tell the notes template where it should render its subroutes by adding an `{{outlet}}` expression to the template. The following listing shows the updated notes template.

Listing 1.12 Adding an outlet to the notes template

```
<script type="text/x-handlebars" id="notes">
    <div id="notes" class="azureBlueBackground azureBlueBorderThin">
        {{input valueBinding="newNoteName"}}
        <button class="btn btn-default btn-xs"
            {{action "createNewNote"}}>
            New Note
        </button>

        <div class="list-group" style="margin-top: 10px;">
            {{#each controller}}
                <div class="list-group-item">
                    {{#linkTo "notes.note" this}}
                        {{name}}
                    {{/linkTo}}
                </div>
            {{/each}}
        </div>
    </div>
```

```
    {{outlet}}                                                    #A
</script>
```

#A: Specifies where to render notes template's subroutes

After telling the notes template where to render the `notes.note` route, you can add a template that shows the selected note. Create the new template inside your index.html file with the id `notes/note`, the contents of which are shown in the following listing.

Listing 1.13 Adding the notes.note template

```
    <script type="text/x-handlebars" id="notes/note">                 #A
        <div id="selectedNote">                                       #B
            <h1>name: {{name}}</h1>                                   #C
            {{view Ember.TextArea valueBinding="value"}}              #D
        </div>
    </script>
```

#A: Defines template with name notes/note
#B: Wraps template in div with id selectedNote
#C: Prints name of note in a header tag
#D: Creates text area for viewing and updating note content

Although you've added only a small amount of code to allow the user to select a note and view its contents, you now have an application that allows the user to do the following:

- Create a note and add it to the list of notes
- View a list of all notes added to the application
- Select a note, which transitions the user to a new route and updates the URL
- View and edit the contents of the selected note
- Refresh the application while viewing a specific note, which initializes the application and displays the same note to the user

Figure 1.11 shows the updated application.

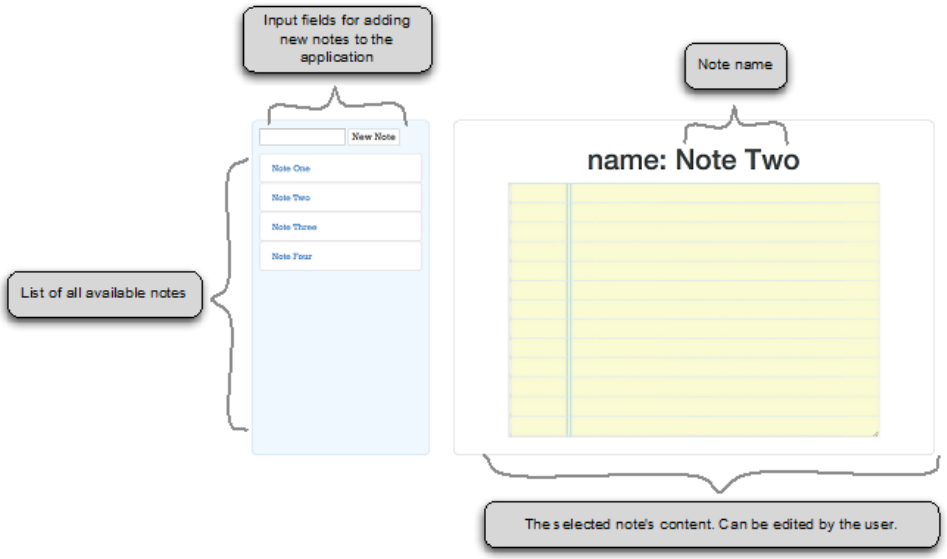


Figure 1.11 The selected note's content appears at the right.

Before we move on to the deletion of notes, let's fix two issues:

- The application doesn't indicate which note is currently selected.
- There is no way to persist changes made to the selected note.

To fix the first issue, use Twitter Bootstrap CSS styling in combination with the addition of a CSS class to the `{{linkTo}}` expression,. as shown in the following listing.

Listing 1.14 Highlighting the selected note

```
<script type="text/x-handlebars" id="notes">
    <div id="notes" class="azureBlueBackground azureBlueBorderThin">
        {{input valueBinding="newNoteName"}}
        <button class="btn btn-default btn-xs"
            {{action "createNewNote"}}>
            New Note
        </button>

        <div class="list-group" style="margin-top: 10px;">
            {{#each controller}}
                 {{#linkTo "notes.note" this class="list-group-item"}}  #A
                    {{name}}
                 {{/linkTo}}
            {{/each}}
         </div>
     </div>

    {{outlet}}
</script>
```

#A: Removes div and adds CSS class

That subtle change, removing the `div` element and adding a CSS class name to the `{{linkTo}}` expression is enough to successfully highlight the selected note in a blue color. Notice also that this feature works whether you click a note or you enter the `notes.note` route directly via a URL (or hit refresh).

To fix the second issue, start by adding an Update button to the notes/note template as shown in the following listing.

Listing 1.15 Adding a button to the notes/note route

```
    <script type="text/x-handlebars" id="notes/note">
        <div id="selectedNote">
            <h1>name: {{name}}</h1>
            {{view Ember.TextArea valueBinding="value"}}
            <button class="btn btn-primary form-control mediumTopPadding"
{{action "updateNote"}}>Update</button><br />                          #A
        </div>
    </script>
```

#A: Adds button that will trigger the action updateNote on the NotesNoteController

Once this button is in place, add an action to the `Notes.NotesNoteController` to perform the update of the note. Up until now, you've managed perfectly OK without overriding the default NotesNote controller that Ember.js created for you. The following listing shows the updated controller in app.js.

Listing 1.16 Adding a NotesNote controller to update the note

```
Notes.NotesNoteController = Ember.ObjectController.extend({            #A
    actions: {
        updateNote: function() {                                       #B
            var content = this.get('content');
            console.log(content);
            if (content) {
                content.save();                                        #C
            }
        }
    }
});
```

#A: Creates Notes.NotesNoteController extending from Ember.ObjectController
#B: Adds action to catch click of Update button
#C: Saves any changes made

Your application should now look like figure 1.12. Note that the selected note is highlighted in blue, the URL is updated to reflect this, and an update button now appears below the text area.

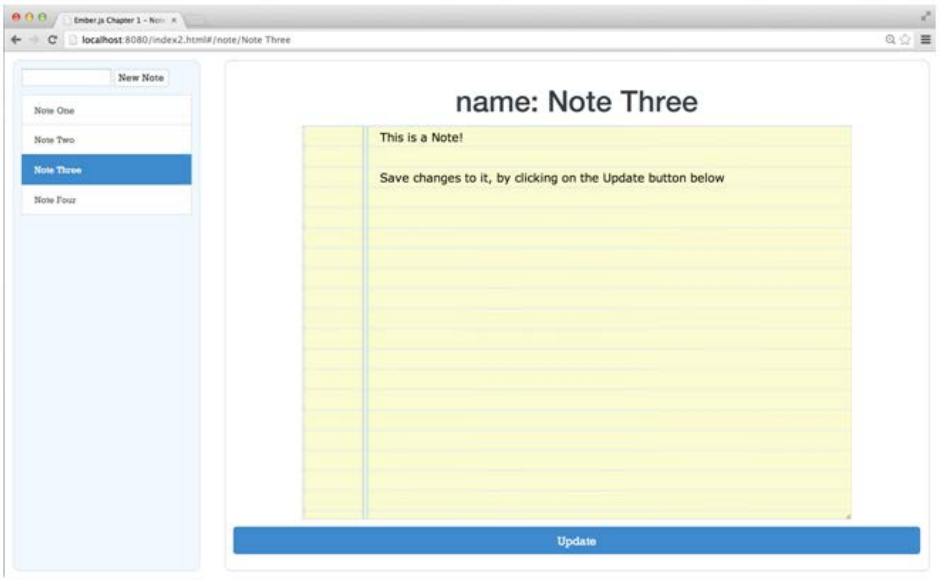


Figure 1.12 The application now indicates which note is selected and saves updates made to the selected note.

You can now move on to the final piece of the Notes application: deleting notes.

1.4.6 Deleting notes

In this section, you'll implement the third and last part of the Notes application.

> **NOTE** The complete source code for this section is available as app3.js in either your source code directory or online at GitHub: https://github.com/joachimhs/Ember.js-in-Action-Source/blob/master/chapter1/notes/js/app/app3.js.

To delete notes, add a Delete button to the selected note in the list at left. When the user clicks this button, the Notes application presents a modal panel asking for confirmation before the note is deleted. Once the user confirms that the note deserves to be deleted, the note is removed from the `Notes.NotesController's content` property and the `selectedNote` property is reset to `null`. To implement this feature, add a modal panel to your application, which is available from the Twitter Bootstrap framework. You also need to add a couple of new actions to the `Notes.NotesController`.

Start by adding the Delete button to the `notes` template as shown in the following listing.

Listing 1.17 Adding a Delete button to the notes template

```
<script type="text/x-handlebars" id="notes">
    <div id="notes" class="azureBlueBackground azureBlueBorderThin">
```

```
            {{input valueBinding="newNoteName"}}
            <button class="btn btn-default btn-xs"
                {{action "createNewNote"}}>
                New Note
            </button>

            <div class="list-group" style="margin-top: 10px;">
                {{#each controller}}
                    {{#linkTo "notes.note" this class="list-group-item"}}
                        {{name}}

                        <button class="btn btn-danger btn-xs pull-right"
                            {{action "doDeleteNote" this}}>
                            Delete
                        </button>                                    #A
                    {{/linkTo}}


                {{/each}}
            </div>
        </div>

        {{outlet}}
    </script>
```

#A: Adds button that fires doDeleteNote action on NotesController

Once you add the button to the user interface you can add the new `doDeleteNote` action to the `Notes.NotesController`. This time, you pass in `this` to the `doDeleteNote` action to tell the action which note you're attempting to delete. The updated controller is shown in the following listing.

Listing 1.18 Adding the `doDeleteNote` action to the NotesController

```
Notes.NotesController = Ember.ArrayController.extend({
    needs: ['notesNote'],

    newNoteName: null,

    actions: {
        createNewNote: function () {
            //Same as before
        },

        doDeleteNote: function (note) {                              #A
            this.set('noteForDeletion', note);                       #B
            $("#confirmDeleteNoteDialog").modal({"show": true});     #C
        },
```

#A: Adds new doDeleteNote action to NotesController
#B: Stores deleted note in noteForDeletion property on controller
#C: Displays the confirmation modal dialog

As you can see, the `doDeleteNote` action now takes a single parameter. Because you passed in the note you want to delete in the third argument of the `{{action}}` expression, Ember.js makes sure that this object is passed into your action. At this point, you don't want to delete

the note without first making sure that this is what the user wants to do. Before you display a confirmation message to the user, you need to temporarily store which note the user wants to delete. Once that's done you show the modal panel to the user, which you'll create next.

Because the HTML code to render the Twitter Bootstrap modal panel is slightly verbose, and you can potentially reuse it in multiple parts of your application, let's create a new template that renders the modal panel onto the screen. Start by creating a template called `confirmDialog` inside index.html as shown in the following listing.

Listing 1.19 Adding a template for the modal panel

```
<script type="text/x-handlebars" id="confirmDialog">                    #A
    <div id="confirmDeleteNoteDialog" class="modal fade">               #B
        <div class="modal-dialog">
            <div class="modal-content">
                <div class="modal-header centerAlign">                  #C
                    <h1 class="centerAlign">Delete selected note?</h1>
                </div>
                <div class="modal-body">                                #D
                    Are you sure you want to delete the selected Note?
                    This action cannot be be undone!
                </div>
                <div class="modal-footer">
                    <button class="btn btn-default"
                        {{action "doCancelDelete"}}>
                        Cancel
                    </button>                                           #E
                    <button class="btn btn-primary"
                        {{action "doConfirmDelete"}}>
                        Delete Note
                    </button>                                           #F
                </div>
            </div>
        </div>
    </div>
</script>
```

#A: Adds template named confirmDialog
#B: Creates div element with id confirmDeleteNoteDialog for modal panel
#C: Adds text to display in header of panel
#D: Adds text to display in body of panel
#E: Creates Cancel button that fires the action doCancelDelete
#F: Creates Delete button that fires the action doConfirmDelete

The modal panel is straightforward once you get past the Twitter Bootstrap markup, which in this case is somewhat verbose. The panel includes a header area, a body area, and a footer area. For the Notes application, you add text to the modal panel that prompts the user to confirm that they want to delete the note as well as informs the user that the operation can't be undone. In the footer you add two buttons: one that cancels the deletion and one that deletes the note. The Cancel button calls on its controller's `doCancelDelete` action; the Delete button calls on its controller's `doConfirmDelete` action.

To display the modal panel, you need to add only one line of code that tells the notes template where to render your new `confirmDialog` template. To achieve this, use the `{{partial}}` expression, as shown in the following listing.

Listing 1.20 Rendering the confirmDialog template

```
<script type="text/x-handlebars" id="notes">
    <div id="notes" class="azureBlueBackground azureBlueBorderThin">
        //Content same as before
    </div>

    {{outlet}}

    {{partial confirmDialog}}                                    #A
</script>
```

#A: Renders template

The `{{partial}}` expression finds the template with a name that matches its first argument and then renders that template into the DOM.

Your final task is to implement the actions `doCancelDelete` and `doConfirmDelete` on your `Notes.NotesController`. The following listing shows the updated controller.

Listing 1.21 Implementing the `doCancelDelete` and `doConfirmDelete` actions

```
Notes.NotesController = Ember.ArrayController.extend({
    needs: ['notesNote'],                                        #A
    newNoteName: null,

    actions: {
        createNewNote: function() {
            //Same as before
        },

        doDeleteNote: function (note) {
            //Same as before
        },

        doCancelDelete: function () {                            #B
            this.set('noteForDeletion', null);                   #C
            $("#confirmDeleteNoteDialog").modal('hide');         #D
        },

        doConfirmDelete: function () {                           #E
            var selectedNote = this.get('noteForDeletion');      #F
            this.set('noteForDeletion', null);                   #C
            if (selectedNote) {                                  #G
                this.store.deleteRecord(selectedNote);           #H
                selectedNote.save();                             #H

                if (this.get('controllers.notesNote.model.id') ===
                    selectedNote.get('id')) {
                    this. transitionToRoute('notes');             #I
                }
            }
```

```
            $("#confirmDeleteNoteDialog").modal('hide');                #D
        }
    }
});
```

#A: Calls for access to Notes.NotesNoteController
#B: Implements doCancelDelete
#C: Resets property to null
#D: Hides modal panel
#E: Implements doConfirmDelete
#F: Retrieves note for deletion based on noteForDeletion property
#G: If user has a note to delete
#H: Deletes note and persists changes into local storage
#I: If the deleted note is currently being viewed, transitions user to notes route

A few things are happening with the code in this example. First, you've implemented the `doCancelDelete` action. The contents of this action are rather simple: you reset the controller's `noteForDeletion` property back to `null`, and then you hide the modal panel.

The `doConfirmDelete` action is slightly more involved. You first get the note you want to delete from the `noteForDeletion` property on the controller before you reset the property to `null`. Next, you ensure that the controller has a reference to an actual note to delete. Once you've confirmed this, you tell Ember Data to delete the record from its store. This only marks the note as deleted. To perform the delete operation, you need to call `save()` on the `note` object. Once this is done, the note is deleted from the browser's local storage and is also removed from the user's list of notes.

Before you close the modal panel and finish the `doConfirmDelete` action, you need to consider one more scenario: what should happen if the user is deleting the note currently being viewed? You have two options:

- Notify the user that it's not possible to delete the note that's currently being viewed
- Transition the user back to the notes route

For this application, I felt it more appropriate to do the latter.

If you look at the second line of the controller, a `needs` property has been added. This is one way to tell Ember.js that this controller will, at some point, require access to the instantiated `Notes.NotesNoteController`. You can then access this controller via the `controllers.notesNote` property. This allows you to compare the `id` property of the note being deleted with the `id` property of the note the user is viewing (if any). If these properties match, transition the user to the notes route via the `transitionToRoute()`function.

To try out the delete note feature, reload the completed Notes application in your browser and attempt to delete a note (see figure 1.13)

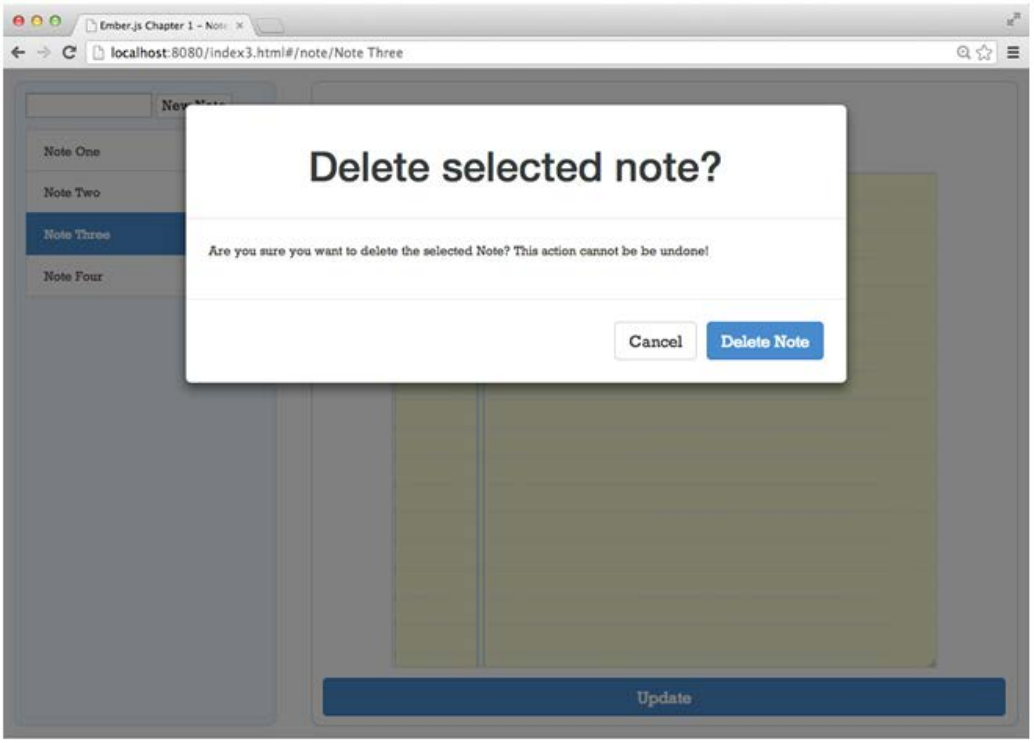


Figure 1.13 The completed Notes application displaying the Delete modal panel

That completes the functionality of the Notes application for this chapter. We will continue working with the Notes application while delving deeper into the Ember.js core concepts in chapter 2.

1.5 Summary

This chapter has provided an overview of the building blocks that Ember.js is based on and introduced the most important concepts of an Ember.js application. I hope that you've gained a better understanding of the Ember.js framework as well as why it exists and where it will be most applicable to you as a web developer.

As an introductory chapter to Ember.js, I guided you through the development process of a simple web application, touching on the important aspects of the framework along the way. The goal of the Notes application was to show you as many of the basic features as possible of Ember.js while trying not to complicate the application's source code.

Ember.js has a steep learning curve, but the benefits to you, as a web developer, are great and this chapter has shown some of the power that lies in this advanced framework.

The next chapter reuses and extends the code you wrote in this chapter to thoroughly explain the core features that Ember.js provides.

2

The Ember.js way

This chapter covers:

- How bindings work and how they affect your programming style
- Using automatic updating templates
- How and when to use computed properties and observers
- The Ember.js object and class model

This chapter builds on the code that you developed in chapter 1 to explain, in detail, the most defining aspects of the Ember.js framework. One of the key design goals of Ember.js is to make sane, reasonable default choices for you to reduce the amount of boilerplate code that you must write on your own. Ember.js uses default settings that work out of the box with the majority of web applications, and it allows you to override these defaults easily where applicable. Thanks to those sane choices, you can write large web applications without having to constantly consider how your data will get from A to B or how your web elements will be updated in a clean and efficient manner, and you can easily integrate with any third-party JavaScript Framework of your choice.

If you've worked with other programming environments that use the concept of bindings, such as Objective-C, Adobe Flex, and JavaFX, then you're already familiar with the mindset required to think of your application in terms of observers, bound variables, and automatic synchronization of data across your application. Otherwise, you need to clear out some of your old programming habits to leave room for these essential concepts in Ember.js, as they'll affect how you think about wiring your application together in a big way. Rewiring your development habits to fit into this loosely coupled and asynchronous way of thinking about your application code may be the hardest part about learning to user Ember.js efficiently.

Figure 2.1 shows the parts of the Ember.js ecosystem this chapter examines—ember-application, ember-views, and Handlebars.js.

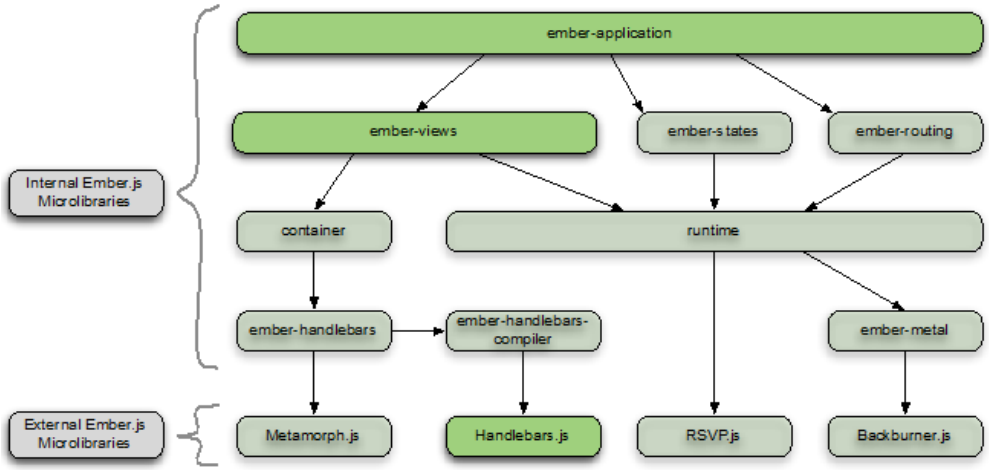


Figure 2.1 The parts of Ember.js addressed in this chapter

Throughout the chapter, you'll extend the Notes application created in chapter 1. The changes made affect the index4.html and app4.html files.

> **NOTE** The source code for these files can be found in this book's folder on the GitHub site: https://github.com/joachimhs/Ember.js-in-Action-Source/blob/master/chapter1/notes/js/app/app4.js.

Let's get started with one of the core key features, bindings, which the rest of Ember.js framework builds upon.

The Ember.js framework is based on a couple of related features that hold the whole framework together and that form the basis of all the other features offered in Ember.js. Knowing how these core features—bindings, computed properties, and observers—work is essential for any developer using Ember.js.

2.1 Using Bindings to Glue Your Objects Together

One set of tasks that you're most likely to encounter in a repetitive manner when writing web applications is requesting data from a backend resource, parsing the response to update your controllers, and then making sure that your views are up to date whenever your data changes.

Then, as the user works with the data, you need to make sure that you update *both* your in-browser cache and your view in a way that ensures that what is persisted in the in-browser cache matches what the user sees on their screen.

You've probably written code to support that use case hundreds, if not thousands of times, yet most web applications lack a site-wide infrastructure to handle these interactions in a clean

and consistent manner, leaving it up to the developer to reinvent the wheel for each application and each of its layers. A common implementation might look like figure 2.2.

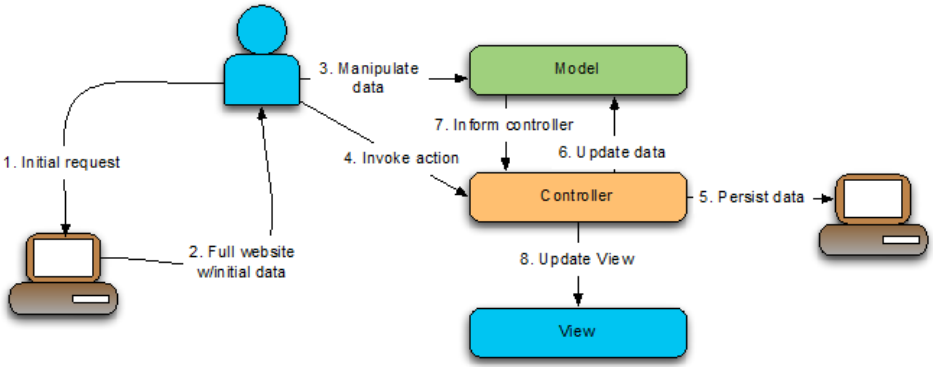


Figure 2.2 A common data synchronization implementation

This model assumes that you've thought about how you want to structure your application, and that you've implemented an MVC-like structure in your application. The Ember.js MVC model is slightly different from the MVC models you've become accustomed to when writing web applications, but fear not, the Ember.js MVC model is thoroughly explained in detail in chapter 3. The problem with the model shown in figure 2.2 is that it leaves it up to the developer to implement a structure that ensures that the data persisted to the server is, in fact, the data presented to the user. Besides the need to implement custom code for each of the six steps listed (steps 3–8), consider the many edge cases:

- What if the server is unable to persist your data?
- What if the server has had updates between the time you loaded the application data (in step 2) and when you persisted it (in step 5)?
- What if the user has created new data and the server needs to generate a unique identifier for that data (steps 3 and 5)?
- If the server changes any of the data persisted, how would the client be notified in order to reflect these changes?
- What happens if the user changes data that has been sent to the server, but a response hasn't arrived yet?
- What happens if the model is updated without the controller being notified of a change?
- What happens if multiple views need to show the same data? How would you synchronize the data so that your user interface is consistent?

These are just a few examples of the decisions that you'd need to make for every part of your application. You might've done your homework and implemented a common solution across

your application. In that case, good for you; now you know the difficulties involved in synchronizing your application data between views and controllers, between models, and between the client and the server. This is a major area where Ember.js will help you immediately with its complete and robust binding implementation. Ember.js also offers a complete persistence layer called Ember Data, discussed in chapter 7.

In their simplest form, bindings are a way to tell your application, "Whenever variable A changes, make sure to keep variable B up to date." Bindings in Ember.js can be either one-way or two-way. Two-way bindings work just the same as one-way bindings, but they keep two variables in sync regardless of which variable changes. The most common binding type you'll use throughout Ember.js is likely to be the two-way binding. There are two reasons for this. First, the two-way binding is the default binding structure in Ember.js; second, it's also the type of binding you're most likely to need when writing a client application.

There are two ways to declare a binding. You can use the `Ember.Binding.twoWay` or `Ember.Binding.oneWay` function calls to create one explicitly, and you need to do this to create one-way bindings. Most likely, however, you'll instead use the `Binding` suffix-keyword in your objects' property declarations. Ember.js is smart enough in its structure that you'll rarely need to implement bindings manually. For this reason, you didn't manually create any bindings within the Notes application developed in chapter 1.

But say you wanted to keep track of which note was selected on your `Notes.NotesController`. You can do this by binding a property, `selectedNote`, to the model object of the `Notes.NotesController`. Here's the updated `NotesController`.

Listing 2.1 Synchronizing two variables using bindings

```
Notes.NotesController = Ember.ArrayController.extend({
    needs: ['notesNote'],
    newNoteName: null,
    selectedNoteBinding: 'controllers.notesNote.model',              #A

    //Rest of controller left unchanged
});
```

#A: Creates binding between property and model

If you reload the application at this point and head over to the browser's console log, you should see the display shown in figure 2.3.

Figure 2.3 Showing the console log

Note here that Ember.js shows the versions of Ember.js, Handlebars.js, and jQuery that are in use by the application. When Ember.js instantiates your controllers and your routes, it puts them into a structure called the *container*. You can ask this container to look up the instantiated `NotesController`, to check the value of the `selectedNote` property. Type the following command in the console and press Enter. Figure 2.4 shows the result.

```
> Notes.__container__.lookup('controller:notes').get('selectedNote')
  undefined
>
```

Figure 2.4 Prompting the container for the value of the `selectedNote` property

As you can see, the `selectedNote` property is returned as undefined. This is the expected result, because you haven't selected any note yet. Now select one of your notes and execute the command again. Figure 2.5 shows the result.

```
> Notes.__container__.lookup('controller:notes').get('selectedNote')
  ▶ Class {id: "Note Three", store: Class, _changesToSync: Object, _deferredTriggers: Array[0], _data: Object…}
> Notes.__container__.lookup('controller:notes').get('selectedNote.id')
  "Note Three"
>
```

Figure 2.5 Prompting the container for the value of the `selectedNote` property with a note selected

As you can see, you can now get hold of the selected note via the `NotesController`'s `selectedNote` property. Note also that you were able to fetch the `id` property of the selected note by calling `get('selectedNote.id')`. Using this dot notation you can fetch and update values deep within your object hierarchy.

Even though you've added only a single statement in listing 2.1, Ember.js has helped you create the following features:

- A two-way binding between two controllers, keeping the variables in sync when changes occur.
- A clean separation of concern between the controllers.
- A high degree of testability and application flexibility through a loose coupling between controllers.
- The certainty of only a single definition of which note is the selected one, throughout the entire application. Knowing that `SelectedNoteController.model` will always represent this information enables you to create simple views that can be automatically updated whenever any change to the selected note occurs.

You'll next add some lines of code to understand how to bind the data all the way out to the view via templates that update automatically.

2.2 Automatically updating templates

By default, Ember.js uses the Handlebars.js template engine. One key aspect of the Ember Handlebars implementation is that whenever you connect your templates to your underlying data, Ember.js sets up two-way bindings between your application layers. You've seen how this works in the Notes application developed in chapter 1.

Consider the code for the notes/note template, shown next.

Listing 2.2 Revisiting the notes/note template

```
<script type="text/x-handlebars" id="notes/note">
    <div id="selectedNote">
        {{#if model}}                                                #A
            <h1>name: {{controller.model.name}}</h1>                 #B
            {{view Ember.TextArea valueBinding="value"}}             #C
            <button class="btn btn-primary form-control mediumTopPadding"
                {{action "updateNote"}}>Update
            </button><br />
        {{/if}}
    </div>
</script>#A: Defining a new template with the template name notes/note
```

#A: Displays content of template only if model defined
#B: Prints name property of model
#C: Binds text area and model values

There are two types of binding going on in this example. First you have template bindings via Handlebars expressions; second you're binding properties on a custom view via the `Binding` keyword, in a manner similar to what you saw in listing 2.1.

Let's first focus on the Handlebars expression `{{name}}`. Even though this is a simple expression in your template, a lot is going on behind the scenes. The notes/note template is injected with a context from its backing controller. In this case, the controller that's driving the data to this template is the `NotesNoteController`.

Behind the scenes, you're working on the `model` property of the `NotesNoteController`. This may seem odd to you at first, but `{{name}}` is just shorthand for `{{model.name}}`, which in turn is shorthand for `{{controller.model.name}}`. In fact, you could use either of these expressions to print out the name of the note in the template.

One neat thing about the Ember-Handlebars implementation is that whenever a change occurs in the properties that your template is bound to, Ember.js ensures that your views are kept in sync and automatically updated. For example, if you go into the console and change the name of your note, observers that are set up by Ember.js ensure that the view is updated. You can try this by launching the Notes application and selecting a note. Then issue the following command from the console:

```
Notes.__container__.lookup('controller:notesNote')
    .set('model.name', 'New Name')
```

You should now see that the name of your note has changed, both in the list of notes at the left, and also in the header for the selected note. Figure 2.6 shows the end result.

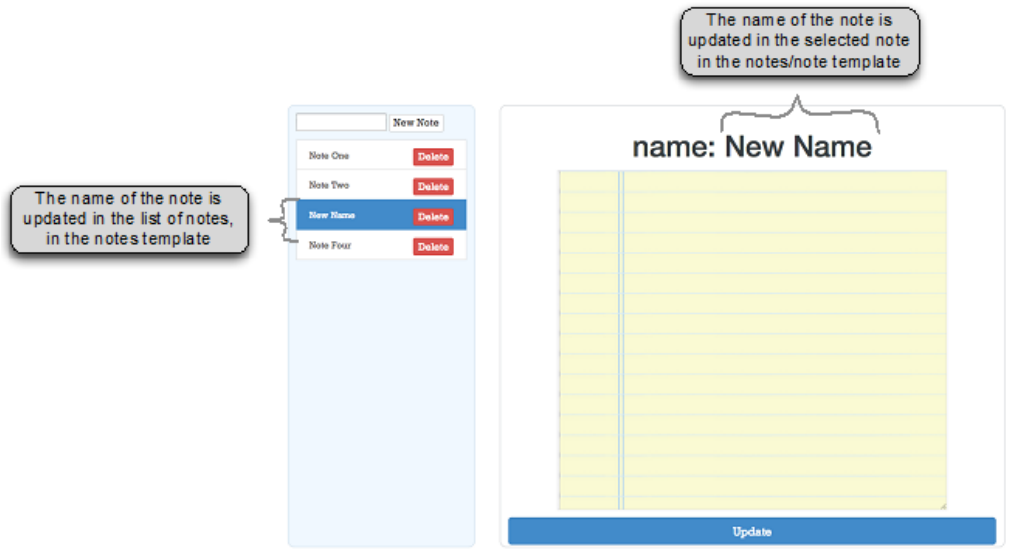


Figure 2.6 Changing the name of the note via the console

To show the selected note to the user only if one is selected, you use the Handlebars `if` helper. The statement `{{#if model}}` ensures that the code inside that `if` helper is executed only if the content attribute of the controller is *not* `null` or `undefined`. With a handful of code lines, Ember.js allows you to implement functionality that you'd otherwise have to handle manually for each of your views:

- Display the selected note only when a note is selected.
- Keep your DOM tree clean. If no note is selected, it keeps the element completely out of the DOM; there's no `display:hidden` in sight.
- Ensure that the user always sees the updated information about the notes, in any template that's rendered onto the website.
- Ensure that when the user changes the selected note's value (through the text area), Ember.js will update the underlying Note data.

Now try one more task before leaving automated templates behind. Create a brand-new `Note` object via Ember Data. You can do this by executing the following command in the browser's console:

```
Notes.__container__.lookup('store:main')
    .createRecord('note', {id: "New Note", "name": "New Note"})
```

First you fetch the Ember Data store from the container, using the keyword `store:main`. Then you create a new note via the `createRecord` function, passing in an `id` and `name` for your note. When you execute this command, notice that the new note is displayed at the bottom of the note list.

At this point it should be easy to envision how your application would behave if you implemented synchronization of notes between the Notes application and a real backend server application. Updating the number of notes available in the system, changing which note is selected, and even changing the selected note's data are all initiated via a push request from the server side. You've written only a handful of code statements, yet you've built a rather large list of features into the application, all while keeping the application structure reasonable.

The Ember.js default template engine, Handlebars, has many features that are integrated into Ember.js. Handlebars is thoroughly explained in chapter 4.

You may wonder how you're going to deal with situations where the data available doesn't exactly match up with the data that you want to display to the user. Similarly, what will you do if the data you're either displaying or dependent on in an `if` helper or `forEach` helper is complex? Cases like these are where computed properties come into action.

2.3 Computed properties

A *computed property* is a function that returns a value derived from other variables or expressions (even other computed properties) in your code. The difference between a computed property and a normal JavaScript function is that Ember.js treats your computed property function as if it were a real Ember.js property. As a result, you can call `get()` and `set()` on your computed properties, as well as bind to them or observe them (more on observers later in this chapter). Normally you find your computed properties on your model object, but every once in a while you need to use them in your controllers or views.

You don't have a computed property in the Notes application just yet, but suppose you want to improve the application by showing the first 25 characters of each note's value in the list of notes at the left of the application window. Instead of having to worry about issuing jQuery selectors and injecting/replacing information into a view somewhere, Ember.js allows you to define computed properties.

Let's create a computed property named `introduction` on the `Notes.Note` class that returns the first 25 characters of a note's text. The updated `Notes.Note` model class is shown in the following listing.

Listing 2.3 Adding a computed property

```
Notes.Note = DS.Model.extend({
    name: DS.attr('string'),
    value: DS.attr('string'),

    introduction: function() {                                  #A
        var intro = "";

        if (this.get('value')) {                                #B
            intro = this.get('value').substring(0, 20);
        }

        return intro;
    }.property('value')                                         #C
});
```

#A: Creates normal JavaScript function called introduction
#B: If value property of model has value, substrings out first 20 characters
#C: Adding .property makes introduction function computed property

Ember.js provides you with a great deal of functionality here. First, Ember.js is smart about when and how often it calculates the return value of a computed property. Until a computed property is used, its value isn't calculated at all. This is great for performance, as your application doesn't waste time calculating a lot of properties that may never be rendered onto the user's screen.

Looking at the structure of how you define a function as a computed property should give you a hint as to the second reason a computed property is calculated. Here, `property('value')` means "whenever the `value` property of `this` object change, recomputed the return value of this computed property". Therefore, as you type in the text area to add information to the `value` property of the note, you can see that the user interface is immediately updated to reflect your changes.

Until now, however, you haven't added the `introduction` computed property to any template. So expand the notes template slightly to show the first 20 characters of the `value` property in the list of notes, as follows.

Listing 2.4 Showing the `introduction` computed property in the notes template

```
<script type="text/x-handlebars" id="notes">
    <div id="notes" class="azureBlueBackground azureBlueBorderThin">
        {{input valueBinding="newNoteName"}}
        <button class="btn btn-default btn-xs" {{action "createNewNote"}}>New
Note</button>

        <div class="list-group" style="margin-top: 10px;">
            {{#each controller}}
                {{#linkTo "notes.note" this class="list-group-item"}}
                    {{name}}
                    {{#if introduction}}
                        <br />{{introduction}}
                    {{/if}}                                          #A

                    <button class="btn btn-danger btn-xs pull-right"
                        {{action "doDeleteNote" this}}>
                        Delete
                    </button>
                {{/linkTo}}


            {{/each}}
        </div>
    </div>

    {{outlet}}

    {{partial confirmDialog}}
</script>
```

#A: Adds introduction computed property to notes template

You've added only a single line of code to your template. This line simply states that if the `introduction` property of the current `Note` model isn't `null` or has a length greater than 0, a new line is printed out, followed by the content of the `introduction` property itself. Figure 2.7 shows the updated Notes application.

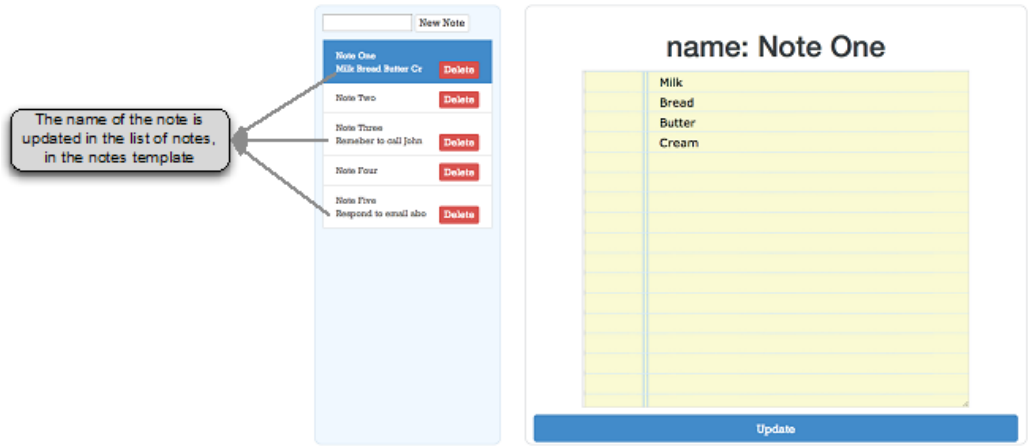


Figure 2.7 Adding the `introduction` computed property to the notes template

I mentioned that you can also use computed properties as setters. But how do you set a value that's derived from a combination of other properties? In listing 2.5, an object named `Notes.Duration` has a single property, called `durationSeconds`. Although it might make sense for a backend service to store the duration as seconds, it's inconvenient for the user to get the duration presented to them in seconds. Instead, you want to convert the seconds into a string with hours, minutes, and seconds separated by colons.

Listing 2.5 Using computed properties as setters

```
Notes.Duration = Ember.Object.extend({
    durationSeconds: 0,

    durationString: function(key, value) {                          #A
        if (arguments.length === 2 && value) {                      #B
            var valueParts = value.split(":");
            if (valueParts.length == 3) {                           #C
                var duration = (valueParts[0] * 60 * 60) +
                    (valueParts[1] * 60) + (valueParts[2] * 1);
                this.set('durationSeconds', duration);              #D
            }
        }
        var duration = this.get('durationSeconds');                 #E
        var hours   = Math.floor(duration / 3600);
        var minutes = Math.floor((duration - (hours * 3600)) / 60);
        var seconds = Math.floor(duration - (minutes * 60) -
```

```
            (hours * 3600));

        return ("0" + hours).slice(-2) + ":" +                          #F
            ("0" + minutes).slice(-2) + ":" + ("0" + seconds).slice(-2);
    }.property('durationSeconds').cacheable()
});
```

#A: Defines computed property with two arguments, key and value
#B: Determine if we are setting values
#C: Verifies that value splits into three parts as expected
#D: Updates durationSeconds with new duration
#E: Starts getter part of computed property
#F: Formats return value according to requirements

The first thing to notice about the code in listing 2.5 is that the computed property function now includes two function arguments, a key and a value. You can use these arguments to determine whether the function call is a getter or a setter by simply checking for exactly two arguments. Depending on your requirements, null might have a logical meaning or not; in this case you want to update the `durationSeconds` property only if the input has a valid format, by splitting the value an array of parts.

After you've verified that the input is valid, you start the conversion of the HH:MM:SS string into seconds before updating the object `durationSeconds` property with the updated value.

The second part of the computed property function is the getter, which as you might expect, does exactly the opposite of the setter part. It starts by fetching the `durationSeconds` property before it generates the `durationString` and returns it.

As you've probably guessed, it's fairly trivial to use a computed property in this manner to fill out an input field in the GUI via a simple binding to an HTML text field element. Ember takes care of automatically formatting the seconds to a human-readable duration, and vice versa when the user updates the duration in the text field.

I mentioned earlier that a computed property uses an observer to compute its value, but you've yet to see an observer in action, which brings us along to see how Ember.js observers work.

2.4 Observers

Conceptually a one-way binding consists of an observer and a setter, and a two-way binding consists of two observers and two setters. Observers go by different names and have different implementations across programming languages and frameworks. In Ember.js an *observer* is a JavaScript function that will be called whenever a variable it observes changes. You'd use an observer in situations where a binding is either not a sufficient mechanism or you simply want to perform a task whenever a value changes.

To implement an observer, use the `.addObserver()` method or the inline `observes()` method suffix.

The following code shows one possible use of an observer, starting and stopping a timer based on the number of items in a controller's content array.

Listing 2.6 Observing the length of a controller to control a timer

```
contentObserver: function() {
    var content = this.get('content');                                  #A
    if (content.get('length') > 0 && this.get('chartTimerId') == null) {
            //start timer
            var intervalId = setInterval(function() {                   #B
                if (EurekaJ.appValuesController.get('showLiveCharts')) {
                    content.forEach(function (node) {
                        node.get('chart').reload();
                    });
                }
            }, 15000);

            this.set('chartTimerId', intervalId);                       #C

        } else if (content.get('length') == 0) {                        #D
            //stop timer if started
            if (this.get('chartTimerId') != null) {
                EurekaJ.log('stopping timer');
                clearInterval(this.get('chartTimerId'));
                this.set('chartTimerId', null);
            }
        }
    }.observes('content.length')                                        #E
```

#A: Get content array from controller
#B: Starting a timer
#C: Store interval Id so we can stop it later
#D: Stopping the timer
#F: Observe number of controller model-items

As you can see, the observer, `contentObserver`, is a normal JavaScript method. It starts by getting the controller's `content` array. If there are items in the content array and a timer isn't already started, it creates a new timer that will fire every 15000 milliseconds. The timer goes through each of the items in the `content` array and reloads its data using a custom `reload()` method. If, on the other hand, there are no items in the content array, the code stops any timer that has already been started.

To make this function an observer, you append the inline `observes()` function with the path to the property you want to observe.

It's possible to construct the observer using the `addObserver()` method instead. The body of the function would remain the same, but its declaration would look slightly different, as follows.

Listing 2.7 Creating an observer using the `.addObserver` method

```
var myCar = App.Car.create({                                            #A
    owner: “Joachim”
    make: “Toyota”
});

myCar.addObserver(‘owner’, function() {                                 #B
    //The content of the observer
});
```

#A: Creates App.Car object
#B: Observes changes to owner property

Although it's possible to create observers this way, I find the inline version shown in listing 2.5 to be cleaner and more readable when looking at the source code of my applications. I also like to add the suffix `Observer` to my observer functions, but that isn't required.

OBSERVING CHANGES TO PROPERTIES WITHIN ARRAYS

Sometimes you may want to observe changes to properties within an array. In the Notes application the `Notes.NotesController` has a content array of Ember Objects with two properties, `name` and `value`. If you wanted to observe changes to each of the name properties, you could use the `@each` observer key, as follows.

Listing 2.8 Observing changes within arrays using @each

```
Notes.NotesController = Ember.ArrayController.extend({
    content: [],
    nameObserver: function() {                                    #A
        //The content of the observer
    }.observes('content.@each.name')
});
```

#A: Observes changes each name property of content array

2.5 The Ember.js object model

All the functionality shown in this chapter is made possible via the Ember.js object model. Ember.js extends the default JavaScript Object class definition in order to build a more powerful object model. In addition, it supports a mixin-based approach to sharing code between modules and between applications.

You may wonder how Ember.js knows that your properties have changed, and how it knows to fire off an observer or a binding. You may also have noticed that Ember.js requires you to use `get()` and `set()` whenever you want to get or update a property from any object that's a subclass of `Ember.Object`. Whenever `set()` is called on any property, Ember.js checks to see if the value you're updating with is different from the value that you already have in your object. If the two are different, Ember.js triggers any bindings, observers, or computed properties derived from the property you're updating.

Even though using `get()` and `set()` may seem awkward when you first get started using Ember.js, this mechanism is one of the features that enable Ember.js to intelligently batch up observers, bindings, computed properties, and DOM manipulations. In fact, the use of `get()` and `set()` is one of the cornerstones of how Ember.js solves the performance issues involved with multiple DOM updates and bindings.

To create a custom Ember.js object, you generally either extend another Ember.js object and enrich it with custom functionality using the `extend()` method, or you create an instance via the `create()` method. Regardless of which you choose, every object within your Ember.js application, in one way or another, extends the `Ember.Object` class, and it's this base class that enables Ember.js to provide you with the functionality discussed throughout this chapter and this book.

Instead of extending `DS.Model` for the `Notes.Note` model object, imagine that you didn't use Ember Data, and that you needed to provide a `Notes.Note` model yourself. Consider the following code.

Listing 2.9 Creating a `Notes.Note` object

```
Notes.Note = Ember.Object.extend({                                    #A
    name: null,
    value: null
});
```

#A: Creates new Notes.Note class definition

Instead of using `Ember.Object.create()` to create an anonymous `Note` instance, you create an explicit `Notes.Note` class by extending the `Ember.Object` class. Notice a couple of things here:

- You don't have an instance of the `Notes.Note` class yet because the `extend()` method doesn't return an instance.
- The `Note` class now starts with an uppercase N to signal, not only to you, as a developer, but also to the Ember.js framework, that this is, in fact, a class definition and not an object instance.

To create a new `Notes.Note` instance, you must use the `create()` method as shown next.

```
Listing 2.10 – Creating a new instance of Notes.Note
Notes.Note = Ember.Object.extend({                                    #A
    name: null,
    value: null
});

var myNewNote = Notes.Note.create({                                   #B
    'name': 'My New Note', 'value': null
});
```

#A: Creates new Notes.Note class definition
#B: Creates new instance of Notes.Note

At this point you may think you haven't gotten much further than with the anonymous `Ember.Object.create()` implementation. However, it's generally a good idea to explicitly define classes for all your data types and all objects that you use within your Ember.js application. Even though doing so requires more code, you clearly show your intent when you instantiate an object, and you're able to separate your domain model objects from one another cleanly.

The resulting code is much easier to read, more maintainable, and easier to test.

Cleanly defining your application's objects also makes it easier to add observers, bindings, and computed properties in the correct place to ensure that your application is as fast as possible.

Consider the scenario in which your backend application changes from supplying the `value` property of each `Note` object as a pure text implementation to, instead, encoding the value in

markdown format. With the `Notes.Note` specification in one place in your application, it's easy to add this functionality. Consider the following code.

Listing 2.7 Adding a computed property to convert from markdown to HTML

```
Notes.Note = Ember.Object.extend({
    name: null,
    value: null,

    htmlValue: function() {                                          #A
        var value = this.get('value');
        return Notes.convertFromMarkdownToHtml(value);
    }.property('value')
});
```

#A: Convert from markdown to HTML

Once you've successfully implemented the `Notes.convertFromMarkdownToHtml` function, it's trivial to change the view template of the application to use the new computed property `htmlValue` instead of using `value`; you simply change the Handlebars template for the view (see listing 2.3) to `{{view Ember.TextArea valueBinding="htmlValue"}}` instead.

Now that we have a working Notes application, lets take a closer look at how Ember.js synchronizes your data between the layers in Ember.js' MVC pattern.

2.5.1 Data synchronization between layers with Ember.js

Recall figure 2.2 in which you had a total of six points in the application where you explicitly had to track and make note of your internal application state to make sure that the data throughout the client-side application and the data sent to the backend really were in sync. Contrast that with how the Ember.js framework wants you to use bindings, controllers, and a clean model layer to automate as much of this boilerplate code as possible. Figure 2.8 shows an updated conceptual model.

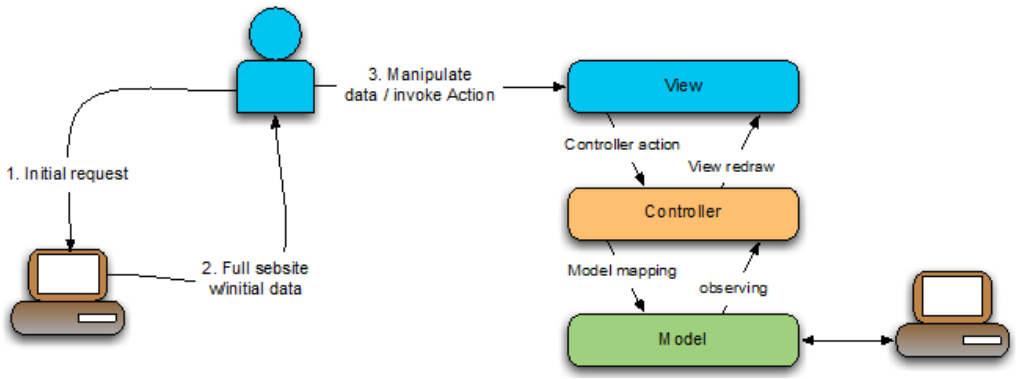


Figure 2.8 The Ember.js data synchronization implementation

As you can see, with the Ember.js approach, you leave more of the boilerplate code up to the Ember.js framework. You're still in full control of how the data flows through your application. The main difference is that the code is now explicit about these operations as close to the source as possible, which is where you tell Ember.js how your application is wired together.

As you'll see throughout this book, Ember.js takes a sane approach to the default operations, while it still lets you override these defaults whenever another approach would be a better fit for your specific use case. This lets you get on with doing what you're here for—writing ambitious web applications to power the future web.

2.6 Summary

This chapter has introduced some concepts that that you may not be familiar with, or that Ember.js treats differently than you may be used to. I've mentioned the Model-View-Controller model a few times, but I haven't delved into the details and how Ember.js helps you build true MVC applications for the web. That's what the next chapter is all about.

3

Putting everything together using Ember.js Router

This chapter covers

- The difference between the server-side and client-side MVC models
- The Ember.js MVC model in detail
- How the Ember.js MVC model is enriched with a statechart implementation
- How controllers and views are bound together
- The Ember.js Container

Ember.js attempts to put the developer into a full-featured, client-side Model-View-Controller (MVC) pattern, while also enriching the controller layer with a full-featured statechart implementation called Ember Router. If you're unfamiliar with statecharts, don't worry. We'll touch upon the key points of a statechart in section 3.3, as well as provide a link where you can read the complete specification.

Ember Router allows you to map out each of your application's states into a hierarchical structure, containing the relationship between the different states in your application as well as the paths that your user can travel through your application. Implemented correctly, these routes allows you to build a web app with a firm and stable structure with states that are loosely coupled, clearly defined, and highly testable.

We'll contrast the Ember.js MVC pattern against a more traditional server-side MVC pattern that has been popular since the early 2000s to show you how Ember Router fits snugly into the center of this picture, connecting your app's parts together to form a uniform user experience. The chapter wraps up with a section explaining how controllers and views are connected and how controllers can be bound together, all via Ember Router.

Figure 3.1 shows the parts of the Ember.js ecosystem this chapter examines—ember-application, ember-views, ember-states, ember-routing, and container.

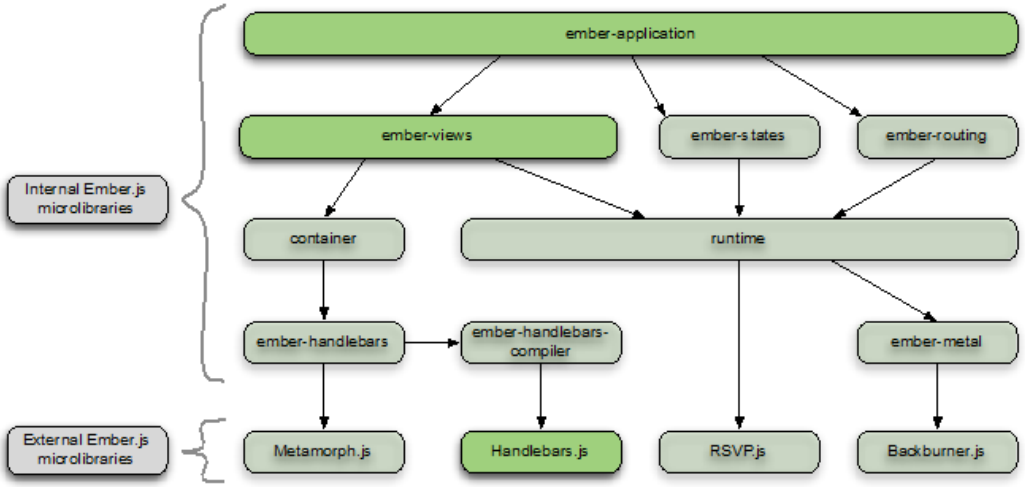


Figure 3.1 The parts of Ember.js you'll work on in this chapter

But before we get into the details of MVC, let's quickly look at the application that you'll build throughout this chapter.

3.1 Introducing the Ember.js in Action Blog

For this chapter you'll build a simple blog application that retrieves its data from a JSON file, located on the file system. In this application the user is presented with a list of available blog posts. The user can choose to view any of these posts by clicking the Full Article link presented beneath the introduction text of each post. Additionally, the top of the page provides the user with Home and About navigation links. You'll build up the application in three parts:

1. Build the blog index route
2. Build the blog post route
3. Define actions

In the first part you'll get acquainted with Ember.js's statechart implementation, Ember Router. From there you'll build up the blog index route. This route, as shown in figure 3.2, presents the user with the following details:

- A header displaying the name of the blog
- A list of all available posts; each entry included the blog post title, the opening text and the published date,
- A link to navigate to the full blog post

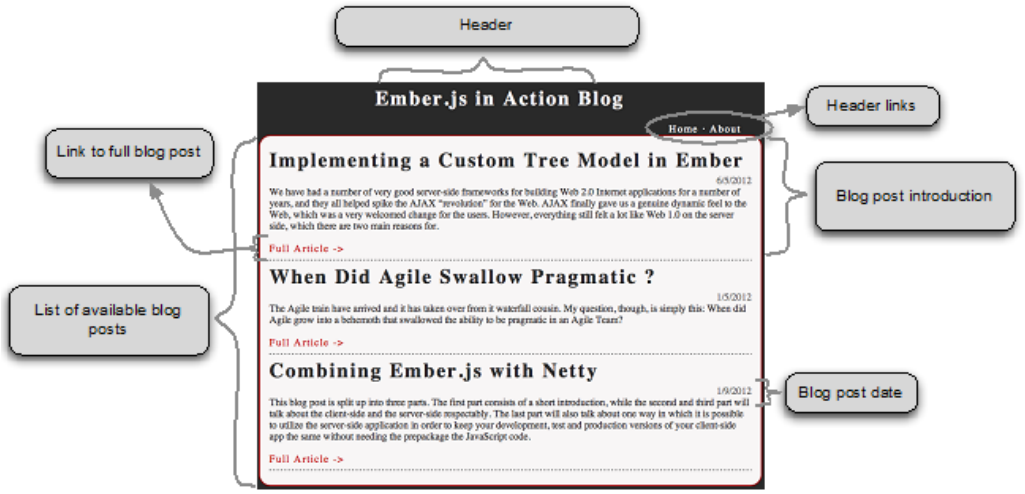


Figure 3.2 The blog index

In the second part you'll extend the router and add a `blog.post` route. This route allows the user to view the selected blog post in its entirety, which you parse from its Markdown format and present to the user in HTML. This route, as shown in figure 3.3, presents the user with the following details:

- The blog post contents, converted from Markdown to HTML
- A link to navigate back to the blog index
- The date the post was published

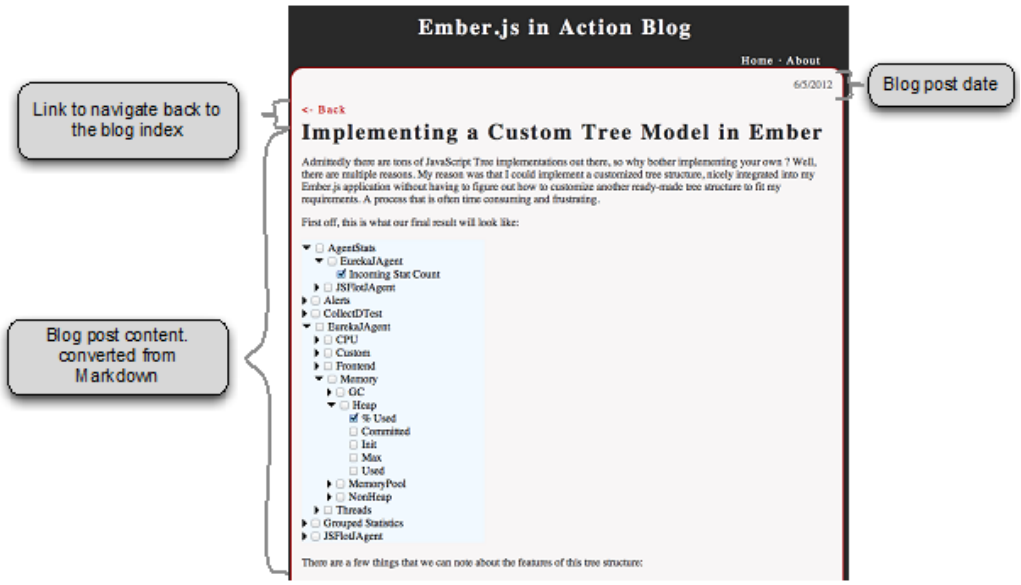


Figure 3.3 The selected blog post

In the final part you'll wrap everything up and include actions to navigate back to the index as well as create the About page.

To get started let's take a look at what Ember Router consists of. To see where the Ember Router fits into the application stack, you first need to understand the Ember.js MVC pattern.

3.2 The predicament of server-side Model-View-Controller patterns

Many MVC patterns are available, but they share common design principles and goals. The controller is the object that the user *uses*, and the view is the object that the user *sees*. In terms of implementation details, the controller is an object that accepts actions from the user and knows the data it needs to retrieve or update in the model layer to serve the user's request. The view, in turn, knows how to use that model to generate the GUI that the user sees. In some MVC implementations the model layer is reduced to a data-storage layer with dumb objects, whereas other MVC implementations have a rich model layer.

A conceptual and simplified view of the MVC pattern is shown in figure 3.4.

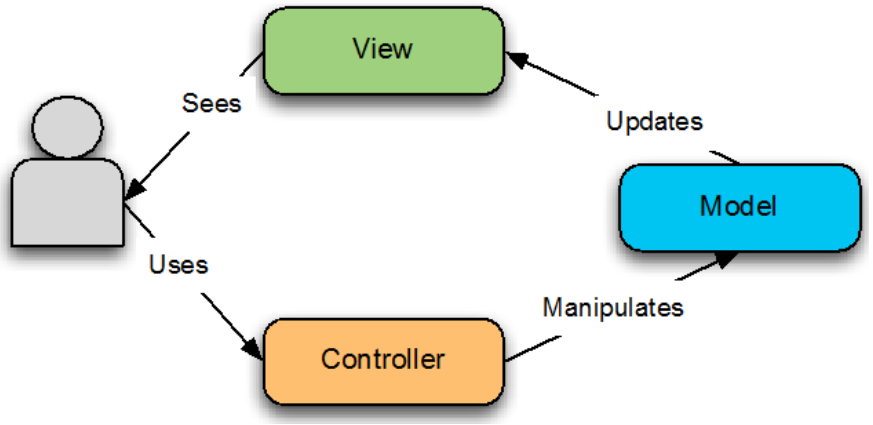


Figure 3.4 Simplified overview of the MVC pattern

When web apps and server-side scripting became popular, the web was restricted to a synchronous request-response cycle in which the server had to feed the browser with the complete web page for each request. The browser would then redraw the complete website based on the new content. The web had little going for it in terms of dynamic quality, which resulted in server-side adaptations of the MVC pattern with a severely reduced dynamic structure. Although many server-side MVC patterns can be found, most of them resemble the Sun MVC Model 2 structure in some way, as shown in figure 3.5.

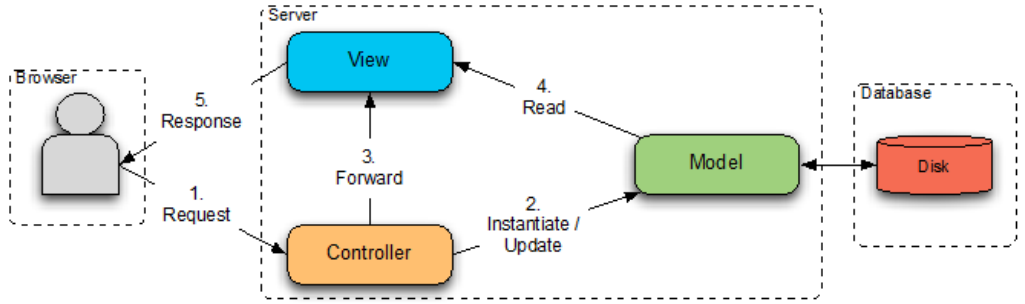


Figure 3.5 Sun MVC Model 2

In this structure, the controller is responsible for parsing the request to know what data to supply. This involves either string-query parsing the URL or parsing the HTTP packet body. The controller then updates or instantiates the model objects required for the view to render before forwarding the request to the correct view. The view uses the model objects supplied by the controller to generate the complete website, which it sends back to the user's browser.

This model has a number of strengths and weaknesses. The obvious weakness for any modern web application is that the model has no dynamic parts. After the view is generated and sent to the browser, no mechanism is in place to update the client-side view based on changes to the server-side model.

When Ajax (Asynchronous JavaScript and XML) became mainstream with the Web 2.0 hype, most frameworks solved this problem by increasing the amount of state that the server would need to keep track of for each of the users logged into the system. The server now needed to know not only how to stitch together complicated views but also the intricate details of the state of each of the users logged in so that it could serve the correct response to an Ajax request.

The difficulties the above approach brings along with it, in regard to both the application's business logic as well as its scalability, have led to the development of powerful client-side JavaScript MVC frameworks that store each user's state where it belongs—in the client.

These frameworks allow the application architecture to scale out more easily as it enables the server to do what the server does best: serve, update, and persist data. Less state on the server also means that implementing a strategy for horizontal scalability is much easier. The client is also left alone to do what it does best: keep state and render and display websites. Web apps replaced traditional client/server apps, but, in many ways, web apps now use the same client/server architecture as the apps they replaced. The major difference is that the client now uses generalized, free, and open technologies to render views, perform client-side business logic, and to request and receive data, wrapped in a neat, competitive bundle called the browser.

Most modern JavaScript web frameworks rely on having a full-featured MVC implementation in the browser to bring back the intended dynamic quality of the MVC pattern, and the Ember MVC pattern is no different.

3.2.1 The Ember MVC pattern

The purpose of the MVC pattern is to separate the concerns of your application's logic into clearly defined groups, or layers. Each layer has a specific purpose, which makes the application's source code more readable, maintainable, and testable. Let's take a deeper look at Ember.js's controllers, models, and views as well as how these fit together in the Ember.js ecosystem.

CONTROLLERS

The controller acts mainly as a link between the models and the views. Ember ships with some custom controllers, most notably the `Ember.ObjectController` and the `Ember.ArrayController`. Generally, you would use the `ObjectController` if your controller is representing a single object, (like a selected blog post), and the `ArrayController` if your controller represents an array of items (like a list of all blog posts).

I'll discuss the functionality within these controllers in more detail when you start to use them later in this chapter.

In the examples in this book I've used Ember Router to enrich the functionality of the controller layer to keep the individual controllers as small and independent as possible. Both are important aspects of scalable web applications.

MODELS

The model layer holds the data for the application. The data objects are specified through a semi-strict schema. The models have little functionality, and as you'll see, the model object is responsible for such tasks as data formatting. The view will bind the GUI components against properties on the model objects, via a controller.

I've enriched the model layer with Ember Data, a framework that implements an in-browser cache for the model objects and provides a means to implement Create, Read, Update, and Delete (CRUD) operations on that data via a unified API. We'll discuss Ember Data in detail in chapter 5.

VIEWS

The view layer is responsible for drawing its elements onto the screen. The views hold no permanent state of their own, with few exceptions.

Ember ships with a number of default views, and it's good practice to use these when you need simple HTML elements. For more complex elements in your web app you can easily create your own custom views that either extend or combine the standard Ember views.

The Ember.js view layer is enriched with Handlebars.js templates, which I've used extensively throughout this book as well as in any Ember.js project I've ever come across.

3.2.2 Putting everything together

Even though the MVC pattern on the client side has been extended to make the server-side code simpler, the total architecture of your system will be more complex than it was before (as shown in figure 3.4). This added complexity does come with a significant benefit: clean, structured, maintainable, and highly testable client-side code. A complete overview of the Ember.js MVC pattern, along with a typical server-side implementation is shown in figure 3.6.

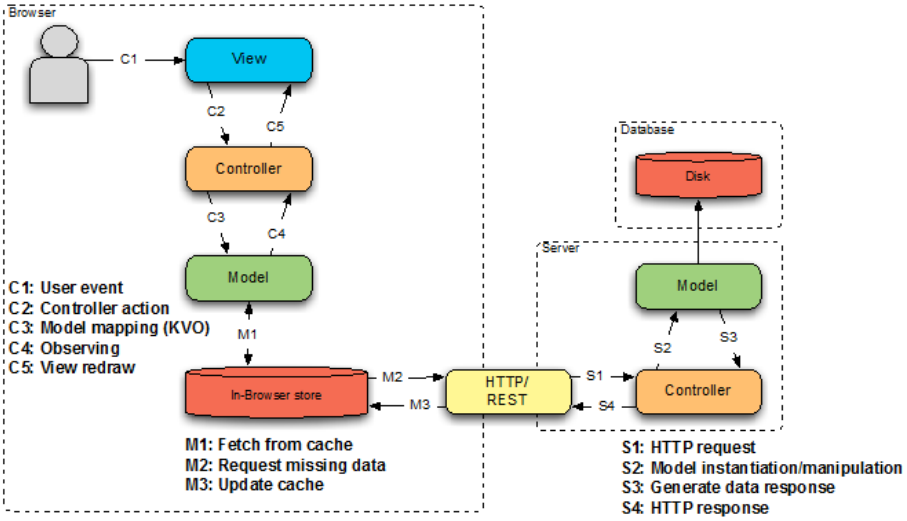


Figure 3.6 Ember MVC model

As you can see, the client side has been enriched with a complete life cycle of its own. I've split the complete `model` into three parts (not including the database). Imagine a client application in which the user selects an item from a list in the GUI. That selection (C1) triggers an action on the item's controller (C2) or, as we'll see later, on the router. This issues a request to the model layer (C3) for the data related to the item being selected (M1). As that data isn't available in the Ember Data in-browser cache, it must be fetched from the server (M2). While the client waits for the response to come back in an asynchronous manner, Ember Data will create a temporary record representing the item being requested. This record updates the controller via the Ember Observer mechanism (C4). Because the controller's content is bound to the view, the view is updated immediately (C5), even if that data currently holds little information—most likely only the identifier of the item selected in C1.

On the server side, the server-side controller receives the HTTP request (S1), generates or manipulates the data required for the request (S2 and S3), and sends an appropriate HTTP response (S4) back to the client application. This HTTP response contains data in a JSON format. When the item is received in Ember Data (M3), Ember Data updates the model (C3), which triggers steps C4 and C5 and updates both the controller and the view. As you can see, this pattern is built to allow for full dynamicity throughout your application layers, and it works well even with push-style or WebSocket implementations for the data transfer.

Although this approach, when viewed as a whole, is more complicated than the Sun MVC Model 2, it's an approach that enables you, as a developer, to deliver what Ember.js promises: the ability to build ambitious web applications that push the envelope on what's possible on the web.

3.3 Ember Router: Ember.js's take on statecharts

I mentioned earlier that Ember.js enriches the controller layer with a statechart implementation. This statechart, called *Ember Router*, is based loosely on the SproutCore statechart implementation called *Ki*. But unlike Ki, Ember Router is built around the fact that web applications have one important feature that native applications don't—URLs—and it uses this fact to serialize and de-serialize the state of the application into the URL.

> **Learning about statecharts**
>
> A thorough explanation of statecharts is out of scope for this book, but if you're interested in reading more about them, you can read David Harel's scientific paper, *Statecharts: A Visual Formalism for Complex Systems* (1987), http://www.wisdom.weizmann.ac.il/~harel/SCANNED.PAPERS/Statecharts.pdf. Keep in mind that Ember Router is in no way a strict implementation of the statechart described by Harel, but the underlying concepts do apply.

The Ember Router organizes your application's states into a hierarchical structure that uniquely identifies any given route in your application. Because the relationship between the routes is clear, it's possible to transition from any one route to any other route in your application, and you, as a developer, can be certain that the user interface remains consistent. Separating your application into small finite states is an important concept that brings power and versatility to Ember.js applications.

The Ember Router achieves this organization by making each route responsible for setting up everything required for that state to function as intended upon route entry, as well as for tearing down anything that's specific to that route upon route exit. Because the routes are organized in a hierarchical structure, Ember Router ensures that your routes are initialized in the correct order while constructing your application's complex views and templates.

Throughout this book I'll draw routes as boxes. As routes can consist of subroutes, each box might have subroutes drawn in it. Figure 3.7 shows a single route at left and a parent route with one subroute at right.

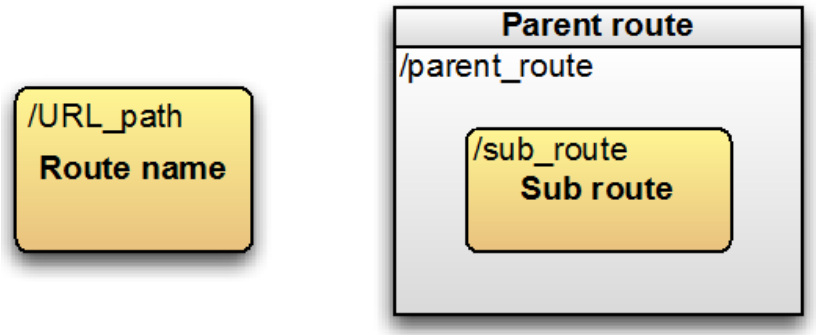


Figure 3.7 A simple route (at left) and a route with a subroute (at right)

Figure 3.8 shows an `Admin` route containing two subroutes, User admin and Payment admin.

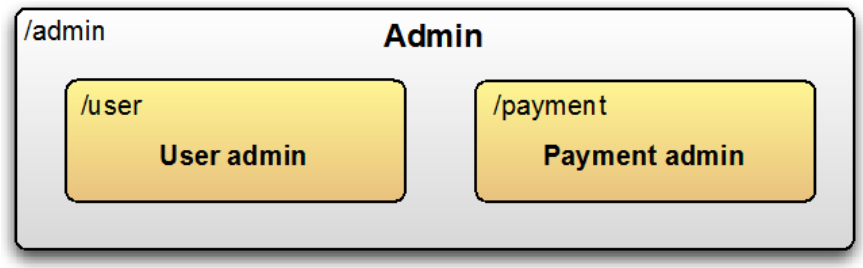


Figure 3.8 Visualizing an Ember route with three connected routes

Each route in the Ember Router has a chain of functions that are called when the user enters and exits a route.

Ember Router has sensible default functionality for these functions, as you'll see throughout this chapter. But, if necessary, you can override these functions. Figure 3.9 shows the route's life cycle and the functions you can override.

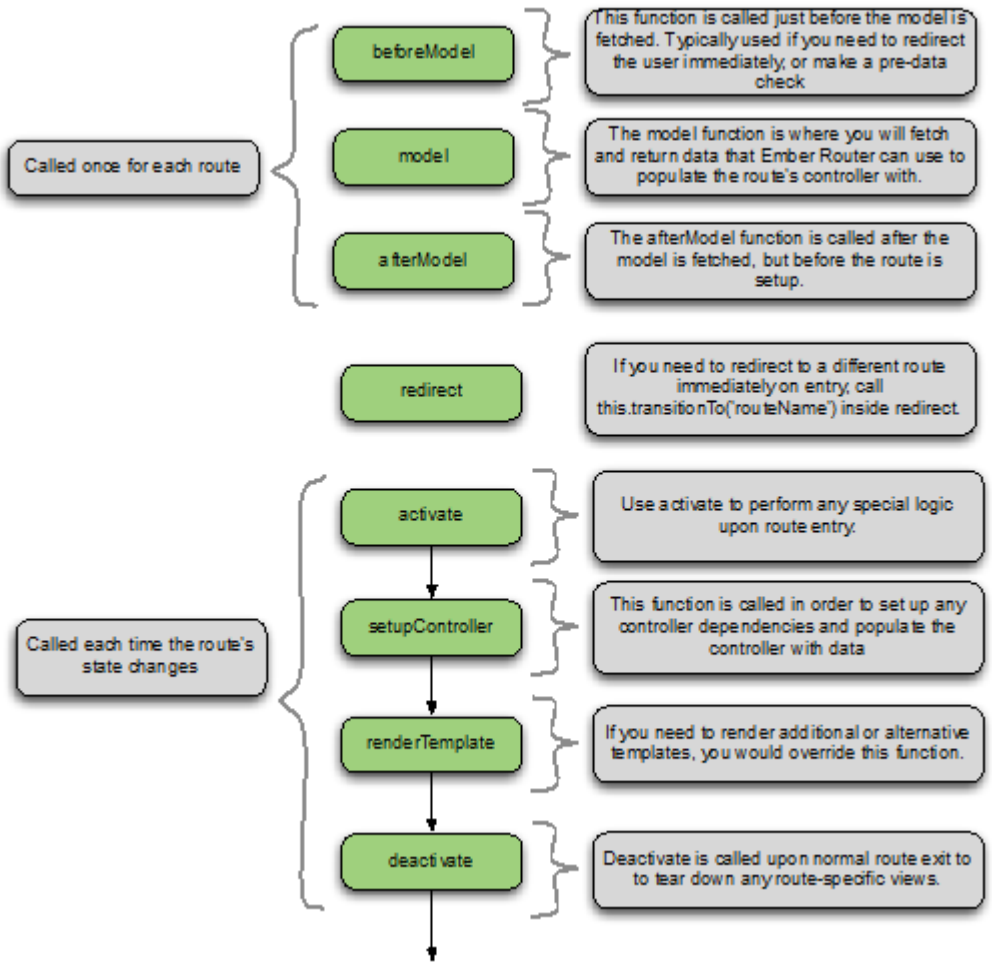


Figure 3.9 The Ember Router life cycle

As I mentioned, Ember Router enriches the controller layer of the application. The final Ember MVC pattern is shown in figure 3.10, which also includes the Ember Router.

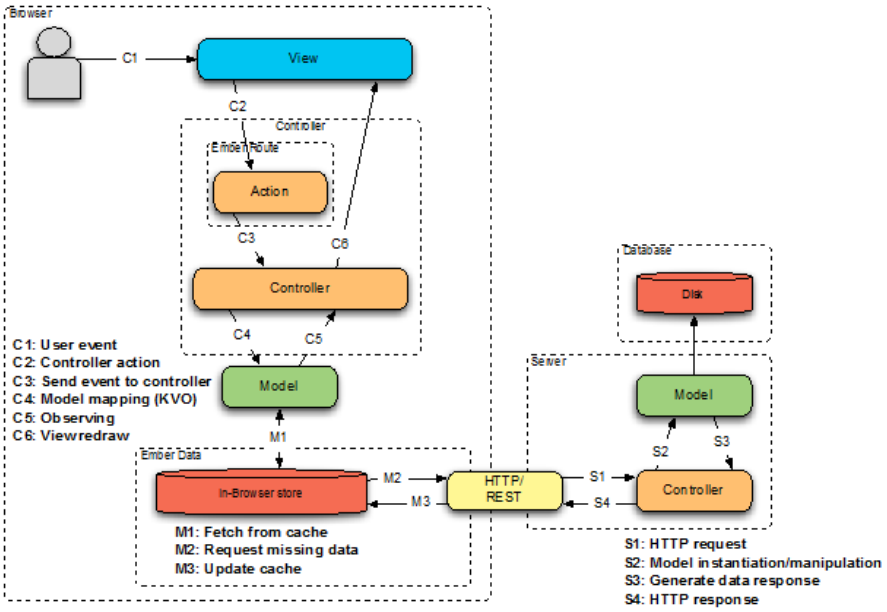


Figure 3.10 Ember MVC model including Ember Router

Having gone through what Ember Router is and how it works, it's time to write an application that utilizes Ember Router.

3.4 Ember.js in Action Blog part 1: the blog index

In this section you'll build the first part of the blog application.

> **NOTE** The complete source code for this section is available as app1.js in either the code source or online at GitHub: https://github.com/joachimhs/Ember.js-in-Action-Source/blob/master/chapter3/blog/js/app/app1.js.

To get started you build up the router for the sample blog application. The blog index has the URL /blog and lists a short summary along with a link to each blog post. When the user clicks an item in the blog index, the application will display that blog post individually via the URL /blog/post/:post_id, where :post_id is the unique identifier of that blog post. When you draw a diagram for the blog route you have the routes shown in figure 3.11.

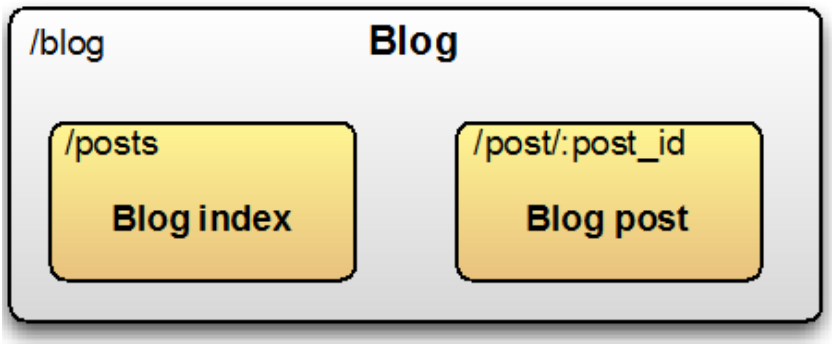


Figure 3.11 Initial application statechart

As you can see from figure 3.11, the website consists of three routes. In addition you have the `about` route with the URL /about and an `index` route with the URL /. Figure 3.12 shows the complete diagram for the Ember.js in Action Blog router.

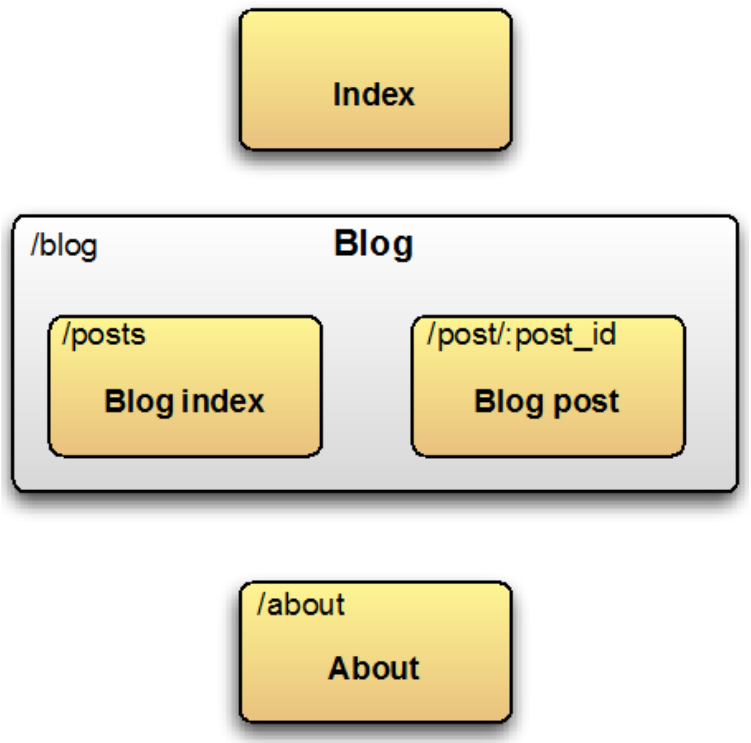


Figure 3.12 Complete Ember.js in Action Blog statechart includes three routes and two subroutes.

Now that we have defined our applications routes and their relationship, lets start building our Blog application.

3.4.1 Getting the blog router started

For this first part you'll build the blog index, which lists the introductory text of each blog post.To you'll implement a number of features in the application:

- An index route that responds to the / URL
- A router with a single route named "blog" that responds to the /blog URL
- An application view and controller
- A blog route, controller, and template
- A means to fetch the blog posts from a JSON file

Running your application

Although you can simply run the Notes application by dragging the index.html file into the browser, I would recommend hosting the application in a proper webserver. You can use the webserver that you are most comfortable with. If you just want to start up a small lightweight webserver that will host the current directory you are in, you can use either the asdf Ruby Gem, or a simple Python script.

If you have Ruby installed, install the asdf Gem by typing `gem install asdf` into your terminal (Mac, Linux) or command prompt (windows). Once the gem is installed, you can host the current directory by executing `asdf –port 8080` in your terminal or command prompt. Once the gem is started, you can navigate to http://localhost:8088/index.html in order to load your Notes application.

If you have Python installed, you can execute `python -m SimpleHTTPServer 8088` in you terminal or command prompt. Once Python is started, your can navigate to http://localhost:8088/index.html in order to load your Notes application.

Let's get started with the implementation of `Blog.Router`, as shown in the following listing.

Listing 3.1 Application router source code

```
Blog.Router = Ember.Router.extend({                                  #A
    location: 'hash'                                                 #B
});

Blog.Router.map(function() {                                         #C
    this.route("index", {path: "/"});                                #D
    this.route("blog", {path: "/blog"});                             #D
});
```

```
Blog.IndexRoute = Ember.Route.extend({                                  #E
    redirect: function() {                                              #E
        this.transitionTo('blog');                                      #E
    }
});

Blog.BlogRoute = Ember.Route.extend({                                   #F
    model: function() {                                                 #F
        return this.store.find('blogPost');
    }
});
```

#A Creates new class, extending from Ember.Router
#B Specifies the use of hash-based URLs (default)
#C: Creates map of routes
#D Defines route to match for the URL / and /blog
#E Defines route for the index and redirects to /blog route
#F Creates route for BlogIndex and specifies model objects for the controller's content

This code includes new concepts that we haven't discussed yet. You start out by specifying the URLs that the application responds to via the `Blog.Router.map` construct (#C). Inside this construct you tell Ember.js which routes belong to which URLs within the application. The `/` URL belongs to the `index` route, but the `/blog` URL belongs to the `blog` route (#D).

Because you specified `location: 'hash'` (which is the default) (#A), Ember Router uses hash-encoded URLs, which means that the URLs are prepended with a hash symbol. A direct URL to a blog post looks like this:

```
/#/blog/post/:post_id
```

You can use the history API instead by specifying `location: 'history'`, in which case the same URL is represented as

```
/blog/post/:post_id
```

Depending on your preference and your back-end server, toggle this option accordingly. Because hash is the default URL pattern, you could omit this declaration of `Blog.Router`, but it's included here to show how this property can be defined.

By specifying the two routes inside `Blog.Router.map`, you tell Ember.js to instantiate three routes, three controllers, and three views using three templates. By default Ember.js uses a standard naming convention to find and instantiate these objects automatically. If you follow these naming conventions, no extra boilerplate code is required to wire everything together. The Ember Router injects the instantiated objects into the Ember Container, which we'll look at near the end of this chapter.

Unless you're doing something specific, you don't need to define any of the objects manually. Ember.js doesn't instantiate them all at once, but Ember Router makes sure that your views are instantiated before they're required. Ember Router also destroys your views when they're no longer needed because the user has navigated away from the route where the view was needed.

Ember Router uses the names of your routes to determine the default names of your routes, controllers, and views. Based on this naming convention, if your route is named `blog`, Ember.js looks up classes named `BlogRoute`, `BlogController` and `BlogView`. In addition,

it assumes that the template used to render the `BlogView` is named `blog`. Unless you have a specific override you don't have to specify these classes in your code. Ember.js and Ember Router assume that the following classes are made available (if not, they are created at runtime):

Route name	Default classes	Template
application	ApplicationRoute ApplicationController ApplicationView	application
index	IndexRoute IndexController IndexView	index
blog	BlogRoute BlogController BlogView	blog

Because you won't have any content for the `index` route in this application, you redirect to the `blog` route (#E). To do this you implement `Blog.IndexRoute` and specify the `redirect` property. In this case you call `this.transitionTo('blog')` to redirect to the `blog` route automatically when the user enters the `index` route.

The blog state is attached to the URL /blog. To tell Ember.js to fetch the blog posts from the server and place them in the `BlogController.model` property, you implement the `model()` function (#F). Inside this function you tell Ember.js to find all posts of type `Blog.BlogPost`, via the Ember Data Store using `this.store.find('blogPost')`. When all the posts are found, Ember.js inserts these posts into the `model` property of this route's controller. As you can see, these few lines of code have a lot going on, so let's break it down to make the process clearer.

`this.store.find('blogPost')` comes from Ember Data. I'll discuss Ember Data in detail in chapter 5, but for now just note that this statement fetches each of the blog posts available from the file `/blogPosts` and stores them in the `model` property of the controller named `Blog.BlogController`.

> **NOTE** Because we don't have any specific functionality that we need to add to the `Blog.BlogController`, we do not need to define it. Ember Router will instantiate a default `Blog.BlogController`, which will hold a list of blog posts. Ember Route uses the return value in the `Blog.BlogRoute`'s model function in order to determine if `Blog.BlogController` should extend `Ember,ObjectController` or a `Ember.ListController`.

Ember Router then injects the `blog` template into the `{{outlet}}` expression inside the `application` template.

At this point, you've implemented enough of the Ember Router to satisfy the requirements for part 1 of the blog application, as shown in figure 3.13.

Figure 3.13 Ember Router as implemented so far

Now that we have the skeleton for out Blog application up and running, its time to add some views and templates in order to give the application some content to its users.

3.4.2 Adding views and templates

Let's build up the minimum parts that the application must include to work with the router shown in figure 3.13. First, let's recap what you'll provide to the Ember Router:

- An `IndexRoute` and a `BlogsRoute`
- An application `template`, an `index` template, and a `blog` template
- A `Blog.BlogPost` model object

You've already seen the two routes, so let's move on to the templates. The following listing shows the implementation of the templates that you need to override to implement the functionality that you want.

Listing 3.2 Application and blog templates

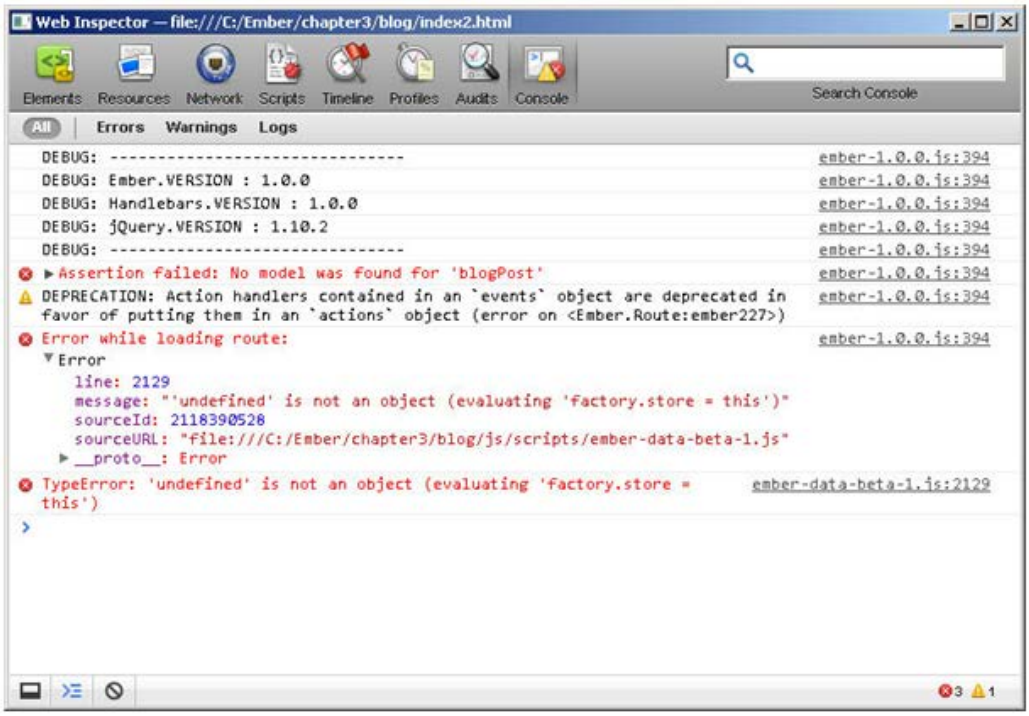
```
<script type="text/x-handlebars" id="application">                    #A
    <div id="mainArea">                                               #B
        <h1>Ember.js in Action Blog</h1>

        {{outlet}}                                                    #C
    </div>
</script>

<script type="text/x-handlebars" id="blog">                           #D
    <div id="blogsArea">
        {{#each controller}}                                          #E
            <h1>{{postTitle}}</h1>
            <div class="postDate">{{postDate}}</div>
            {{postLongIntro}}<br />

            <hr class="blogSeparator"/>
        {{/each}}
```

```
    </div>
</script>
```



#A: Override application template and print out header and {{outlet}} for subroutes
#B: Wrap contents of application in div element
#C: Add {{outlet}} expression into which any subroutes will be rendered
#D: Override blog template
#E: Print introduction for each blog post loaded into BlogController's model

Notice that you haven't created any view classes for either the `application` template or the `blog` template. Because you don't have any specific functionality for these views yet, you continue with the default view that Ember.js creates for you.

Ember Router connects the router to the controller, the controller to the view, and the view to the template, as they're needed by the application.

At this point it should be clear that Ember Router also instantiates your controllers. You might wonder how it knows what type of controller you need for each of your routes. More specifically, how does Ember.js know that your `BlogController` should be an `ArrayController` and not an `ObjectController`? The answer to this lies in the return of the `model()` function of the `BlogRouter`. Because this returns an array, Ember.js knows that it should instantiate the default `BlogController` as an `ArrayController`.

The templates are simplistic. The only dynamic element in them is in the `application` template. Chapter 4 discusses the templates in more detail, but for now think of `{{outlet}}` as the placeholder where the router injects your subroutes. In the `application` route, the router will inject the `blog` template.

You're free to override any of these conventions, but I find it handy that my routes, controllers, and views share a similar nomenclature because I always know which are related based on the name. If I'm currently writing code in the `Blog.BlogController`, I know that the code for the view for that controller is inside `Blog.BlogView`, which in turn is using the template `blog`, and so on.

Next, we need to fetch the blog posts from disk in order to give our application some content

3.4.3 Displaying a list of blog posts

For the Ember Data `Blog.BlogPost` model to work, you need to define the structure of the data, as well as specify a URL from which Ember Data can fetch its data. Listing 3.3 shows the Ember Data model definition for our `Blog.BlogPost` model.

Listing 3.3 `BlogPost` model structure

```
Blog.BlogPost = DS.Model.extend({                    #A
    postTitle: DS.attr('string'),
    postDate: DS.attr('date'),
    postShortIntro: DS.attr('string'),
    postLongIntro: DS.attr('string'),
    postFilename: DS.attr('string'),
    markdown: null                                   #B
});
```

#A: Defines the properties of each BlogPost
#B: This property isn't part of the Ember Data model object

This code introduces several new concepts. The first you might notice is that you extend from `DS.Model`, which is the high-level model class from Ember Data, used to clearly define the intended structure of your data models. As you can see, the `Blog.BlogPost` model defines four properties of type `string` and one of type `date`. All `DS.Model` classes also define an implicit fifth property named `id`. The `id` property serves as the primary key of your data models.

Note also that you have an extra property called `markdown`. It's not strictly necessary to specify the `markdown` property here, but I like to include all the properties that each of my model definitions must have. This makes it clear when looking at the source code for the model that the application uses this property. Because the `markdown` property doesn't have a `DS.attr` type, Ember Data ignores this property when translating the object to and from JSON.

You use the default `DS.RESTAdapter` when you fetch data. Both the format of the URL and the data returned from the server must adhere to a specific pattern. For your purposes,

you map `Blog.BlogPost` to the URL `/blogPosts`, but the `postTitle` property maps to the JSON key `postTitle`, and so on.

Your `Blog.BlogPost` object is serialized and deserialized from and to the JSON data as shown in the following listing.

Listing 3.4 JSON structure for the `BlogPost` model

```
{
    "id": "2012-05-05-Ember_tree",
    "postTitle": "Implementing a Custom Tree Model in Ember",
    "postDate": "2012-05-05",
    "postShortIntro": "Explaining the Client-side MVC model …",
    "postLongIntro": "We have had a number of very good …"
}
```

You'll use the standard `DS.RESTAdapter` throughout this chapter. Chapter 6 will discuss how you can create your own custom adapter if you're talking to a back end that has implemented a different data structure. Chapter 5 will discuss Ember Data and the `RESTAdapter` in detail.

Because you're receiving a list of blog posts, you need to wrap your single blog post JSON object into an array and store the contents in a file named blogPosts, the content of which is shown in the following listing. Notice that you add an `id` property to the JSON as well. Ember Data uses `id` implicitly, even though it's not specified in the `Blog.BlogPost` model object.

Listing 3.5 Contents of the blogs.json file

```
[
    {
        "id": "2012-05-05-Ember_tree",
        "postTitle": "Implementing a Custom Tree Model in Ember",
        "postDate": "2012-05-05",
        "postShortIntro": "Explaining the Client-side MVC model …",
        "postLongIntro": "We have had a number of very good …"
    }
]
```

Now you can list the blog posts you had in your `blogPost.` If you refresh your application you should be greeted with an application that looks like figure 3.14.

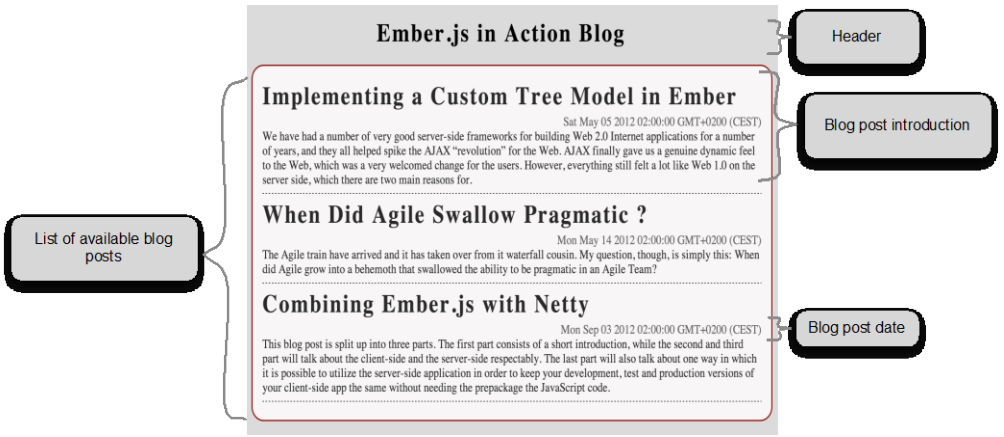


Figure 3.14 Updated Ember.js in Action Blog for part 1

If you look at the posted date for each of the posts, you'll notice that the date isn't easy to read. You need to format the standard JavaScript date output to something more readable by humans. There are multiple ways to do this. You can bring in a third-party date-formatting library and build a custom Handlebars.js expression to handle the conversion (more about custom expressions in chapter 4). But for your purposes create a computed property on the `BlogPost` model that returns a properly formatted date. The following listing shows the updated `BlogPost` model.

Listing 3.6 Updated `BlogPost` model

```
Blog.BlogPost = DS.Model.extend({
    postTitle: DS.attr('string'),
    postDate: DS.attr('date'),                                    #A
    postShortIntro: DS.attr('string'),
    postLongIntro: DS.attr('string'),
    postFilename: DS.attr('string'),
    markdown: null

    formattedDate: function() {                                   #B
        if (this.get('postDate')) {
            return this.get('postDate').getUTCDay()
                + "/" + (this.get('postDate').getUTCMonth() + 1)
                + "/" + this.get('postDate').getUTCFullYear();
        }

        return '';
    }.property('postDate')                                        #C
});
```

#A: Change from string to date
#B: Add computed property to format the date in human-readable way.
#C: Computed property is calculated based on the postDate property and will update accordingly

You've added a computed property to the `Blog.BlogPost` model object called `formattedDate`. This computed property updates whenever the `Blog.BlogPost.postDate` property changes, and it prints the date using the formatting, "`mm/dd/yyyy`". Computed properties are a powerful mechanism in Ember.js that lets you add complex computational properties to functions. In addition, computed properties are `bindable` and auto-updatable. In this case, whenever the `postDate` property changes on the `BlogPost` model, the return value for the `formattedDate` also updates. Because the computed property is bindable, any template using the `formattedDate` property will be updated as if `formattedDate` was a normal property. This is an extremely powerful concept.

The only thing you need to update in the template is a single expression. To implement this, you introduce a new view and template for the content of the blog post, named `blog`. The following listing shows the new `blog` template.

Listing 3.7 Updated templates

```
<script type="text/x-handlebars" id="blog">
    <div id="blogsArea">
        {{#each controller}}
            <h1>{{postTitle}}</h1>
            <div class="postDate">{{formattedDate}}</div>                #A
            {{postLongIntro}}<br />

            <hr class="blogSeperator"/>
        {{/each}}
    </div>
</script>
```

#A: Use the computed property formattedDate to print a correctly formatted date

You might argue that a library like moment.js would be better suited for date formatting. Although I agree with that statement, bringing in an extra library is unnecessary in the context of explaining computed properties.

This concludes the first part of the blog application. Next, you'll add links to individual blog posts as well as display them.

3.5 Ember.js in Action Blog part 2: adding the blog post route

In this section you build the second part of the blog application.

> **NOTE** The complete source code for this section is available as "app2.js" in the code source, or online at GitHub https://github.com/joachimhs/Ember.js-in-Action-Source/blob/master/chapter3/blog/js/app/app2.js.

The next part involves connecting the blog index to the individual posts. Let's start by updating the `blogPosts` file with some extra entries, as shown in the following listing. I've also updated the contents with real blog posts taken from my company's blog to have real data to work with.

Listing 3.8 Updated blogPosts file

```
[
    {
        "id": "2012-05-05-Ember_tree",
        "postTitle": "Implementing a Custom Tree Model in Ember",
        "postDate": "2012-05-05",
        "postShortIntro": "Explaining the Client-side MVC model and how …",
        "postLongIntro": "We have had a number of very good …"
    },
    {
        "id": "2012-05-14-when_did_agile_swallow_pragmatic",
        "postTitle": "When Did Agile Swallow Pragmatic ?",
        "postDate": "2012-05-14",
        "postShortIntro": "My question, is simply this: When did Agile … ",
        "postLongIntro": "The Agile train have arrived and it has …"
    },
    {
        "id": "2012-09-03-Combining_Ember_js_with_Netty",
        "postTitle": "Combining Ember.js with Netty",
        "postDate": "2012-09-03",
        "postShortIntro": "Combining Ember.js with Netty ",
        "postLongIntro": "This blog post is split up into three parts…"
    }
]
```

To navigate from the list of blog posts to an individual post you first change the definition of the blog from a route to a resource

> **NOTE** For each route in your application, Ember Router creates a default template associated with that route. A route that can have subroutes is defined as a resource. Each of your resources will automatically have an index route associated with it. This means that our blog resource will automatically get a blog.index route inserted as a subroute, which belongs to the blog/ URL.

. Next, you add two subroutes to the blog resource, called `index` and `post`. After the routes are in place, you add a link in the `blog.index` template to transition out of the `blog.index` route and into the `blog.post` route. You also need to make sure that the URL is updated with a dynamic ID that identifies a unique blog post so that users can bookmark a single blog post. But first, let's review the router diagram, shown in figure 3.15.

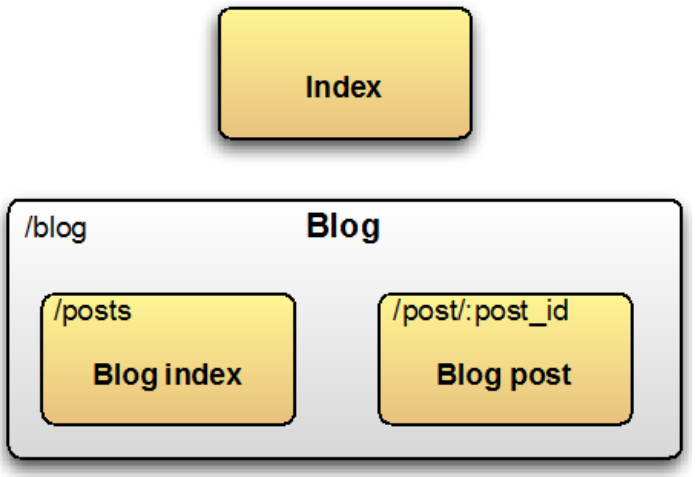


Figure 3.15 Adding the blog index and blog post routes.

As you can see you've added two new routes. You need the `blog.index` routeso that you can navigate to both the `/blog/` URL and the `/blog/post/:post_id` URL. You also add an action to transition from the `blog.index` route to the `blog.post` route. The updated `Blog.Router` is shown in the following listing.

Listing 3.9 Adding the blog index and blog post states

```
Blog.Router.map(function() {
    this.route("index", {path: "/"});
    this.resource("blog", {path: "/blog"}, function() {                #A
        this.route("index", {path: '/posts'});                         #B
        this.route("post", {path: '/post/:blog_post_id'});             #C
    });
});
```

#A: Adds and indexes route to the /blog route
#B: Moves the blogIndex route so that it becomes a subroute of the blog's route
#C: Adds a subroute to the blog's route called blogPost

We already explained the new routes, but you should note two important things. First, because the `blog` route is now defined as a resource, the Ember Router automatically creates an `index` route as a direct child of the `blog` route. But because you want to attach the custom URL /posts to the `blog.index` route, you must define it here.

Second, note the URL that you attached to the `blog.post` route, particularly the `:blog_post_id` part of the URL. This part tells the Ember Router that you want to have a dynamic part attached to this route. In this case, you want the URL to reflect the blog post that the user is currently viewing. Although this lets your users bookmark and share direct links to individual blog posts, it also specifies to the `blog.post` route exactly which blog post to load and pass on to the `BlogPostController`.

Now that you've added the new routes to the Ember Router definition, let's look at defining the implementation for each of the routes. The following listing shows the updated and new routes.

Listing 3.10 Updated and new routes

```
Blog.BlogIndexRoute = Ember.Route.extend({                              #A
    model: function() {
        return this.store.find('blogPost');
    }
});

Blog.BlogPostRoute = Ember.Route.extend({                               #B
    model: function(blogPost) {
        return this.store.find('blogPost', blogPost.blog_post_id);      #C
    }
});
```

#A: Rename Blog.BlogRoute to Blog.BlogIndexRoute
#B: Add a BlogPostRoute for the blog.route route
#C: Use the dynamic part of the URL to populate the BlogPostController

As you can see, the `Blog.BlogRoute` is renamed to `Blog.BlogIndexRoute`. In addition, you added a new `Blog.BlogPostRoute`. By overriding the `model()` function of the `BlogPostRoute`, you're able to find the right blog post from Ember Data and pass it on to the `model` property of the `BlogPostController`. Also note that the object that gets passed into the `BlogPostRoute`'s `model()` function has a single property, with a name that matches the name that you gave the dynamic part of the route definition in listing 3.9.

Next, you update the `template` for the blog index so the user can select a blog post, as shown in the following listing.

Listing 3.11 Addinglink from `blog.index` template to `blog.post` template

```
<script type="text/x-handlebars" id="blog/index">                       #A
    <div id="blogsArea">
        {{#each controller}}
            <h1>{{postTitle}}</h1>
            <div class="postDate">{{formattedDate}}</div>
            {{postLongIntro}}<br /><br />
            {{#linkTo "blog.post" this}}Full Article ->{{/linkTo}}      #B
            <hr class="blogSeperator"/>
        {{/each}}
    </div>
</script>
```

#A: Renames blog template to blog.index
#B: Adds link to the blog.post route via the linkTo expression

This template has two new details. First, you've renamed the template from `blog` to `blog/index`. As the user transfers from the blog index to view an individual blog post you remove the blog index from the document object model and replace it with the `blog.post` route. Because you're no longer implementing a template for the blog route, Ember Router will

create a default template for you. This default template will only contain a single `{{outlet}}`, which will render the template for its active subroute.

The second detail you've added is a link from the `blog.index` route to the `blog.post` route, using the `{{linkTo}}` expression. This expression takes either one or two parameters, in which the first parameter is the route that you want to link to and the second parameter is a context that you want to pass into the route you're linking to.

Now that you've added a link to the `blog.post` route, it's time to implement the template for this route. You add this template to the index2.html file, as shown in the following listing.

Listing 3.12 Adding the blog/post template

```
<script type="text/x-handlebars" id="blog/post">                    #A
    <div id="blogPostArea">                                         #B
        <div class="postDate">{{formattedDate}}</div>
        <br />{{#linkTo "blog.index"}}&lt; back{{/linkTo}}          #C
        {{markdown}}                                                #D
        <br />{{#linkTo "blog.index"}}&lt; back{{/linkTo}}          #C
    </div>
</script>
```

#A: Create a new handlebars template with the id blog/post
#B: Wrap the contents of the template inside a div for CSS styling
#C: Link back to the blog.index route with a {{#linkTo}} expression
#D: Display the blog post content

No new concepts are added to the `blog.post` template. But notice that you're printing out the `markdown` property of the model. If you remember the `BlogPost` model definition, you may remember that you initialize this property to `null`. In addition, you haven't told the application what goes into this `markdown` property.

When the user enters the `blog.post` route, you load the Markdown-based content from the server and place the loaded content into this `markdown` property. You can do this in multiple ways, but for this purpose you implement an observer on the `Blog.BlogPostController` that fetches the blog post content and assigns it to the `markdown` property. The following listing shows the definition of the controller.

Listing 3.13 The `Blog.BlogPostController`

```
Blog.BlogPostController = Ember.ObjectController.extend({           #A
    contentObserver: function() {                                   #B
        if (this.get('content')) {
            var page = this.get('content');
            var id = page.get('id');                                #C

            $.get("/posts/" + id + ".md", function(data) {          #D
                var converter = new Showdown.converter();           #E
                page.set('markdown',
                  new Handlebars.SafeString(converter.makeHtml(data))); #F
```

```
            }, "text")
                .error(function() {
                    page.set('markdown',  "Unable to find specified page");
                    //TODO: Navigate to 404 state
                });

        }
    }.observes('content')                                                    #G
});
```

#A: BlogPostController extends ObjectController as it proxies single blog post object
#B: Create an observer-function that will listen to any changes to controller's content property
#C: Get id-property of the controller's content
#D: Fetch blog post content from /posts/id.markdown URL
#E: Initialize Showdown converter to convert markdown to HTML
#F: Escape HTML to add converted HTML as HTML and not text to the controller
#G: Mark function as observer using observes suffix

The code has some things worth noting for the `Blog.BlogPostController`. You first create a function called `contentObserver`. By adding the suffix `observes('content')` to the function you tell Ember.js that this function will be executed whenever the controller's `content`-property changes. Whenever the user enters the `blog.post` route to view a different blog post, the `contentObserver` function is triggered and updates the model's `markdown` property.

You're using `showdown.js` to convert the `markdown` that's returned from the server into HTML before you assign the generated HTML to the model's `markdown`-property.

You could implement this functionality in a number of ways. I chose this method to show how observers work, as well as how to combine Ember Data with standard Ajax.

Now, you refresh the application and select a blog post to navigate to the `blog.post` route and view the blog post. Figure 3.16 shows what the application looks like.

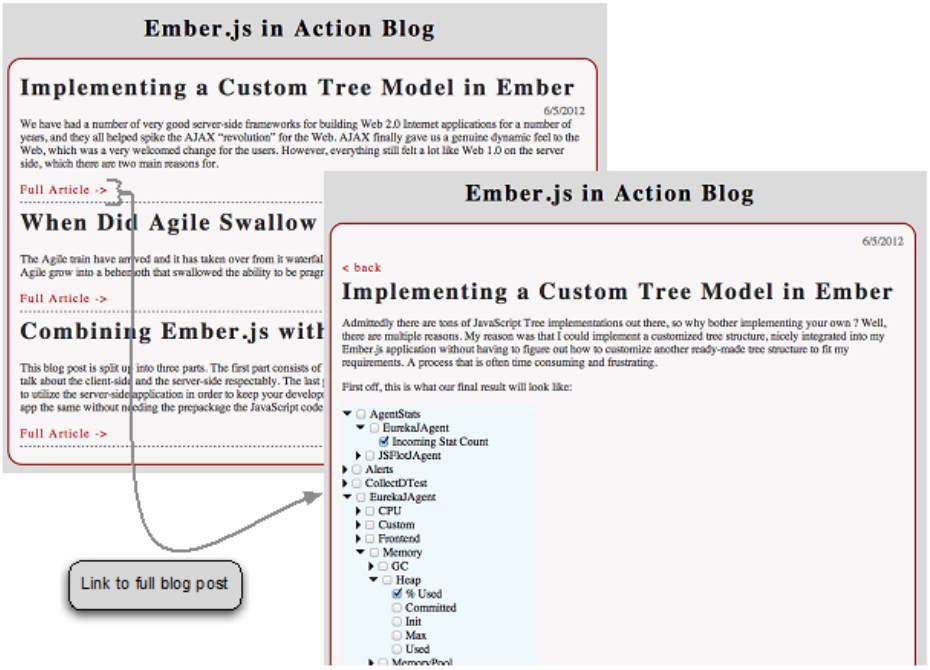


Figure 3.16 Select a blog post and transition into the blog.post route

As you can see, you can click on the Full Article link to transition from the `blog.index` route to the `blog.post` route. In addition, you can click on the "back" link to transition back into the `blog.index` route.

You do need to fix one thing before moving on. If you hit "refresh" while inside the `blog.post` route, Ember.js shows an "Error while loading route" exception because you haven't loaded the blog post (they're loaded inside the `blog.index` route). You also haven't supplied the server with a way to provide the blog application with individual blog posts, which Ember Data expects to load from the URL /blogPosts/:post_id when it enters the `model` function of the `Blog.BlogPost` route.

You can fix this in one of two ways:

- Supply the server with the individual blog posts through the URL /blogPosts/:post_id
- Load all the blog posts higher up in the route hierarchy

Although supplying the server side with a way to respond to the URL `/blogPost/:post_id` would be trivial in a real server, it's not straightforward for the simple, file-based server implementation. You solve this issue by loading all the blog posts directly in the `Blog.BlogRoute` instead of in the `Blog.BlogIndex` route. But because you also want access to all the blog posts inside the `Blog.BlogIndex` route (you're listing the blog index

from the `blog.index` template), you have to ensure that the blog posts are available in both routes.

You can achieve this in multiple ways. I'll use the approach that requires the least number of changes to the rest of the application now. When we disuss dependency injection and the container at the end of this chapter, I'll show the other approaches.

To load the blog posts higher up in the route hierarchy, you change the implementation that you have for the `Blog.BlogIndexRoute` route, while also overriding the default implementation of the `Blog.BlogRoute`. The following listing shows the updated routes.

Listing 3.14 Updated `Blog.BlogIndexRoute` and `Blog.BlogRoute` routes

```
Blog.BlogRoute = Ember.Route.extend({                                  #A
    model: function() {                                                #B
        return this.store.find('blogPost');                            #C
    }
});

Blog.BlogIndexRoute = Ember.Route.extend({
    model: function() {
        return this.modelFor('blog');                                  #D
    }
});
```

#A: Introduce a definition of the Blog.BlogRoute
#B: Override the model-function to load the blog posts
#C: Load all the blog posts
#D: Ensure that the model function of Blog.BlogIndexRoute returns the same data loaded into Blog.BlogRoute

If you look closely at the updated routes, you'll notice that you've effectively moved the model definition that you had in the `Blog.BlogIndexRoute` into the `Blog.BlogRoute`. To ensure that you populate the `Blog.BlogIndexController` with the same models that you're loading into the `Blog.BlogController`, you use the routes' `modelFor()`-function. This function takes the name of a route that's higher up in the route hierarchy and returns the model that's loaded into that route.

Incidentally, you could achieve the same result by having the `model()` function of the `Blog.BlogIndexRoute` return `this.store.find('blogPost')`, as you had before. This option would work in this situation because Ember Data implements an identity map for the data that it has loaded from the server side. We'll discuss Ember Data in detail in chapter 5.

At this point, you should be able to refresh the application and see that you can now access the application directly at the `blog.post` route, while also supporting the user refreshing the application at this route.

The only thing missing from our application now is to implement the About page. I'll leave you to implement this page as an exercise, but I've also provided an implementation for you in the blog posts source code.

I mentioned that we'll discuss the Ember Container as well as how dependency injection works in Ember.js, so let's shift gears and discuss this central concept.

3.6 Dependency injection and using the Ember Container

You can connect different parts of your application in two ways. If you want to connect controllers you can use the controllers' `needs` property, or you can register and inject objects via dependency injection.

3.6.1 Connecting controllers through the needs property

If you only want to connect two controllers you use the `needs` property of your controller. In the blog application the `blog.index` route is dependent on the data that's loaded into the `blog` route. You solved this earlier by using the `modelFor()` function inside the `model()` function of the `Blog.BlogIndexRoute`.

Another approach is to connect the `Blog.BlogIndexController` with the `Blog.BlogController` using the `needs` property of the `Blog.BlogIndexController`. The following listing shows the updated routes and controllers.

Listing 3.15 Using the needs property to connect controllers

```
Blog.BlogController = Ember.ArrayController.extend({                    #A

});

Blog.BlogIndexController =  Ember.ObjectController.extend({             #B
    needs: ['blog']                                                      #C
});
```

#A: Extends ArrayController
#B: Creates BlogIndexController as an ObjectController
#C: Specifies that this controller needs the BlogController

You might be wondering what the `needs` property does for your application and how Ember.js uses it to connect the two controllers.

When the `BlogIndexController` is initialized, Ember.js will look at the `needs`-property of the controller to determine whether the controller is dependent on other controllers. If the controller specifies at least one dependent controller, Ember.js will look up each of the controllers in the Ember Container and inject them into the controller's `controllers`-property.

To get the `BlogController` from the `BlogIndexController`, you access it through the controller's `controllers` property. If you're inside the `BlogIndexController`, you can get the controller via a call to `this.get('controllers.blog')`.

You also must update the `blog.index` template to reflect this change. The following listing shows the updated template.

Listing 3.16 Updated `blog.index` template

```
<script type="text/x-handlebars" id="blog/index">
    <div id="blogsArea">
        {{#each controllers.blog}}                                       #A
            ...
        {{/each}}
```

```
    </div>
</script>
```

#A: Uses controllers.blog to access the content of the blog controller

I've omitted the contents of the `{{each}}` expression, because it's the same as before. As you can see, the only change you've made to the template is inside the `{{each}}` expression. The logic here is the same as inside the controller. Because you've added the `BlogController` instance to the `BlogIndexController`'s `controller.blog` property, you can access it directly via the `controllers.blog` expression inside the `blog.index` template.

Although this method uses the Ember Container in the background its only use case is to connect controllers. If you want to connect other objects, you need to take a close look at the Ember Container.

3.6.2 Connecting objects via the Ember Container

The concept of dependency injection is a whole book in itself. In short, the Ember Container allows you to assign objects to common names. After an object is registered it's possible to inject it into properties on other objects that are also registered into the Ember Container.

Ember Data is an excellent example of a use case that uses the Ember Container to its advantage. The following listing shows the initialization of Ember Data.

Listing 3.17 Ember Data's initialization and the Ember Container

```
Ember.onLoad('Ember.Application', function(Application) {
  Application.initializer({
    name: "store",                                                    #A

    initialize: function(container, application) {
      application.register('store:main', application.Store || DS.Store);#B
      application.register('serializer:_default', DS.JSONSerializer);
      application.register('serializer:_rest', DS.RESTSerializer);
      application.register('adapter:_rest', DS.RESTAdapter);

      container.lookup('store:main');
    }
  });

...

  Application.initializer({
    name: "injectStore",                                              #C

    initialize: function(container, application) {
      application.inject('controller', 'store', 'store:main');        #D
      application.inject('route', 'store', 'store:main');             #D
      application.inject('serializer', 'store', 'store:main');        #D
      application.inject('dataAdapter', 'store', 'store:main');       #D
    }
  });
```

#A: Creates an application initializer named "store"
#B: Registers the store into the container's store:main property
#C: Creates an application initializer named injectStore

#D: Injects the container's store:main property into the store property of the corresponding object

I don't expect you to follow the logic behind the application initializers here. The point of this example is to give you a sense of the power behind the Ember Container. At the top of the listing you call `application.register()`, which takes two arguments. The first argument is a string that represents the unique name that the object you're registering will be tied to. The second argument is the object or class that you want to register with the container. In this case you register a `DS.Store` class into the container as `store:main`.

Then, in the second initializer you use `application.inject` to inject one stored property into a property on another registered object. You ensure that whatever you've registered as `store:main` will be injected into the store property of every `controller`, `route`, `serializer` and `dataAdapter` within your application. This is a big deal, and it's what allows you to call `this.store.find('blogPost')` from within the `model()` functions you've seen in use throughout this chapter.

After something is registered with the container, you'll be able to get these objects via the non-public `App.__container__.lookup` function. You shouldn't rely on this within your own application, but it can be quite useful if you need to debug your application via the browser console. The following listing shows a few examples to give you an idea of when this might come in handy.

Listing 3.18 Retrieving registered objects from the container

```
Blog.__container__.lookup('store:main').find('blogPost')          #A
Blog.__container__.lookup('controller:blog')
    .get('model.length')                                          #B
Blog.__container__.lookup('controller:blogPost')
    .get('postTitle')                                             #C
Blog.__container__.lookup('controller:blogPost')
    .set('model.markdown', 'Test')                                #D
```

#A: Find all blog posts from the loaded Ember Data Store
#B: Get the number of blog posts loaded into the BlogController
#C: Get the postTitle of the selected blog post
#D: Set the markdown property to "TEST" for the selected blog post

The examples only show a few instances of when being able to interact with objects that are registered in the Ember Container can be useful. Still, they do show how to both retrieve and manipulate the objects that are loaded. Go ahead, load up the blog application, and play with the examples provided. It will quickly give you an insight into the power that Ember.js provides you as a web application developer, and it will help you understand how it's connected inside your application.

3.7 Summary

This chapter represents what will become the heart of your Ember.js application. Your Ember Router implementation represents the way your application is glued together as well as what routes your application can be in and how the user can navigate between these routes. The

router also defines the way your controllers are bound together as well as the way data flows between your controllers and, in extension, between your views.

The benefits of moving the user-specific business logic out to the client, in terms of both the total architecture and the user experience, are clear. Letting the client and the server applications do what they do best allows you to build highly scalable web applications with features that rival native applications, wrapped up with the most powerful distribution channel there is—the web.

Through the sample blog application I've shown how you can structure your application into logical routes and how a complete Ember.js application is connected.

There's one major piece missing before we can move on to part 2 of the book, so without further ado, let's dive into Ember.js's preferred template engine, Handlebars.js.

4

Automatic updating templates with Handlebars.js

This chapter covers

- Understanding why you need templates in the first place
- Working with Handlebars.js expressions
- Using simple and complex expressions
- Understanding the relationship between Ember.js and Handlebars.js
- Creating your own custom expressions

Ember.js doesn't include a default template library, and you're free to use your favorite JavaScript library. But because the same people who are behind Ember.js are also behind Handlebars.js, Handlebars.js is an especially nice fit for Ember.js applications. That being said, Handlebars.js also has all the features that you're looking for in a solid template library.

Handlebars.js is based on Mustache, which is a logic-less template library that exists for many programming languages, including JavaScript, Python, Java, Ruby, and most likely your favorite language.

In this chapter, you'll start by getting a clear understanding of what a template is and why you should be excited about using one, before moving on to the features that Handlebars.js provides. In the second half of the chapter, you'll look at how Ember.js extends the standard Handlebars.js features to provide automatically updated templates for your application. This chapter should leave you with a comprehensive understanding of the template features that are built into Handlebars.js and Ember.js.

4.1 What's in a template?

A Handlebars template is regular static HTML markup interspersed with dynamic elements called *expressions*. The template library replaces these expressions at runtime to bring you dynamic web applications that update in real time, whenever your underlying data changes.

> **NOTE** The top-most template in Ember.js is called application. By default this template only includes a single Handlebars.js expression; `{{outlet}}`. If you need to include anything special in the top-most template, you need to override the application template. Just remember though, that your application template needs to have an `{{outlet}}` somewhere, otherwise none of your other templates will be rendered.

Multiple types of template libraries exist. Traditionally, template libraries reside on a server, where they're generally used to combine a model and a template to generate a view, as shown in figure 4.1.

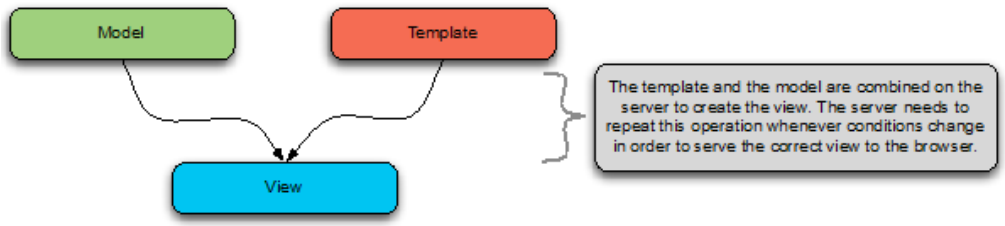


Figure 4.1 The traditional server-side template model

Because Handlebars.js lives on the client side and because Ember.js has a rich MVC pattern, Ember.js is able to replace this one-time template compilation with a much more powerful dynamic pattern, as shown in figure 4.2.

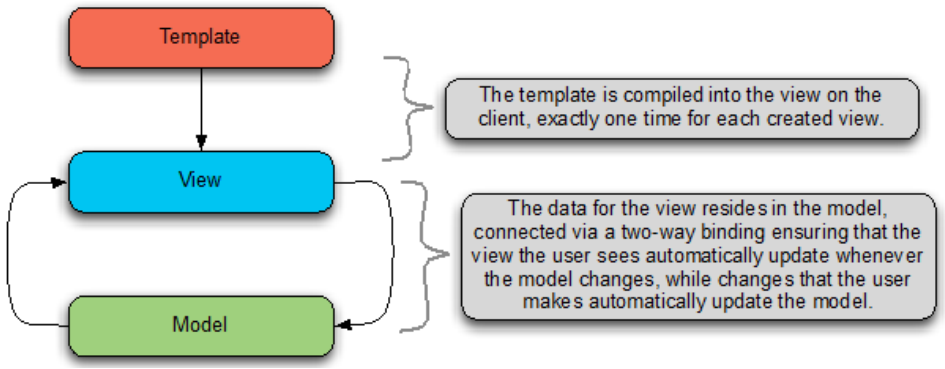


Figure 4.2 The Ember.js + Handlebars.js template solution

Handlebars.js, as well as most other template libraries, identifies its expressions by enclosing them in double curly braces. Expressions can either be simple expressions representing dynamic values, or block expressions that also contain logic. You'll start by getting a clear understanding of how simple expressions look and how they work.

4.1.1 Simple expressions

Throughout this chapter, you'll use a simple book-cataloguing system that could be used to keep track of books in your home library. For each book, you want to see a few pieces of basic information: the book's title, its author(s), and a short description of the book.

Simple expressions are identifiers that tell Handlebars.js which variable to use to replace the template contents at runtime. Consider the code in the following listing.

Listing 4.1 A simple expression

```
<h1>{{title}}</h1>                                    #A
```

#A: Replaces title at runtime

When the preceding template is rendered, Handlebars.js looks up the value of the `title` variable in the current context and replaces the `{{title}}` expression with the correct value.

You may also use dot-separated paths inside your Handlebars.js expressions. If you have a model object named `Book` that has three properties—`title`, `author`, and `text`—you may use the Handlebars template shown in the following listing to display the book's details.

Listing 4.2 Using dot-separated paths in expressions

```
<h1>{{book.title}}</h1>                               #A
<p>By: {{book.author}}<br />                          #B
    {{book.text}}</p>                                 #C
```

#A: Displays book's title
#B: Displays book's author
#C: Displays book's text

As you can see, you can use dot-separated paths to access properties in an object in Handlebars. You can chain these properties as deep as your object's references go, but whenever you find yourself needing more than three parts in an expression, you'll most likely want to start splitting your templates into smaller, more-specific templates. Also, whenever you have more than three parts in an expression, it might be time to rethink your application structure to see if you're missing a route or a controller.

You'll learn how to split your templates into smaller templates with Ember.js and Handlebars later in the chapter, but first let's look at the other type of Handlebars expression, the block expression.

4.1.2 Block expressions

A *block expression* has not only a value but also a body that can contain plain markup, simple expressions, or even other block expressions. Handlebars.js identifies a block expression by its prepending pound, or hash, symbol (#). The block expression ends by prepending a backslash

(/) to the ending block helper. Anything that's listed between the start and end tag is part of the block expression and builds up the block expression's body. An example of the structure of the each block helper is shown in figure 4.3.

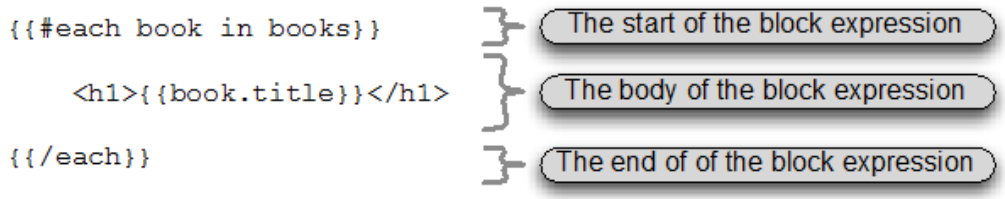


Figure 4.3 The each block expression

A block expression also can have a different context than its enclosing template. Let's continue with our book catalog analogy, and assume that you are using the context shown in the following listing.

Listing 4.3 The context

```
{
    "title": "Books",
    "books": [
        { "title": "Ember.js in Action",
          "author": "Joachim Haagen Skeie",
          "text": "A thorough overview of the Ember.js Framework"
        },
        { "title": "Secret of the JavaScript Ninja",
          "author": "John Resig and Bear Bibeault",
          "text": "A book about mastering modern JavaScript development"
        }
    ]
}
```

You can now create a Handlebars template that lists the details of each book inside the books array, as shown in the following listing.

Listing 4.4 Listing each book detail by using block expressions

```
<h1>{{title}}</h1>                                          #A
{{#each books}}                                             #B
    <div class="book">
        <h1>{{this.title}}</h1>                             #C
        <p>By: {{this.author}}<br />                        #D
        {{this.text}}</p>                                   #E
    </div>
{{/each}}                                                   #F
```

#A: Displays page title
#B: Iterates over books array
#C: Displays book's title
#D: Displays book's author
#E: Displays book's text

#F: Closes the each block expression

When looking at the code in listing 4.4, you should notice a couple of things. First, you use the built-in `each` block helper to iterate over the `books` array. Notice how you can use the keyword `this` inside the block expression to identify the book you're currently at in the iteration over the `books` array. The rest of the template is the same as before, except that you make sure to close the `each` block at the end of the template.

When this template is rendered with the context from listing 4.3, it results in the following markup.

Listing 4.5 The result of the each block expression

```
<h1>Books</h1>                                                    #A
<div class="book">                                                #B
    <h1>Ember.js in Action</h1>
    <p>By: Joachim Haagen Skeie<br/>
    A thorough overview of the Ember.js Framework</p>
</div>
<div class="book">                                                #C
    <h1> Secret of the JavaScript Ninja </h1>
    <p>By: John Resig and Bear Bibeault<br/>
    A book about mastering modern JavaScript development</p>
</div>
```

#A: Main title of page
#B: Contents of first book
#C: Contents of second book

I mentioned that a block expression can have a different context than its containing block or template. With the `each` block expression, you can introduce a context variable that Handlebars will use to identify each object in the `books` array. The updated code is shown next.

Listing 4.6 The updated each block

```
<h1>{{title}}</h1>
{{#each book in books}}                                           #A
    <div class="book">
        <h1>{{book.title}}</h1>                                   #B
        <p>By: {{book.author}}<br />                              #C
        {{book.text}}</p>                                  #D
    </div>
{{/each}}
```

#A: Creates context-specific variable named books to hold current item in the iteration over books array
#B: Displays book's title
#C: Displays book's author
#D: Displays book's text

By using the `{{#each book in books}}` expression, you can create a new variable called `book` at the same time as declaring the `each` block expression. You can then refer to the `book` variable inside the block statement instead of relying on using `this`.

Handlebars.js has several built-in block expressions. The next section presents each of them and shows examples of how they're used.

4.2 Built-in block expressions

Most of the general block expressions that you're used to from programming languages are built into Handlebars.js and include the following:

- `each`
- `if`
- `if-else`
- `unless`
- `with`
- `comments`

You've just seen how to use the `each` block expression to iterate over an array of items, to generate a template for each item inside the array, so we'll skip the explanation of the `each` block helper here and dive straight into the `if` block expression.

4.2.1 The if block expression

Whenever you have templates containing options that control whether parts of the template are rendered, you'll most likely use an `if` block to express this logic. If you want to render a book only if it has an author assigned to it, for example, you could use the template shown in the following listing.

Listing 4.7 The `if` block expression

```
{{#if book.author}}                                          #A
    <h1>{{book.title}}</h1>
    <p>By: {{book.title}}<br />{{book.text}}</p>
{{/if}}
```

#A: Conditionally renders the book's details

The `if` block expression takes a single parameter, the value that it will evaluate to determine whether the body of the expression will be rendered. In this case, the book's details won't be rendered if `book.author` evaluates to `null`, `undefined`, `0`, `false`, or any other *falsy* value.

But what if you want to include a simple error message in the template for the case when the book's author isn't defined yet? Luckily, Handlebars.js supports {{else}} as well. The following listing shows how `if-else` is defined.

Listing 4.8 The `if-else` block expression

```
{{#if book.author}}
    <h1>{{book.title}}</h1>
    <p>By: {{book.title}}<br />{{book.text}}</p>
{{else}}                                                     #A
    <p>{{book.title}} does not have an assigned author</p>
{{/if}}
```

#A: Completes if-else expression

You're using `{{else}}` to specify the section that's added to the rendered template if `book.author` returns a falsy result. Notice that the `{{else}}` expression doesn't start with a hash symbol because that expression is part of the `{{if}}` expression.

4.2.2 The unless block expression

Sometimes you're interested in rendering a block only if its condition is falsy. You use the `if` block expression whenever you want to say, "If this is true, I want to…." In contrast, you use the `unless` block expression if the opposite is the case and you want to say, "If this is false, I want to…."

Instead of having to specify an empty `if` section to an `if-else` block expression, Handlebars.js includes the `unless` block expression, shown in the next listing.

Listing 4.9 The `unless` block expression

```
{{#unless book.bookIsReleased}}                                    #A
    <p>{{book.title}} is not released yet.</p>
{{/unless}}
```

#A: Specifies only the else part of if-else expression

Here you've added the property `bookIsReleased` to your books. In this case, you want to include a notice in the rendered template only if a book has yet to be released. The `unless` block is a nice fit in these situations.

4.2.3 The with block expression

Even though Handlebars.js supports using paths in its expressions, it can sometimes be handy to be able to shorten a long path (for example, `book.author.address.postcode`) by using the `with` block expression to shift the context for a subsection of the template. Consider the following code, which is an updated version of listing 4.7.

Listing 4.10 Using `with` to shift the context for the book subsection

```
<h1>{{title}}</h1>
{{#each book in books}}
    {{#with book}}                                                 #A
        <div class="book">
            <h1>{{title}}</h1>
            <p>By: {{author}}<br />
            {{text}}</p>
        </div>
    {{/with}}
{{/each}}
```

#A: Shifts context of its body

As you can see, using the `with` block expression shortens the paths for each of the expressions inside the `with` block. This can prove handy when you need to use complex and long paths in your expressions.

4.2.4 Handlbars.js comments

Because any logic you put inside your templates will be part of your Handlebars expressions, you might sometimes want to annotate your code with comments to explain more in depth what's happening. Handlebars.js uses the `{{! }}` notation to indicate a comment. An important note, though, is that comments won't be part of the generated markup. If you'd like to include them in the generated HTML, you should instead use standard HTML comments. The following listing shows how comments are used.

Listing 4.11 Using Handlebars comments

```
<div class="comments">
    {{! A Handlebars comment that won't be part of the rendered markup}}
    <!—- An HTML comment that will be part of the rendered markup -->
</div>
```

So far you've looked at Handlebars.js simple and complex expressions, as well as the built-in expressions that are part of the library. You've seen that each expression has a context and that the library uses this context to generate each template's output. But you haven't seen how Handlebars fits into Ember.js, how Ember.js controls the context of each of the templates, and how to split complex templates into smaller, more-manageable templates. So let's dive into these issues in the next section.

4.3 Using Handlebars.js with Ember.js

Ember.js extends Handlebars.js and enriches it with the powerful features that you've come to expect from your Ember.js applications. After you tell Ember.js to render a Handlebars.js template, you can rest assured that Ember.js will keep your view up-to-date whenever your application models change, without you having to specifically implement any logic to perform these updates.

To know what part of the DOM tree to update when your application models change, Ember.js injects `metamorph` tags before and after each expression's content. You'll review `Metamorph` later in the chapter.

You'll look at how Ember.js extends Handlebars.js, which new expressions it adds, as well as how Ember.js views are tied into Handlebars.js. But first, let's look closer at how to define templates in an Ember.js application.

You'll truly learn to love writing applications in this way.

4.3.1 Defining templates inside index.html

Handlebars.js supports defining your templates inside index.html, which can serve as a convenient and easy-to-get-going alternative. Keep in mind, though, that placing all your templates inside this one file quickly becomes inconvenient.

But if you do choose to define your templates inside your index.html file, your templates will be inside a `script` tag with the type `text/x-handlebars`. You can define your application template inside your `body` tag by creating an anonymous `script` tag, as shown in the following listing.

Listing 4.12 Creating the application template

```
<html>
    <head><title>My Book Catalogue Page</title></head>
    <body>
        <script type="text/x-handlebars">                                    #A
            Welcome, {{user.fullName}}!
        </script>
    </body>
</html>
```

#A: Defines application template inside body tag

The application template is displayed on the page by your application's router. Refer to chapter 3 to read more about Ember Router.

Obviously, having only one application template won't do you any good. Each of your additional templates needs to be defined inside the `head` element and have a unique name applied to it via either the `data-template-name` attribute or the `id` attribute, as shown in the following listing.

Listing 4.13 Creating the books template

```
<html>
    <head>
        <title>My Book Catalogue Page</title>

        <script type="text/x-handlebars" id="books">                         #A
            <div class="books">Book Catalogue</div>
        </script>
    </head>
    <body>
        <script type="text/x-handlebars>
            Welcome, {{user.fullName}}!
        </script>
    </body>
</html>
```

#A: Defines named template in head element

You'll most likely be using some sort of build tool to manage all your application's assets, including precompiling your Handlebars templates and making them available to your Ember.js application. Build tools are covered in more detail in chapter 11.

Because this approach quickly becomes less than practical, throughout this book you'll define your templates in separate *.hbs files that you'll bring into your application via either build tools or Ajax calls. You have a third option, though, which is to define your templates directly in the `Ember.TEMPLATES` hash.

4.3.2 Defining templates directly in the Ember.TEMPLATES hash

When an Ember.js application initializes, it reads through the index.html file and places any templates that it finds inside the `Ember.TEMPLATES` hash. You can compile your templates directly into this hash. This is an approach that's OK during development, but it quickly gets

ugly as you constantly have to manage string concatenation. The following code shows how you could name and compile the templates from listing 4.13 into the `Ember.TEMPLATES` hash.

Listing 4.14 Compiling your templates into `Ember.TEMPLATES`

```
Ember.TEMPLATES['application'] = Ember.Handlebars.compile('' +
    'Welcome, {{user.fullName}}!'
);                                                                    #A

Ember.TEMPLATES['books'] = Ember.Handlebars.compile('' +
    '<div class=”books”>' +
        'Book Catalogue' +
    '</div>'
);                                                                    #B
```

#A: Defines application template
#B: Defines book's template

This approach has two advantages: it's easier to split your templates into multiple files, and it keeps your templates out of your index.html file. If you aren't using any build tools, you have to constantly juggle your single and double quotes. If you want to keep your templates nice and formatted, you also need to use string concatenation to combine the code lines in your templates.

> **NOTE** The drawbacks of using this approach should be evident. In the long run you would be better off either defining your templates inside index.html, or as separate *.hbs files that you can compile in your build tools. This approach is covered in chapter 11.

After defining your templates, you can create views that will use those templates to define their rendered content, which is covered in the next section.

4.3.3 Creating Handlebars template-backed Ember.js views

Ember.js views are created by either extending or instantiating a class of type `Ember.View`. In this case, you want to create a new view that uses the template books from listing 4.14, so let's create a view that uses this template. The code is shown in the following listing.

Listing 4.15 Creating a template-backed view

```
App.BookView = Ember.View.extend({                                    #A
    templateName: 'book'                                              #B
});
```

#A: Creates new view that extends Ember.View
#B: Defines view's template

You start by creating a new view, `App.BookView`, that extends `Ember.View` via the `extends` keyword. Further, you specify that this view use a template whose name is defined in the `templateName` property.

So far, so good, but Ember.js also lets you define your templates inline, directly inside your views, instead of referring to an external template via its name.

Whenever you have views that you want to use in multiple parts of your application, you might want to create a reusable custom view. Whenever I create views that are reusable, especially if they are reusable across applications, I tend to inline the template. The following listing shows how `App.BookView` could be written using an inline template via the `template` property.

Listing 4.16 Creating a view with an inline template

```
App.BoookView = Ember.View.Extend({
    template: Ember.Handlebars.compile('' +                          #A
        '<div class=”books”>' +
            'Book Catalogue' +
        '</div>')
})
```

#A: Creates inline template

Notice that you need to call `Ember.Handlebars.compile` to compile your Handlebars.js template into the `template` property of the view. I generally use inline templates when I am creating custom reusable views and the templates are rather simple and small. If my template stretches over many lines of code and has multiple block expressions, I tend to specify them separately from the view and refer to the template via the `templateName` property.

I mentioned earlier that Ember.js provides additional Handlebars expressions, so let’s go ahead and look at those.

4.4 Ember.js-provided Handlebars.js expressions

Ember.js extends Handlebars.js with additional expressions that you’ll most likely use often throughout your application. Ember.js provides the following additional expressions:

- `view`
- `bind-attr`
- `action`
- `outlet`
- `unbound`
- `partial`
- `linkTo`
- `render`
- `control`
- `input`
- `textarea`
- `yield`

In this section, you’ll review each of the Ember.js-provided expressions and see how they’re used.

4.4.1 The view expression

As you might have guessed by its name, the `view` expression is used to add a view into a Handlebars.js template and it's often used to inject self-contained views into a template. Let's create a view for our book catalogue example called `App.BookDetailsView` and inject this view into the application template from listing 4.14. The combined result is shown in the next listing.

Listing 4.17 Injecting a view with the `view` expression

```
Ember.TEMPLATES['bookDetails'] = Ember.Handlebars.compile('' +          #A
    '<div class=”book”>' +
        '<h1>{{title}}</h1>' +
        '<p>By: {{author}}<br />' +
        '{{text}}</p>' +
    '</div>'
);

Ember.TEMPLATES['books'] = Ember.Handlebars.compile('' +                #B
    '{{#each book in books}} ' +
        '{{view App.BookDetailsView valueBinding=”book”}}' +
    '{{/each}}'
);


App.BookDetailsView = Ember.View.extend({                               #C
    templateName: 'bookDetails'
});

Ember.TEMPLATES['application'] = Ember.Handlebars.compile('' +          #D
    '<h1>Welcome, {{user.fullName}}!</h1>' +
    '{{view Ember.View templateName=”books”}}'                          #E
);
```

#A: Adds bookDetails template into Ember.TEMPLATES hash
#B: Adds books template to Ember.TEMPLATES hash
#C: Creates App.BookDetailsView to use bookDetails template
#D: Adds application template to Ember.TEMPLATES hash
#E: Injects anonymous view using books template into application template

Notice here that you're compiling your templates directly into the `Ember.TEMPLATES` hash, to demonstrate how this works. As you can see, you're creating two new templates, `books` and `bookDetails`. Together these two views represent the list of books as well as the details for each book. You're also creating a new view, `App.BookDetailsView`, that uses the `bookDetails` template. Finally, you're using an anonymous view inside the application template to render the `books` template.

Even though it's possible to create anonymous views in this manner, my experience is that creating anonymous views works for only small and simple views. For anything more complex than the preceding view, you'd be much better off creating a proper `Ember.View` instance, like the `App.BookDetailsView`, and using that instead. Nonetheless, sometimes you only need to bring up a simple view to render a template, and in this case anonymous views will serve that purpose quite well. After all, it's fairly easy to refactor the anonymous view later. Another

approach to render a template without having to define a view for it is to use the `{{partial}}` expression, which we'll discuss later.

4.4.2 The bind-attr expression

Whenever Ember.js renders an expression, it injects `metamorph` `script` tags into your code to be able to re-render each expression when the underlying model changes. `Metamorph` works almost anywhere in your HTML code, except when you bind your model to HTML element attributes. To amend this situation, Ember.js also includes the `bindAttr` expression to be able to use bindings to also update your HTML element attributes. Consider the HTML code in the following listing, which specifies the `src`, `height`, and `width` attributes of the HTML `img` tag.

Listing 4.18 Binding HTML tag attributes to a backing model

```
<script type="text/x-handlebars" id=" image-template">
    <img {{bind-attr src="imageUrl"}}                          #A
        {{bind-attr height="imageHeight"}}                     #B
        {bind-attr width="imageWidth"}} />                     #C
</script>
```

#A: Binds src attribute
#B: Binds width attribute
#C: Binds height attribute

Whenever you want to bind your model object to an HTML tag attribute, you need to use the `bind-attr` expression, which is always a simple expression. The expression takes one argument, which specifies the HTML tag attribute to render. The value of the argument specifies which property to bind to in the current context.

USING BIND-ATTR WITH A BOOLEAN VALUE

You can also use `bind-attr` with a Boolean value. The Boolean result specifies whether the attribute will be included in the rendered markup. Consider the code in the following listing.

Listing 4.19 Using `bindAttr` with a Boolean value

```
<script type="text/x-handlebars" id=" image-template">
    <input type="checkbox" {{bind-attr disabled="canEdit"}} />      #A
</script>
```

#A: Toggles tag attribute

If the result of `canEdit` resolves to `true`, Ember.js renders the template with the disabled attribute included: `<input type="checkbox" disabled />`. If `canEdit` resolves to `false`, Ember.js omits it: `<input type="checkbox" />`.

Metamorph

For Ember.js to know what DOM elements to update whenever your application models change, it injects special `script` tags into your DOM just before and just after your

Handlebars expressions. These tags have a type defined as `text/x-placeholder` and indicate the area where Ember.js will replace the contents.

For each expression in your templates, Ember.js surrounds the generated HTML markup with `script` tags, as shown here:

```
<script id="metamorph-30-start" type="text/x-placeholder"></script>
<!—- The Contents of the convMarkdown expression -->
<script id="metamorph-30-end" type="text/x-placeholder"></script>
```

Ember.js does all the bookkeeping necessary to know which metamorph script it will use to update your views whenever your model changes. Most of the time, you won't notice that Ember.js injects these metamorph tags, but you still need to know that they exist because they do add elements to the DOM tree that can affect your CSS styling, for instance.

4.4.3 The action expression

The `action` expression is, as its name indicates, used to fire DOM actions on HTML elements. The action is forwarded into the template's `target`, which will most likely be the current route's controller. The `action` expression takes three arguments: a name, a context, and a set of options. Any event triggered via the `action` expression will have `preventDefault()` called on it.

Consider the following code, which creates a link with an appropriate action.

Listing 4.20 Using the `action` expression

```
<script type=”text/x-handlebars” id=”bookDetails”>
    <div class=”book”>
        <h1>{{title}}</h1>
        <p>
            By: {{author}}<br />
            {{text}}<br />
            <button {{action “editBookDetail” this}}>Edit Book</button>  #A
        </p>
    </div>
</script>
```

#A: Triggers action when button is clicked

This template renders a standard HTML button tag for each of your books. When the user clicks this button, the action `editBookDetail` fires. The first parameter to the `action` expression is the name of the action. Ember.js uses the action name to trigger an action on the template's `target`, and it expects to find a function with the exact same name as the action name. If you haven't supplied a target, Ember.js assumes you want to send the event to the current route's controller.

The second argument is the context (data), which it will supply to the invoked function—in this case, the current `book`.

You can supply various options to the `action` expression:

- The DOM event type
- A target
- A context

SPECIFYING THE DOM EVENT TYPE

By default, the DOM event type used by the `action` expression is the `click` event. You can override this by specifying the `on` option with a supported event name. `Ember.View` specifies 28 supported event names, which are grouped in five categories, as shown in table 4.1.

Table 4.1 The DOM event types associated with Ember Views

Mouse events	Keyboard events	Touch events	Form events	HTML5 drag-and-drop events
click	keyDown	touchStart	submit	dragStart
doubleClick	keyUp	touchMove	change	drag
focusIn	keyPress	touchEnd	focusIn	dragEnter
focusOut		touchCancel	focusOut	dragLeave
mouseEnter			input	drop
mouseLeave				dragEnd
mouseUp				
mouseDown				
mouseMove				
contextMenu				

If you want to specify that the Edit Book button's action triggers on double-click, you specify `on="doubleClick"` as the option argument to the `action` expression. The following listing provides an example.

Listing 4.21 Specifying a DOM event type

```
<script type="text/x-handlebars" id="bookDetails">
    <div class="book">
        <h1>{{title}}</h1>
        <p>
            By: {{author}}<br />
            {{text}}<br />
            <button {{action editBookDetail this on="doubleClick"}}>Edit
Book</button>                                                          #A
        </p>
    </div>
</script>
```

#A: Specifies DOM event type

SPECIFYING A TARGET

If you're using Ember Router in your application (refer to chapter 3 for a detailed overview of Ember Router), the default target for your `action` expression will always be the current

route's controller. If you haven't defined your action inside this controller, the action will bubble up to the current route and up the route hierarchy until it finds the action.

If you need your action to go anywhere else, you must manually override the `target` option, as shown in the following listing.

Listing 4.22 Overriding the `target` option of the `action` expression

```
<script type="text/x-handlebars" id="bookDetails">
    <div class="book">
        <h1>{{title}}</h1>
        <p>
            By: {{author}}<br />
            {{text}}<br />
            <button {{action editBookDetail this
                target="App.editBookController"}}>
                Edit Book
            </button>                                                    #A
        </p>
    </div>
</script>
```

#A: Overrides default target

When this target is invoked by clicking the link, Ember.js tries to invoke the `editBookDetail` function of the `App.editBookController` instance.

You can also specify a path relative to the current view by using `target="view"`.

SPECIFYING A CONTEXT

If you specify a context as the second argument to the `action` expression, you can pass data along to the invoked action method. Considering the `action` expressions from listing 4.20 through 4.22, the `editBookDetail`method will be called with a single parameter containing the `book` object as its `context`.

The following listing shows how you can use the context that you pass in via the `action` expression.

Listing 4.23 Retrieving the `action` context

```
App.EditBookController = Ember.Route.extend({
    actions: {
        editBookDetails: function(book) {                                #A
            console.log(book.get('name');                                #B
        }
    }
});
```

#A: Specification of action method with context provided

#B: Retrieving book via first parameter book, and printing its name to console

4.4.4 The outlet expression

The `{{outlet}}` expression is simply a placeholder in your templates where your controllers can inject a view. Whenever the current controller's `view` property changes, Ember.js makes sure to replace the outlet with the new view. Using Ember Router, you update the controller's

`view` property via the `renderTemplate` method. The following listing shows how you can use `renderTemplate` to update the outlet.

Listing 4.24 Using `connectOutlet` to update the outlet

```
Ember.TEMPLATES['application'] = Ember.Handlebars.compile('' +
    '{{outlet books}}' +                                              #A
    '{{outlet selectedBook}}'                                         #B
);

App.BooksRoute = Ember.Route.extend({
    renderTemplate: function() {
        this.render('books', { outlet: 'books'});                     #C

        var selectedBookController = this.controllerFor('selectedBook');

        this.render('selectedBook', {                                 #D
            outlet: 'selectedBook',
            controller: selectedBookController
        });
    }
});
```

#A: Adds outlet for list of books
#B: Adds outlet for selected book
#C: Renders list of books into book's outlet
#D: Renders selected book into selectedBook outlet

In the preceding code, you start by defining two outlets in your application template: one for a left-hand menu containing the books and one for the selected book. Then in your `App.BooksRoute` route, you use the `renderTemplate` function to render your views into the outlets via `this.render()`.

This example is a bit contrived. Normally you'd solve this problem via Ember Router, having one route named `books` and one route named `books.book`. That way, you could simply use the `{{outlet}}` expression without any arguments in the book's template to tell it where to render the `books.book` template.

4.4.5 The unbound expression

The `unbound` expression allows you to output a variable to the template without using the binding capabilities of Ember.js. Note, though, that the contents of this expression won't be automatically updated whenever your model objects change. The following listing shows how to use the `unbound` expression.

Listing 4.25 Using the unbound expression

```
<script type="text/x-handlebars" id="book">
    '<div>{{unbound book.name}}</div>'                                #A
</script>
```

#A: Specifies one-time evaluated variable

4.4.6 The partial expression

The `partial` expression allows you to render another template in the current template. This allows you to easily reuse your templates. Consider the following code.

Listing 4.26 Using the partial expression

```
<script type="text/x-handlebars" id="books">
    {{#each book in books}}
        {{partial "book"}}                                          #A
    {{/each}}
</script>

<script type="text/x-handlebars" id="book">                         #B
    Title: {{title}}<br />
    Author: {{author}}<br />
</script>
```

#A: Injects another template
#B: Book template you want to inject

Using the `partial` expression, you can easily embed another template into the current template. But I still find it more convenient and cleaner to create actual routes that represent these templates.

4.4.7 The linkTo expression

The `linkTo` expression is used whenever you want to create an HTML link that will take the user from one route to the next. Consider the following router, taken from chapter 3.

Listing 4.27 The blog router

```
Blog.Router.map(function() {
    this.resource('index', {path: '/'}, function() {
        this.resource('blog', {path: '/blogs"}, function() {
            this.resource('posts', {path: '/posts'}, function() {
                this.route('index, {path: '/'};                     #A
                this.route('post', {path: '/:blog_post_id'});       #B
            })
        })
    });
    this.route('about');
});
```

#A: The posts.index route you want to link from
#B: The posts.post route you want to link to

Whenever you're in the `blog.index` route, you want to provide the user with a link that can be clicked for each blog post and that will fetch the selected blog post content and transition the user to the `blog.post` route. This can be done via the `linkTo` expression, as shown in the following listing.

Listing 4.28 Using the `linkTo` expression

```
<script type="text/x-handlebars" id="blogIndex">
    {{#each blog in blogs}}
```

```
        {{#linkTo "posts.post" blog}}View Post{{/linkTo}}              #A
    {{/each}}
</script>
```

#A: Adds link to the posts.post route

As you can see, you're simply using the `linkTo` expression with two parameters. The first parameter is the name of the route to transition to when the link is clicked, and the second parameter is the context, which you'll pass into the route's `setupController` function.

Up until now, you've seen how to use the built-in expressions from Handlebars.js, as well as the additional expressions that Ember.js includes. You might be wondering how you can create your own expressions, which is what you'll look at in the upcoming section about Handlebars helpers.

4.4.8 The render expression

Whereas the `{{partial}}` expression is used to render a template using the current context, the `{{render}}` expression is used to render a template that's backed by its own singleton controller and view. As a result, if you render a template named header, you also instantiate a new singleton `HeaderController` and a `HeaderView`. This is important if you want to have a template that isn't tied directly into the current controller context. But note that the `{{render}}` expression will render the template belonging to the current route, meaning that actions inside the template will bubble up through your route hierarchy.

The following listing shows an example of how the `{{render}}` expression can be used.

Listing 4.29 The `render` expression

```
<script type="text/x-handlebars" id="books">
    {{render header}}                                                  #A
    <ul>
        {{#each book in books}}
            <li> {{partial "bookDetails"}} </li>                        #B
        {{/each}}
    </ul>
</script>
```

#A: Renders header template backed by singleton HeaderController and HeaderView
#B: Renders template bookDetails via {{partial}} expression

Using the `{{render}}` expression enables you to render the same template in multiple parts of your application. Because the controller and the view that the render template will be connected to are singletons, had you rendered multiple header templates, each of these would share the same controller and thus the same data.

Sometimes, though, you don't want your templates to share their controller and view, which is where the `{{control}}` expression comes in.

4.4.9 The control expression

Unlike the `{{render}}` expression, the `{{control}}` expression is backed by its own controller and view. Consider the following code.

Listing 4.30 The `control` expression

```
<script type="text/x-handlebars" id="books">
    {{render header}}                                          #A
    <ul>
        {{#each book in books}}
            {{control "bookDetails" book}}                     #B
        {{/each}}
    </ul>
</script>
```

#A: Renders header template backed by singleton HeaderController and HeaderView
#B: Renders template bookDetails via {{control}} expression

This code is similar to listing 4.29, with one significant difference. Instead of using the `{{partial}}` expression to render the `bookDetails` template, you're now using the `{{control}}` expression. As a result, each `bookDetails` template that you render will have its own `BookDetailsController` and `BookDetailsView`. In addition, you can inject the current book into the `{{control}}` expression, which then serves as the context for each of the templates.

4.4.10 The input and textarea expressions

I am bundling the `{{input}}` and `{{textarea}}` expressions together because they serve a similar purpose. The `{{input}}` expression simply renders an HTML `input` tag into the DOM, and the `{{textarea}}` expression renders an HTML `textarea` tag into the DOM.

An `{{input}}` without a type or a `type="text"` renders as a standard HTML `textfield`. The `{{input}}` expression has the following attributes:

- `type`
- `value`
- `size`
- `name`
- `pattern`
- `placeholder`
- `disabled`
- `maxlength`
- `tabindex`

The `{{textarea}}` expression has the following attributes:

- `value`
- `name`
- `rows`
- `cols`
- `placeholder`

- disabled
- maxlength
- tabindex

When the attributes on either the {{input}} or the {{textarea}} expression are set with quotes, their values are directly inserted into the DOM as strings. If the attributes are set without quotes, they are bound to the current context. The following listing shows examples of both.

Listing 4.31 Using the input and textarea expressions

```
App.AwesomeController = Ember.Controller.extend({
    userCanEdit: true,
    placeholder: "Enter a value",
    fieldLength: 20,
    defaultValue: "Food"
});
<script type="text/x-handlebars" id="awesome">
    {{input type="text" value="Groceries" size="25"}} <br/>

    {{input type="text" value=defaultValue size=fieldLength}} <br/>
    {{textarea value="My text area text"}} <br/>

    {{textarea value=defaultValue}}

</script>
```

#A: Renders textfield with values outputted as string directly
#B: Renders textfield with attributes bound to controller
#C: Renders textarea with values outputted as strings directly
#D: Renders textarea with values bound to controller

The difference between bound and unbound attributes should be clear to you by now. The way these two expressions are set up gives you full control of how these expressions are rendered onscreen and which attributes are bound to the context and which aren't.

4.4.11 The yield expression

The {{yield}} expression is of limited use in Ember.js, and is applicable only for views that have a layout attached to them, and for Ember.js components. If you have a view that uses a layout, you use the {{yield}} expression to tell the view's layout where to render the view's template. In this case, you can think of the {{yield}} expression as doing the same as the {{outlet}} expression does for routes. The following listing shows how to use the {{yield}} expression from within a view that has a layout.

Listing 4.32 The yield expression

```
App.MyLayoutView = Ember.View.extend({
    layout: Ember.Handlebars.compile('' +                                    #A
        '<div class=”layoutClass”>{{yield}}</div>'),                          #B
    templateName: 'viewsTemplate'
});
```

#A: Creates layout this view will use
#B: Defines where to render view's template inside layout

As you can see, `App.MyLayoutView` does, in essence, have two templates. One template controls the layout, and you use the `{{yield}}` expression to tell the layout where to draw the template into the layout template.

Even though Ember.js does pack a large number of expressions that you can use in your application, sometimes you need to use an expression that's not built in. Luckily, Ember.js allows you to create your own expressions as well.

4.5 Creating your own expressions

Internally, Handlebars.js calls expressions *helpers*. You'd use the `registerHelper` method to register your own custom helpers, which can then be invoked from any of your Handlebars.js templates. In the following listing, you'll create and register a new helper called `convMarkdown` that uses the `Showdown` library to convert text from Markdown format to HTML.

Listing 4.33 Creating a helper to convert from Markdown to HTML

```
Ember.Handlebars.registerHelper('convMarkdown',
        function(value, options) {


    var converter = new Showdown.converter();

    return new Handlebars.SafeString(converter.makeHtml(value));

});
```

#A: Registers new expression convMarkdown
#B: Creates new Showdown converter
#C: Returns converted markup via Handlebars.SafeString

You start by registering a new helper via the `Handlebars.registerHelper` method, passing in the name that you want your new expression to have. In the callback function, you start by getting the value of the passed-in function name before using the contents of this value to convert its Markdown markup to HTML. But Handlebars will escape any HTML markup contained in its returned value. To return the actual HTML markup, you return a new `Handlebars.SafeString` object instead. You can now use this new expression in any of your applications Handlebars templates, as shown in the following listing.

Listing 4.34 Using a custom expression

```
{{convMarkdown markdownProperty}}                                    #A
```

#A: Using newly created convMarkdown expression

4.6 Summary

This chapter serves as a summary of the built-in template library Handlebars.js. Even though you're free to use your favorite template library, Handlebars.js will most likely have the functionality you're after in your web application. If you require additional logic, Handlebars.js makes it easy to create custom expressions that provide your application with the specific logic that your application needs.

We've reviewed the expressions built into Handlebars.js and shown an example of how you can use each of them. As Ember.js extends the core Handlebars.js features, we've also shown how you can use the Ember.js-specific expressions in your application as well as how you can create your own.

This chapter concludes part 1 of this book. As you move along to part 2, you'll be introduced to a real open-source Ember.js application, Montric. Montric appears in examples throughout the rest of this book and is used to explain the finer details of how you can interact with your server side, build complex custom components, as well as assemble and test your Ember.js application.

5

Bringing home the bacon – Interfacing with the server side using Ember Data

This chapter covers

- Introducing Ember Data and core concepts
- Using Ember Data models and model associations
- Using the built-in RESTAdapter to interface with your server
- Customizing the RESTAdapter

Distilled down to a single statement, Ember Data is the Object Relational Mapping framework for the web. Ember Data lets you interact with your server side in a straightforward and intuitive manner, while keeping the required code to a minimum. If you can also customize the format that your server provides its data in, you'll be up and running on your client side with a minimal amount of code.

But not everyone can adapt the backend application to fit with the standard REST-based API that Ember Data expects out of the box. For these situations, Ember Data offers pluggable adapter and serializer APIs so that your Ember.js application can understand your specific server-side data APIs.

This chapter starts with the basic building blocks and patterns that make up the core of Ember Data before it delves into how you can use the built-in RESTAdapter and RESTSerializer to get data in and out of your Ember.js application. It wraps up with an overview of how you can implement custom adapters and serializers so that Ember Data works with the server-side API that you already have in place.

Figure 5.1 shows the parts of the Ember.js ecosystem this chapter examines—ember-application, ember-views, ember-states, ember-routing, and container.

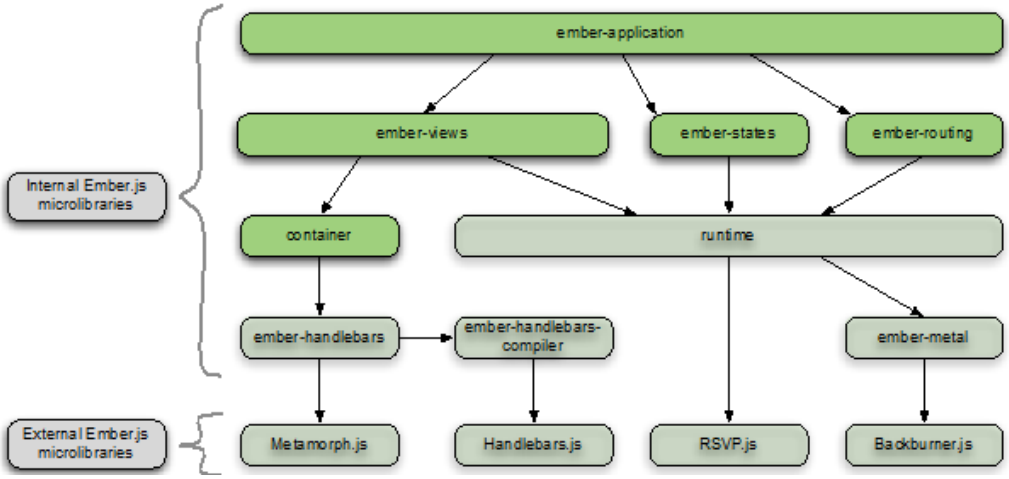


Figure 5.1 The parts of Ember.js you'll be working on in this chapter

NOTE At the time of writing, the newest version of Ember Data is 1.0.0-beta.2 . As a result, this chapter, and indeed this book, covers Ember Data beta 2 throughout.

Now, let's roll up our sleeves and get going.

5.1 Using Ember Data as an application cache

Ember Data is effectively a caching layer inside your web application. Whenever you load data from your server to your client, you populate this cache with data. To maintain a rigorous structure to your cache, you need to define model objects for your models. Before we go into how the Ember Data cache works, let's quickly go over what an Ember model object is.

5.1.1 Defining Ember Data models

Ember Data model objects serve as class definitions for your data and tell Ember Data what attributes each model object has and what type each attribute has.

The following listing shows the `MainMenu` object from the Montric project.

Listing 5.1 The `MainMenu` object

```
Montric.MainMenu = DS.Model.extend({                          #A
    name: DS.attr('string'),                                  #B
    nodeType: DS.attr('string'),
    parent: DS.belongsTo('Montric.MainMenu'),                 #C
    children: DS.hasMany('Montric.MainMenu'),                 #D
    chart: DS.belongsTo('Montric.ChartModel'),                #E
```

```
    isSelected: false,                                              #F
    isExpanded: false,
});
```

#A: Extends DS.Model object
#B: Specifies name attribute as type string
#C: If model object has parent, it's of type Montric.MainMenu
#D: Model object can have zero, one, or more children of type Montric.MainMenu
#E: Model object can have one chart associated with it of type Montric.ChartModel
#F: Neither isSelected nor isExpanded are Ember Data attributes

Note that the model here doesn't specify an `id` property because the `id` property is implicit for `DS.Model` objects. Ember Data automatically adds an `id` property, and in fact, it raises an error if you attempt to specify one yourself. Ember Data uses this `id` property to keep track of all your loaded objects. You created two properties, `name` and `nodeType`, that are both of type `string`, which you specify via `DS.attr()`. Ember Data uses this information to automatically serialize data to and from your backend via the specified serializer. `DS.attr` supports the attributes `string`, `number`, or `date`, but as you'll see later in this chapter, you can specify your own attributes.

Getting your data in and out of your application isn't the only strength of Ember Data, as it also supports one-to-one, one-to-many, and many-to-many relationships between your data. In listing 5.1, both the `parent` and the `chart` properties specify a one-to-one relationship using `DS.belongsTo`, whereas the `children` property specifies a one-to-many relationship using `DS.hasMany`. Relationships will be explained in more detail later in this chapter.

You also specify two properties that aren't backed by Ember Data. These properties, `isSelected` and `isExpanded`, aren't strictly necessary to define in the class definition of `Montric.MainMenu`, but you can include them to make it clear that the rest of the application expects to find and use these properties. They are purely for human readability, as they have no Ember.js-specific meaning.

One of the major features of `DS.Model` objects is that they're also `Ember.Object` objects. You can therefore combine the model object with the core features of Ember.js itself, which include bindings, observers, and computed properties.

Often you want to know if a specific `Montric.MainMenu` has children or if it's a leaf node. The following listing shows how you can add computed properties to achieve this functionality across the application in one easy-to-find place.

Listing 5.2 Adding computed properties to the `MainMenu`

```
Montric.MainMenu = DS.Model.extend({
    name: DS.attr('string'),
    nodeType: DS.attr('string'),
    parent: DS.belongsTo('mainMenu'),
    children: DS.hasMany('mainMenu'),
    chart: DS.belongsTo('chart'),

    isSelected: false,
    isExpanded: false,
```

```
    hasChildren: function() {                                          #A
        return this.get('children').get('length') > 0;
    }.property('children').cacheable(),

    isLeaf: function() {                                               #B
        return this.get('children').get('length') == 0;
    }.property('children').cacheable()
});
```

#A: Returns true if number of children is greater than 0
#B: Returns true if number of children is 0

I'm sure you can see the huge advantage of enriching your model objects with computed properties in this manner. In fact, you can chain your computed properties together to create complex properties, and you can bind to these computed properties right out to the templates.

5.1.2 Ember Data is an identity map

A common problem with JavaScript-based web applications that fetch their data via a JSON- or a REST-based interface is that they tend to store that data right in the DOM tree itself. Although this may be a quick way to update the web application's views, it's also error prone because the developer needs to ensure that the old data isn't still displaying somewhere on the web page.

Ember Data solves this issue by implementing its data store as an identity map. Ember Data does the bookkeeping necessary to keep one and only one copy of your data in the cache. This copy is the master data that the rest of your application refers to. Whenever your application requests a specific model, by asking for it with the model's unique `id`, Ember Data makes sure that the object instances you receive are in fact the same each and every time you request a model of the same type and the same `id`. It doesn't matter if you get the data via a direct query by your model's `id` or if you iterate through a list of models. Each time you encounter a model object with the same `id`, you're working on the same instance of that object.

Figure 5.2 shows how Ember Data manages its data and how an identity map implementation works.

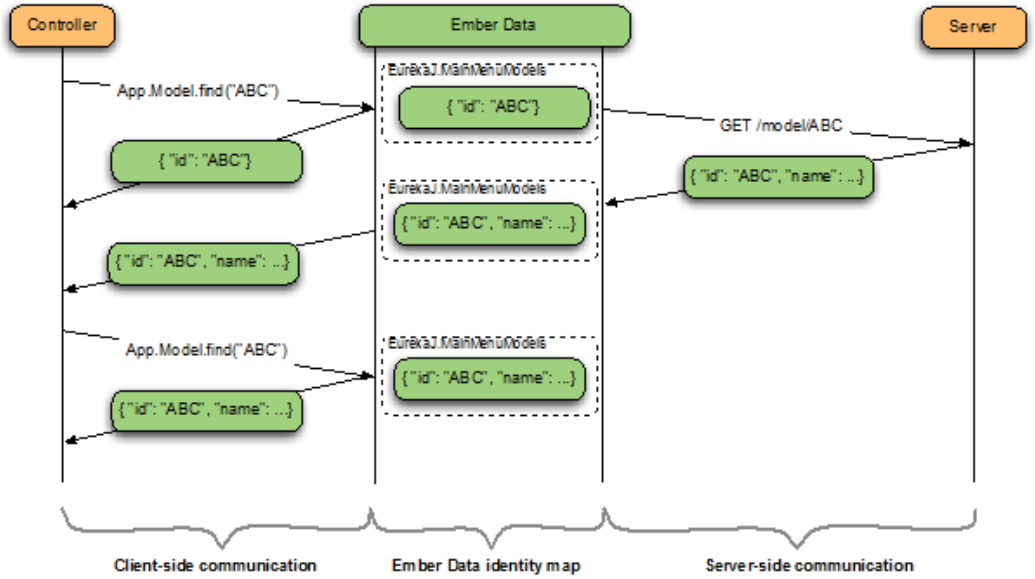


Figure 5.2 The data flow of the Ember Data identity map

In this example, you begin with an empty cache. You then request a model with the `id` "ABC" of type `Model`. Because Ember Data doesn't have a model object with the `id` "ABC", it creates one. The only information Ember Data knows about your model is the `id`, which means that it creates a new object of type `Model` and assigns its `id` property the value "ABC". Next, it synchronously returns that model object back to the controller. At the same time, though, it asynchronously reaches out to the server to fetch the rest of the model.

This asynchronous feature enables your application to set up the views that may be necessary for displaying your model object while you wait for the data to be returned from the server side, often making your application feel more snappy. When the asynchronous response comes back from the server, Ember Data ensures that it updates the object in the identity map before notifying the rest of the application that it's finished loading the model.

Because a binding is in place between the model in the identity map and a property in the controller, and bindings are also between the property on that model and the templates in the view, any changes occurring on the model object propagate through the controller and out to the template automatically.

Later, when you issue a new request to Ember Data of a model of type `Model` with `id` "ABC", Ember Data will already have the model cached in its identity map. An important concept of an identity map is that you'll receive the same object instance in return. This is important to keep your data updated and in sync throughout your application.

The fact that your data store is an identity map is something that Ember Data builds upon throughout its implementation. Ember Data is also smart enough to know when you need to

run a query against your backend and serve the results in an asynchronous manner, and when it can serve your request in a synchronous manner straight from the identity map.

Sometimes, though, you need to refresh your data after it's been loaded in the cache. Luckily, Ember Data has built-in support for this as well.

5.1.3 Relationships between model objects

Ember Data knows that your data can be messy, intertwined, connected, and nonstandard, and it does a good job at providing features and integration points that enable you to structure that data in a sensible manner. Ember Data comes preloaded with a REST adapter and a REST serializer that expect JSON data adhering to a specific contract. It also allows you to override any of these defaults either by telling the `RESTAdapter` how to interpret your JSON keys or by implementing your own custom adapter and serializer.

Relationships on Ember Data are implemented using the `id` of your model objects. In the previous example, the `Montric.MainMenu.children` property is a one–to-many relationship. Ember Data expects your backend to return a JSON array containing the `ids` of each of the children in this one-to-many relationship. Likewise, it also expects that each of the children refer back to the `parent` via a `belongsTo` relationship. This relationship is also made using the `id` property of the `Montric.MainMenu` object that it refers to. The following listing shows an example of how the JSON data is structured to comply with the `RESTAdapter`.

Listing 5.3 JSON data for the `Montric.MainMenu`

```
{
    "mainMenus": [                                              #A
        {
            "id": "JSFlotJAgent",                               #B
            "name": "JSFlotJAgent",
            "children": [                                       #C
                "JSFlotJAgent:Agent Statistics",
                "JSFlotJAgent:CPU",
                "JSFlotJAgent:Custom",
                "JSFlotJAgent:Frontend",
                "JSFlotJAgent:Memory",
                "JSFlotJAgent:Threads"
            ],
            "nodeType": "chart",
            "chart": "JSFlotJAgent",
            "parent": null                                      #D
        },
        {
            "id": "JSFlotJAgent:Agent Statistics",              #E
            "name": "Agent Statistics",
            "children": [
                "JSFlotJAgent:Agent Statistics:API Call Count"
            ],
            "nodeType": "chart",
            "chart": "JSFlotJAgent:Agent Statistics",
            "parent": "JSFlotJAgent"                            #F
        }
    ]
```

```
}
```

#A: Returns data in JSON array
#B: Each model object returned has unique id property for this data type
#C: List of children; each element refers to id property of child model
#D: Top-level menu items have no parent menu item attached
#E: Child menu item
#F: Child menu item refers back to parent via parent_id property

Unless you specify otherwise, the `RESTAdapter` expects you to send in a list of objects and it expects the name of this list to be derived from the model object that it's going to map this data to. The default `RESTAdapter` and `RESTSerializer` expect your keys to be in camelized form, meaning they start with a lowercase letter and subsequent words in the key start with an uppercase letter. Whenever you return a list of items, the key is suffixed with an "s", indicating that the value for that key is plural.

If you look closer at the JSON data, you'll notice that the array provided for the key `children` is a list of strings and not real objects. This list of strings represents the `ids` of each of the objects that the `children` property is associated with. In our case, `children` refers to a list of zero or more `Montric.MainMenu` objects.

This is also true for the `chart` relationship. Even though this relationship is one to one, the JSON returned from the server for the `MainMenu` model also represents the `id` for the object that the `chart` property is associated with. As you may have guessed, Ember Data uses these `ids` to wire your models together correctly.

Ember Data won't, however, materialize your associations ahead of time. What this means is that Ember Data won't try to connect and load in your associations before your code requires them. When you call `MainMenu.get('children')`, Ember Data looks up that `id` in its identity map and returns a result synchronously if it has a model of the correct type with that `id`. If the model object isn't loaded yet, it instead synchronously returns an empty record with only the `id`. It won't attempt to fetch the `chart` object from the server before you access a non-`id` property. You can rely on Ember Data to do the right thing most of the time. If you're not accessing any data that Ember Data hasn't stored in its cache, you can be certain that your application won't fetch data from your server before the user requests that data.

Although you can set up your views while you're awaiting a response from the server side, you may want to hold off rendering certain parts of your view until you're sure that the models that you're going to render have arrived safely in Ember Data, which leads us to take a closer look at the states an Ember Data model can be in.

5.1.4 Model states and events

Because most of the data you bring to your application via Ember Data is loaded in an asynchronous manner, each Ember Data models have a built-in state manager that keeps track of the state that your model objects are in at any given point in time. Ember Data uses this information internally to know how to provide your application with the data it receives from the server, but you can use this information when you build your application. For example, this information comes in handy when you want to implement loading indicators, for example, or

when you want to ensure that your GUI doesn't update until a certain amount of (or all) your data is loaded properly.

> **NOTE** Ember.js version 1.2.0 includes specific "loading" and "error" subroutes that let you handle scenarios where your data is loading and when you receive errors from the backend server.

To provide your application with this information, each model object that extends `DS.Model` has built-in convenience functions you can use both in your controllers and also in your templates. Each model object comes with the state properties shown in figure 5.3.

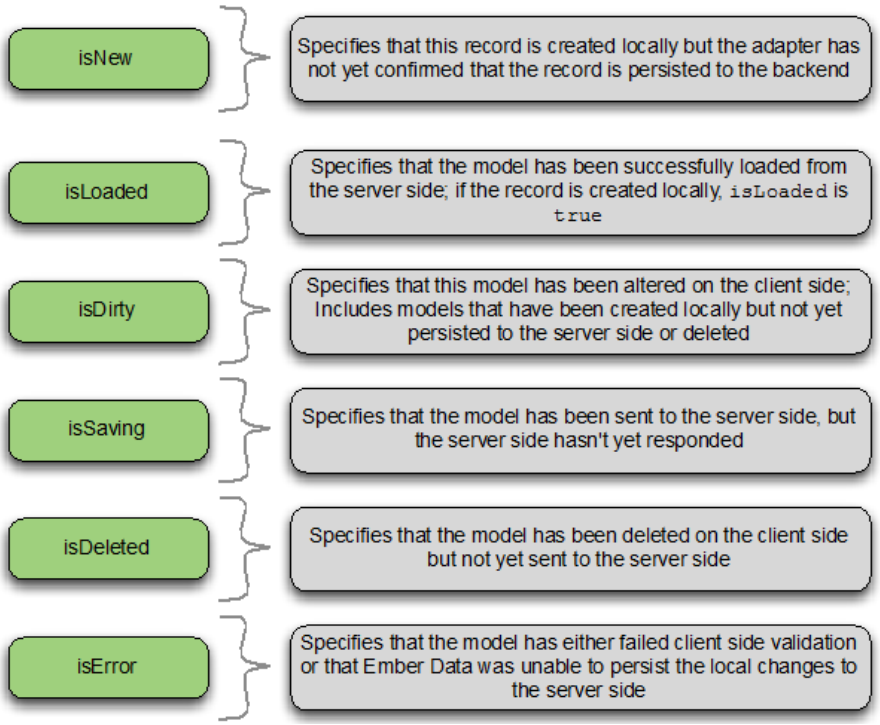


Figure 5.3 The states that an Ember Data model object can have

Note that these state properties aren't mutually exclusive. A model can have both `isDirty` and `isDeleted` return `true`, meaning that the model was deleted locally but not yet persisted, or both `isDirty` and `isSaving` return `true`, meaning that the model was updated locally and sent to the server side but that the server side has not yet responded with a status update.

Sometimes you need your controllers to be notified whenever your models change or when they enter a specific state. Each Ember Data model allows your controllers to subscribe to events. The valid events are shown in figure 5.4.

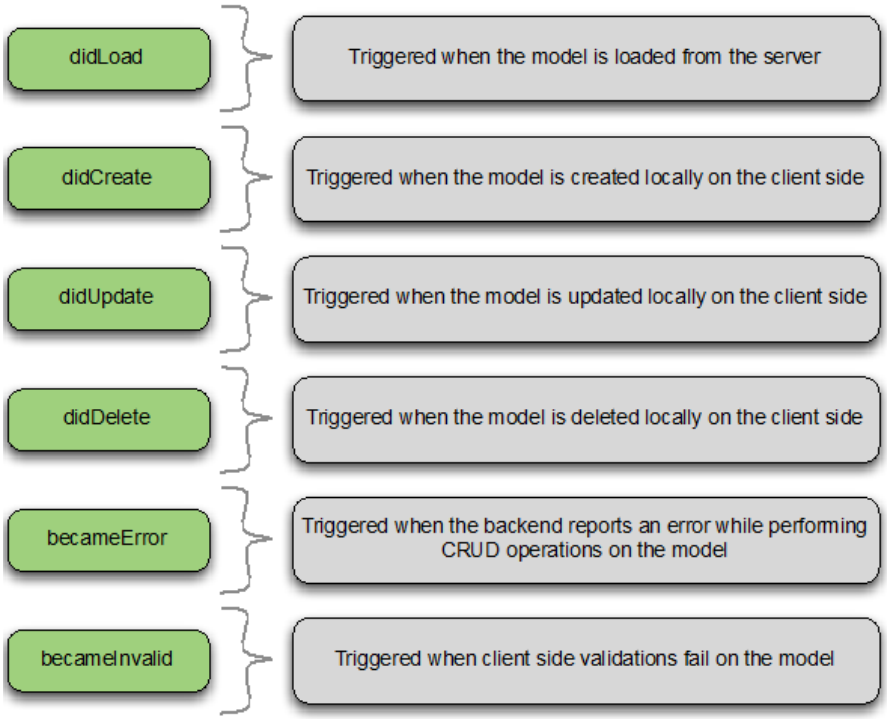


Figure 5.4 The model events

You are able to subscribe to all of these events within your code. If you want to perform an action when a model is loaded, you can use the on function on the model object in order to get notified when your model has finished loading.

```
model.on('didLoad', function() {                                    #A
  console.log("Loaded!");
});
```

#A: Uses on function to subscribe to didLoad event

As you've probably guessed by now, Ember Data models follow a lifecycle in which they can transition from one state to another. In fact, models have a hierarchical structure. Figure 5.5 shows the topmost states that a model can be in. This figure isn't complete, but it does contain the states that your data is most likely to be in.

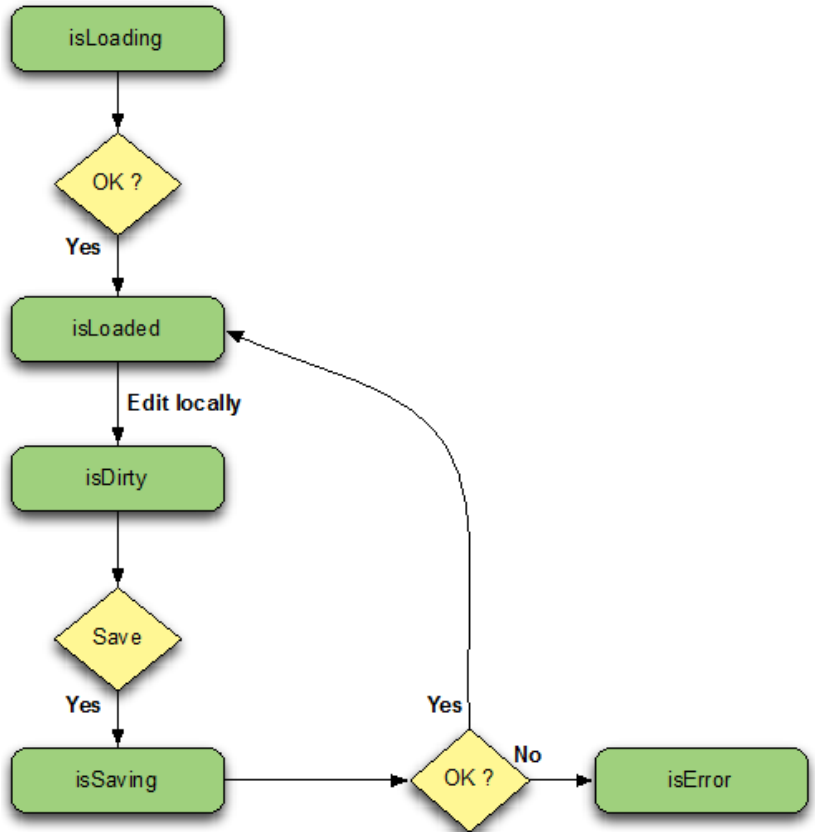


Figure 5.5 The most common Ember Data states that your models will be in

As you can see, a model usually starts out in the `isLoading` state. After the backend returns the model, it transitions to the `isLoaded` state. If the model gets modified locally, via either a user action or another client-side process (such as a timer), the model transitions to the `isDirty` state, where it remains until it's saved. When `save()` is called on a model, it transitions into the `isSaving` state. If the backend returns an OK response (HTTP 200, for example, and possibly an updated model), the model is brought back to the `isLoaded` state. If Ember Data fails while trying to persist the model to the server, or if the server returns a non-200 HTTP status code, the model transitions to the `isError` state.

5.1.5 Communicating with the backend

The default `RESTAdapter` uses XML HTTP Request (XHR) to integrate with the server, but you can provide your own adapter implementation. You may want to use a different type of integration, for example, or you may need to adapt Ember Data to work with an existing API.

Building in support for either LocalStorage or WebSockets is made possible and approachable in Ember Data. As discussed in chapters 1 and 2, you use a third-party LocalStorage adapter.

You're probably eager to learn how you can integrate Ember Data in your own application, so let's get started.

5.2 Firing up Ember Data

To use Ember Data, you need to be using a store. You can think of the store as an in-memory cache that Ember Data uses to retrieve and store its model objects. In fact, the store is also responsible for fetching data from your backend server. To get started, you need to define a store for your application, as shown in the following listing.

Listing 5.4 Creating a store

```
Montric.Store = DS.Store.extend({                                    #A
    adapter:  "Montric.Adapter"                                      #B
});
```

#A: Defines new store for application
#B: Defines which adapter to use when interfacing with server side

You create the store in the same manner you create any other Ember object, in this case, by extending a new `DS.Store` object. When Ember Data is initialized, it initializes a new store object and registers it with the Ember Container as `store:main`. You may have an API that's different from each data type that the server returns. Ember Data supports per-type adapters and serializers precisely for this purpose. We'll look at custom adapters and serializers later in this chapter.

You also need to specify which adapter to use. In this case, you use a custom adapter called `Montric.Adapter`, which is shown in the following listing.

Listing 5.5 The `Motric.Adapter`

```
Montric.Adapter = DS.RESTAdapter.extend({                            #A
    defaultSerializer: "Montric/application"                         #B
});
```

#A: Extends from default DS.RESTAdapter
#B: Uses default serializer Montric.ApplicationSerializer

You're creating a new `Montric.Adapter` that extends the standard `DS.RESTAdapter`. For now, use the standard functionality inherited from this adapter. The only piece you override is the default adapter, telling `Montric.Adapter` to rely on the adapter named `Montric.ApplicationSerializer`. The code for this serializer is shown in the following listing.

Listing 5.6 The `Montric.ApplicationSerializer`

```
Montric.ApplicationSerializer = DS.RESTSerializer.extend({});        #A
```

#A Extends default DS.RESTSerializer

You may wonder why you bothered creating your own implementation of the `RESTAdapter` and the `RESTSerializer`, as you aren't overriding any functionality in either of these two classes. The reason I'm showing you this now is twofold. First, I want to show you early how you can define a custom adapter and serializer for your application. Second, you'll make use of this later on in this chapter.

Now that you've initialized Ember Data, it's time to fetch some data from your server.

5.2.1 Fetching data from your models

You can load data from Ember Data (and, in turn, from your backend) in two ways. You can either call `store.find('model')` to load all your models of a specific type, or you can pass in an `id` to load a specific model object.

Consider the code shown in this listing.

Listing 5.7 Fetching data from your models

```
Montric.MainChartsRoute = Ember.Route.extend({
    model: function() {                                    #A
        return this.store.find('mainMenu');                #B
    }
});
```

#A: Uses model function to specify which model object to load for this route
#B: Returns all instances of Montric.MainMenu

As you learned in chapter 3, the model function specifies which model objects will be populated to a route's controller. If you use the Ember Router, this will, in fact, become the most common way that you'll load models from Ember Data to your controllers. As you can see, `this.store.find('mainMenu')` is how you tell Ember Data to fetch all objects of type `Montric.MainMenu`. Ember Data then looks at its internal cache and returns any objects it has there. If the cache for that model type is empty, it goes out to the backend server and asks it for the data. Ember Data does this by issuing an `HTTP GET XHR` to the URL `/mainMenus`, which it has derived automatically from the model's class name.

Likewise, if you had instead called `this.store.find('JSFlotJAgent')`, Ember Data would've looked at its cache to see if it had an object of type `Montric.MainMenu` with an `id` of `JSFlotJAgent`. When the server side returns, it then populates the cache with the updated data. Ember.js's observers and bindings take care of moving that data out all the way to the DOM. (See figure 5.2 for a schematic of this process.)

5.2.2 Specifying relationships between your models

You've loaded all the `Montric.MainMenu` objects from the server to the Ember Data identity map. But before we go over the relationships, let's quickly review what you'll be using the data for. Figure 5.6 shows the types of models that we have in the Montric application.

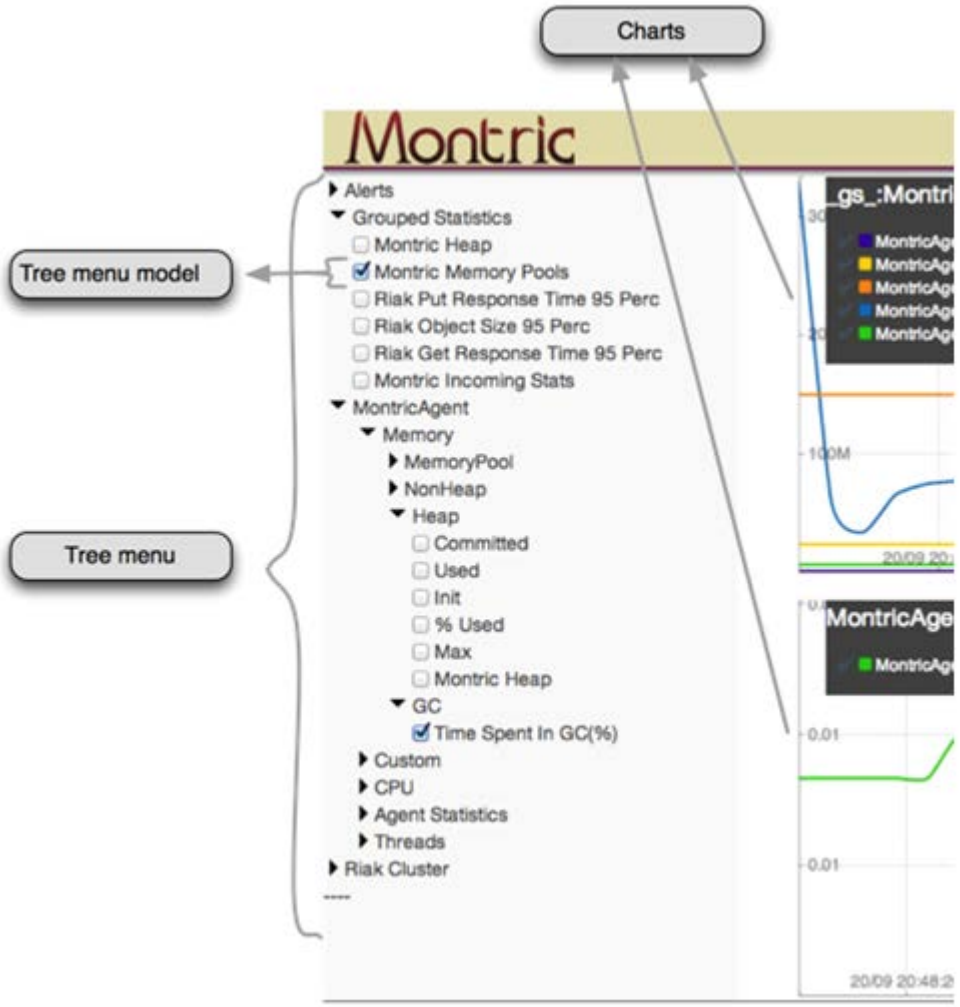


Figure 5.6 The use case for our the models that we have defined

Each of these `Montric.MainMenu` model objects represents a single element in the tree structure on the lefthand side. We'll call the top-level elements `rootNodes`. Each of the nodes in the tree can have zero or more `children` nodes. The `children` nodes are also of the `Montric.MainMenu` model type. If a node has `children`, a disclosure triangle is displayed at the left of the node's name. Users click this triangle to expand it and reveal its `children` nodes.

Users can expand the nodes until they reach a node that has zero children. This node is called a leaf node. Leaf nodes are selectable. By clicking the checkbox to the left of the leaf

nodes, users can select which nodes they want to show charts from. After at least one chart is selected, the area to the right of the tree menu displays each of the selected charts. Each `Montric.MainMenu` node has a `chart` property that Montric follows to load the chart for each selected node.

You've already seen the `MainMenu` model, so before moving on to the relationships present, let's go through the `Montric.Chart` model, which is shown in the following listing.

Listing 5.8 The `Montric.Chart` model

```
Montric.Chart = DS.Model.extend({                    #A
    name: DS.attr('string'),                         #B
    series: DS.attr('raw')                           #C
});
```

#A: Extends DS.Model
#B: Name property is of type string
#C: Series property is of custom type "raw"

The `Chart` model is fairly simple. It has two properties, a `name` and a `series`. The `name` property is a string, but the `series` property is of type `raw`. The `raw` property type isn't something that's supported directly in Ember Data but is rather a custom transformation that's specific to your Montric application. We'll get back to this custom transformation later in the chapter, but for now you can think of this property as holding a plain JavaScript array and not a primitive type of an `Ember.Object`.

Before explaining the different relationship types that Ember Data offers, let's quickly review the relationships that you've set up in the `Montric.MainMenu` and `Montric.Chart` models. Figure 5.7 shows the relationships.

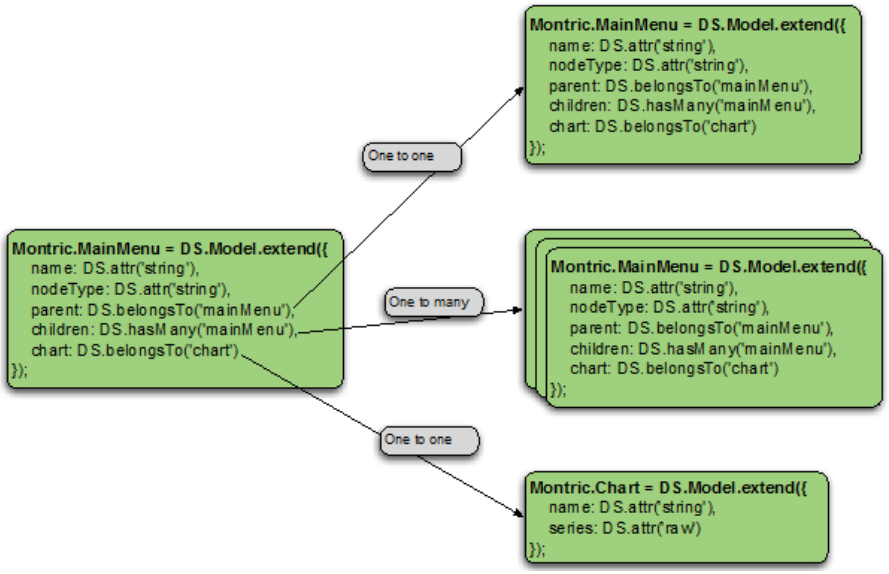


Figure 5.7 The relationships between the `Montric.MainMenu` and `Montric.Chart` models

As you can see, the `MainMenu` model is related to exactly one other `MainMenu` model via the `parent` property, whereas it's related to zero or more `MainMenu` models via the `children` property. This means that each item in the menu will have exactly one parent item, while it can have multiple child-items. In addition, the `MainMenu` model is related to exactly one `Chart` model via the `chart` property. With this in mind, let's explore the different relationship types that are built in to Ember Data.

5.3 Ember Data model associations

Ember Data supports a number of different types of associations, each with its own assumptions as to how it expects the data to be returned from the server by default. Ember Data lets you override these default assumptions and expectations, however it is useful to know which assiciations/relationships that are available as well as their default behaviour and server API expectations.

5.3.1 Understanding the Ember Data model relationships.

Out of the five associations types that are available between models in Ember Data, three of them can be considered true types, while the remainder two associations types can be considered as derivations, or special cases. The available Ember Data model relationships is show shown in figure 5.8.

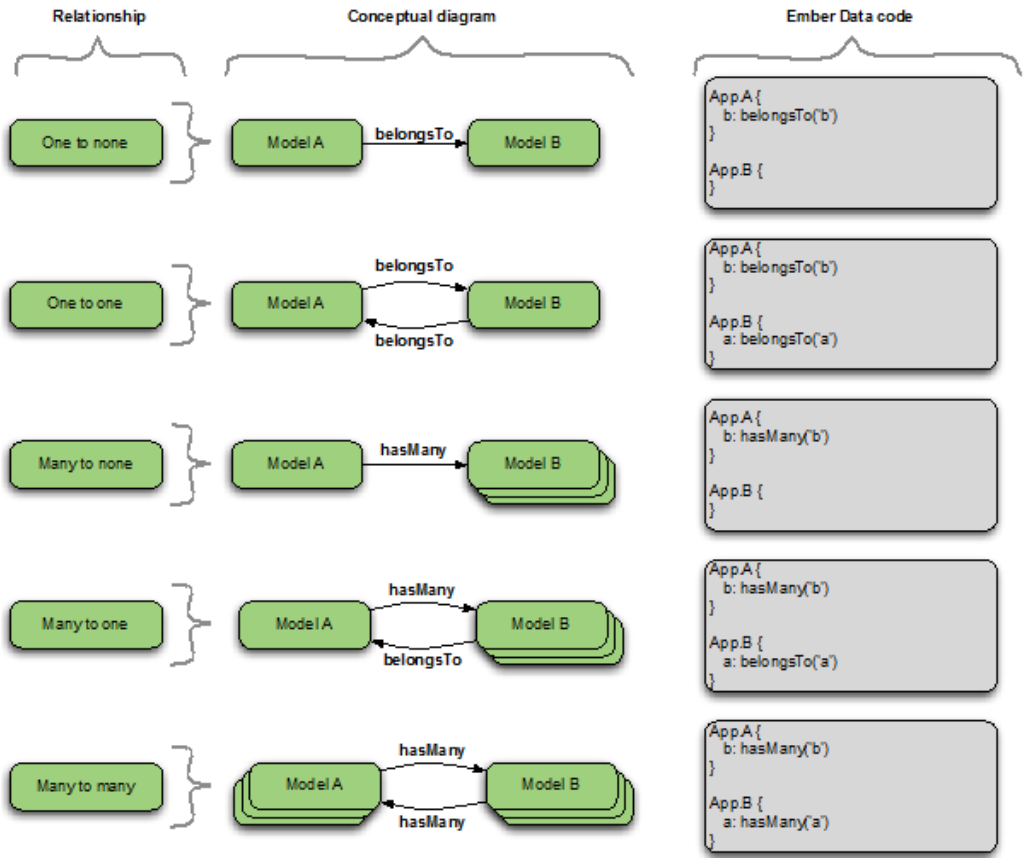


Figure 5.8 The Ember Data associations that allow us to specify relationships between our models

The names of the relationships are similar to the names of the relationships in relational database systems, except that the relational model doesn't distinguish between a `*-to-none` and a `*-to-many` relationship. Relationships are defined in Ember Data models either via the `belongsTo()` function or the `hasMany()` function. These functions tell Ember Data both how to wire your data together and also how it asks the server for the data as well as the format it expects the return to be in. Figure 5.9 shows how Ember Data and the default `RESTAdapter` expect the data to be returned from the server.

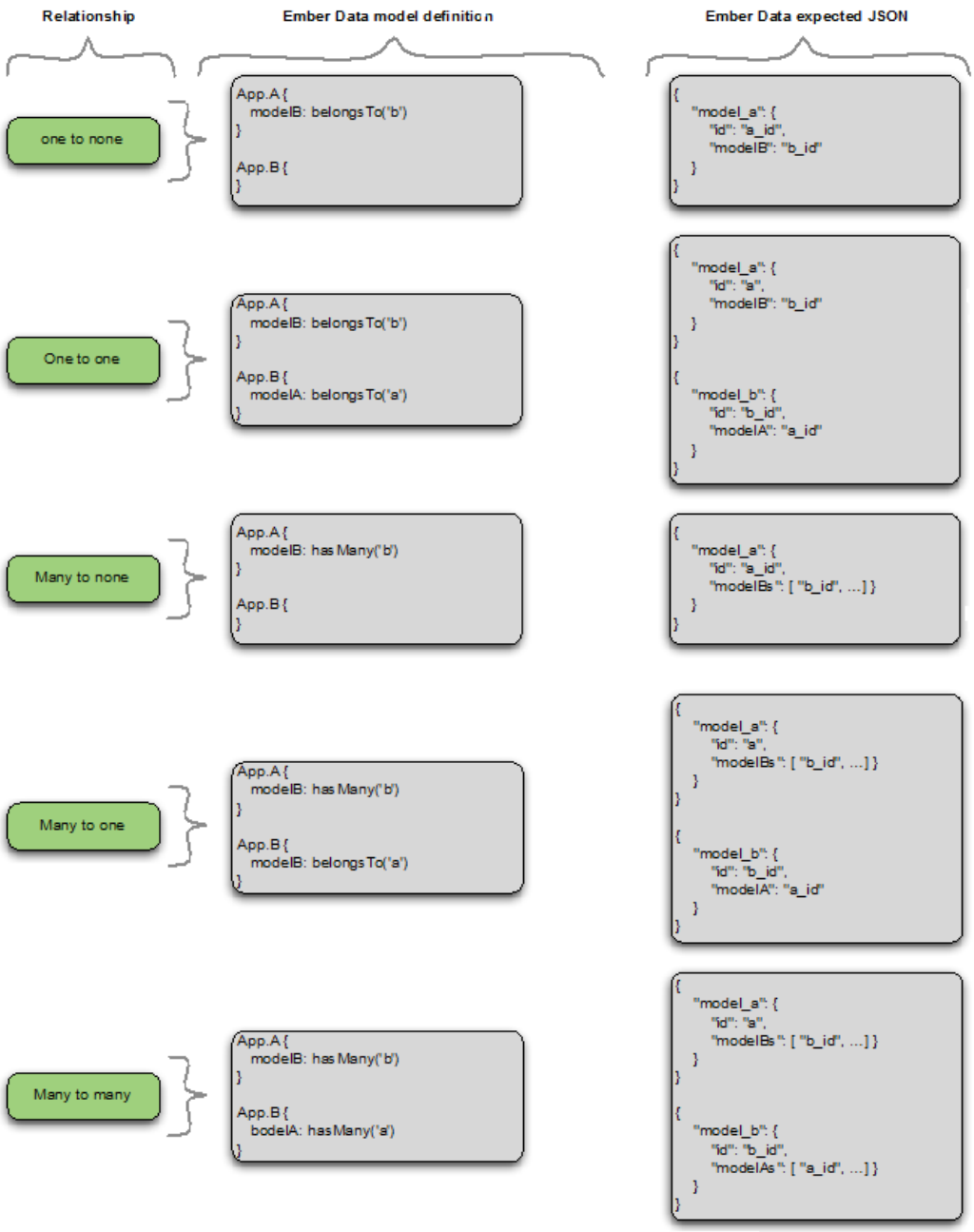


Figure 5.9 The Ember Data JSON mappings that are available by default

Keep in mind one important naming convention. Whenever you create a relationship via either `belongsTo()` or `hasMany()`, the string value you pass in must be the same as the name of the model class you're setting up the relationship to. This string value has the standard Ember.js camelized form that you've become used to. Other than that, the name of the properties directly reflects the expected keys in the JSON hash coming from the server.

I've mentioned that Ember Data employs a lazy loading structure for your data. For the Montric application, the data for a chart won't be loaded before the application requests access to either the `name` or the `series` property of the `Montric.Chart` model. This is fine for this use case, because we're only loading charts, one at a time, when the user selects a chart from tree menu.

But for other types of data, this can lead to a significant amount of AJAX calls between the client and the server. To tackle this issue, Ember Data supports embedding and sideloading data in the JSON hash returned from the server. Embedding works slightly different than standard relationships, so let's take a look at embedded records.

5.3.2 Ember Data sideloaded records

To optimize for a low number of requests between the client and the server, Ember Data supports the ability to both embed and sideload records in the response from the server. Sideloading works by adding multiple top-level hashes in the JSON returned from the server. For this to work, the `id` of each of the hashes needs to map with the camelized model names. Because Montric does not have any sideloaded records, consider the one-to-many relationship in listing 5.9, from what might be a common Blog application where each blog post has a set of comments associated with them.

Listing 5.9 One-to-many association

```
Blog.Post = DS.Model.extend({
    name: DS.attr('string'),
    comments: DS.hasMany('comment')                                    #A
});

Blog.Comment = DS.Model.extend({
    text: DS.attr('string'),
    post: DS.belongsTo('post')                                         #B
});
```

#A: Each Blog.Post has zero or more Blog.Comments
#B: Each Blog.Comment belongs to a single Blog.Post

The two model objects `Blog.Post` and `Blog.Comment` form a standard one-to-many relationship. If you followed the normal path, you'd most likely begin by fetching all or a couple of blog posts from the server. When the user chooses to view a blog post, you'd then fetch that blog post's comments to display them to the user. The exchange would go something like figure 5.10.

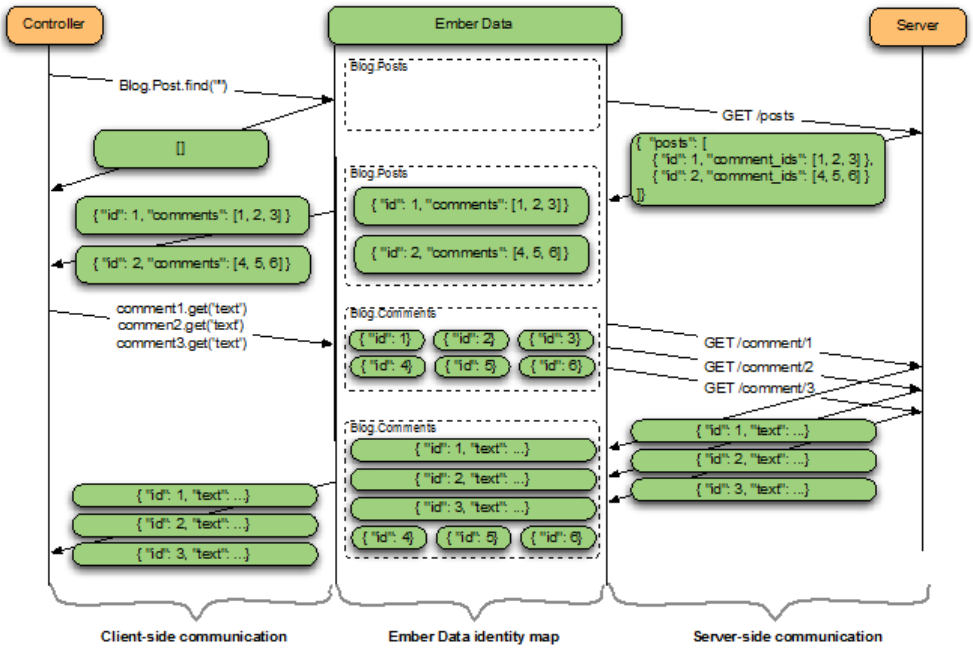


Figure 5.10 The data flow of loading posts and comments using the standard Ember Data control flow through associations

Multiple requests go from Ember Data to the server, one to fetch a list of `Blog.Post` models and then later one request to fetch each comment for the blog post with `id` 1. Depending on your application, data, and requirements, this can become quite inefficient if the data is sufficiently large. We'll look at the possibility of sideloading the comment information in the initial XHR `GET` to /post.

One possible solution is to sideload the comments to the same response as the post they belong to. The following listing shows the JSON for sideloading the comments.

Listing 5.10 The JSON for sideloading `Blog.Comment` records

```
{
  “posts: [                                                    #A
    {“id”: 1, “comments”: [1, 2, 3]},                          #B
    {“id”: 2, “comments”: [4, 5, 6]}                           #C
  ],
  “comments”; [                                                #D
    {“id”: 1, “text”: “Comment 1”, “post”: 1},                 #E
    {“id”: 2, “text”: “Comment 2”, “post”: 1},                 #E
    {“id”: 3, “text”: “Comment 3”, “post”: 1},                 #E
    {“id”: 4, “text”: “Comment 4”, “post”: 2},                 #F
    {“id”: 5, “text”: “Comment 5”, “post”: 2},                 #F
    {“id”: 6, “text”: “Comment 6”, “post”: 2},                 #F
```

```
    ]
}
```

#A: Post's hash is included as before
#B: Post with id 1 is related to comments with ids 1, 2, and 3
#C: Post with id 2 is related to comments with ids 4, 5, and 6
#D: Comments are sideloaded in same response
#E: Comments with ids 1, 2 and 3 are related to post with id 1
#F: Comments with ids 4, 5, and 6 are related to post with id 2

Here, when you load the `Blog.Post` models, instead of having the server return only the data for the `Post` model objects, you also append an array with the key `comments` that contains the comments associated with your two blog posts. By including a JSON hash with the correct `ids`, Ember Data loads the two posts and the six comments into its identity map in one big swoop. Additionally, it's not necessary to tell Ember Data that it has to accept sideloaded objects.

Figure 5.11 shows the updated data flow when sideloading data.

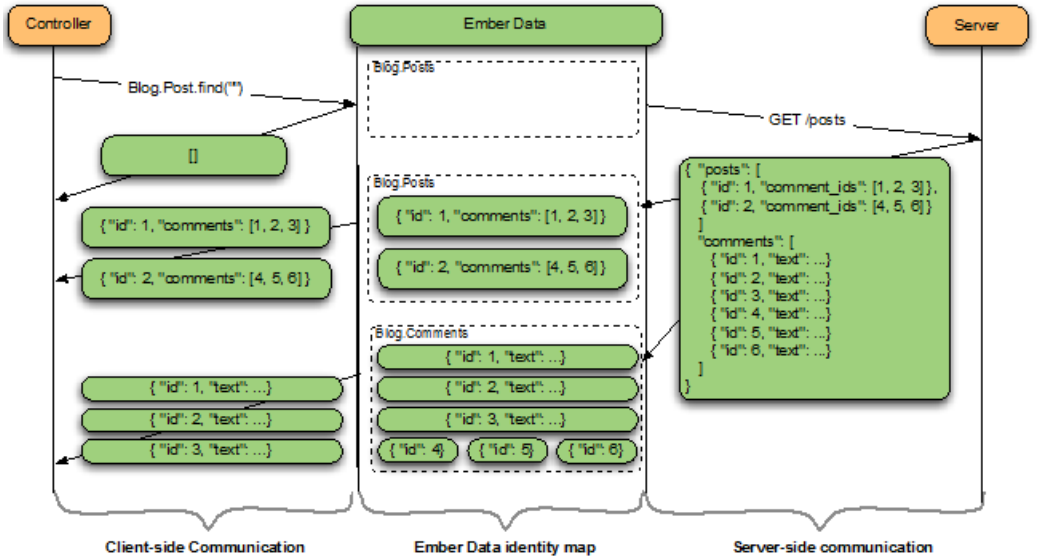


Figure 5.11 Reducing the number of XHR request by sideloading comments while loading the posts

The advantage to the sideloading approach should be clear; you've effectively reduced the number of XHRs from a total of seven down to one, while you've also reduced the total number of bytes sent from the server, even if you were to disregard the time required to negotiate for the seven XHR connections as well as the extra bytes for the six HTTP headers that you've removed.

The downside is that the server needs to send data to the client that potentially will never be displayed to the user (if the user never visits the blog post with `id` 2, for instance). You should therefore consider the implications of sideloading before you implement this in your applications.

I've mentioned that it's possible to override many of the default assumptions that the RESTA`dapter` has toward the JSON that it receives from the server. Before we conclude this chapter, let's take a look at the customizations that the adapter and the serializer have.

5.4 Customizing the adapter and serializer

Because Ember Data supports both default and per type adapters and serializers, you can support any of the following scenarios:

- Write a separate adapter and serializer to support applications for which the server-side API has no common standard across data types
- Write a separate adapter and serializer to support data types that differ from the server-side API that your server specifies
- Write a separate adapter but keep the default serializer in cases where the URL patterns or top-level JSON keys for a specific type differ from the server-side API that your server specifies

Let's get started with a specific example from Montric that illustrates the third scenario: creating a custom adapter for the `Chart` model to customize the URL that Ember Data calls the server with.

5.4.1 Writing a custom adapter but keeping the default serializer

In Montric, when the user selects a chart to view from the main menu, Ember Data looks up which chart to load via the `chart` property of the `Montric.MainMenu` node that the user selected. It then notices that this is a one-to-one relationship. Because Montric won't have the chart loaded in its cache initially, it reaches out to the server via the adapter's `find()` method.

But because the user selects is able to select the time period that the chart will be based on from elsewhere in the application, you need to also tell the server side the time period that you wish to view the chart for. If the user has not made a selection, the timespan will be set to 10 minutes. The user interface where the user selects the chart time period is shown in figure 5.12.

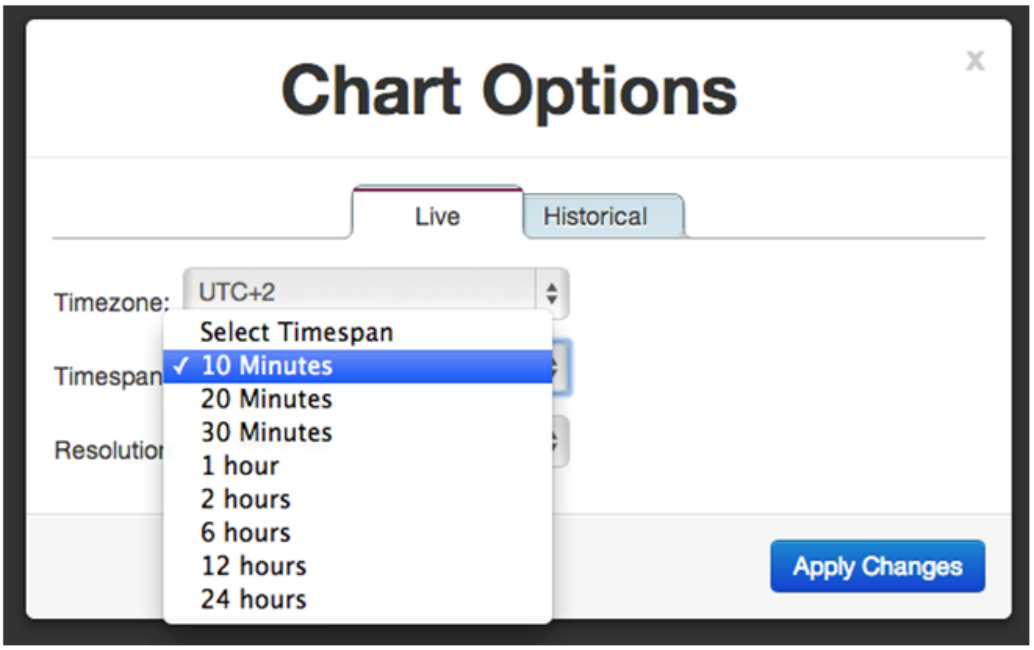


Figure 5.12 Selecting the timespan that the charts displayed will be based on

Next, you need to add a new custom adapter for the `Chart` model. You can do this by implementing a new class called `Montric.ChartAdapter`:

```
Montric.ChartAdapter = DS.RESTAdapter.extend({
    //contents omitted at this point
});
```

Here, you create a new class that extends the default `RESTAdapter`. The name of the adapter tells Ember.js to use this adapter whenever it needs to either get or persist `Chart` models. This follows the naming convention that you've become accustomed to while using Ember.js, and it's nicely implemented for custom adapters, too.

When you create a custom adapter, you can override a few things from the default `RESTAdapter` to customize its behavior. Figure 5.13 shows the methods you can override and their responsibilities.

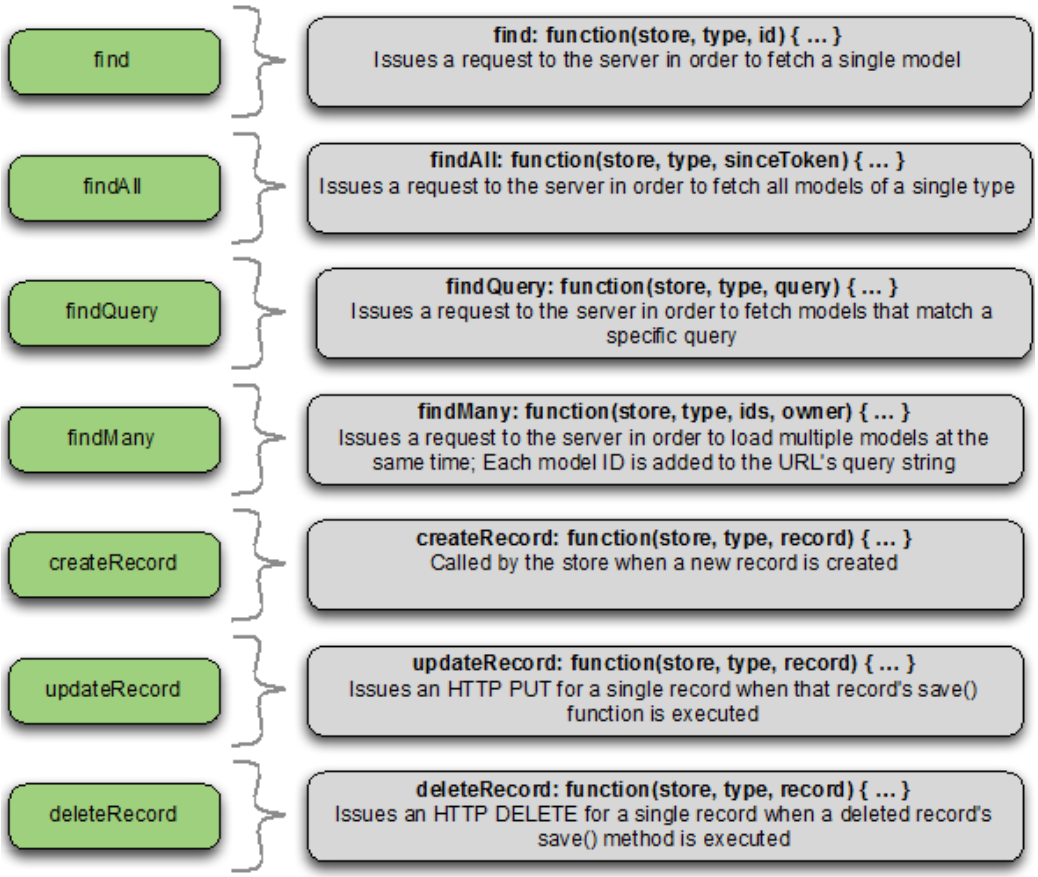


Figure 5.13 The methods available in the Ember Data adapter that you can override in order to create a custom adapter

For your purpose, you only need to override a single function, the `find()` function, to append a query string to the URL whenever you fetch single `Montric.Chart` models.

Now that you've identified which function to override, let's update the `ChartAdapter` as shown in the following listing.

Listing 5.11 The updated `Montric.ChartAdapter`

```
Montric.ChartAdapter = DS.RESTAdapter.extend({
    find: function(store, type, id) {
        return this.ajax(this.buildURL(type.typeKey, id), 'GET');        #A
    },

    buildURL: function(type, id) {                                       #B
        var host = Ember.get(this, 'host'),
```

```
            namespace = Ember.get(this, 'namespace'),
            url = [];

        if (host) { url.push(host); }
        if (namespace) { url.push(namespace); }

        url.push(Ember.String.pluralize(type));
        if (id) { url.push(id); }

        url = url.join('/');
        if (!host) { url = '/' + url; }

        var queryString = this.buildQueryString();                          #C

        return url + queryString;                                           #D
    },

    buildQueryString: function() {                                          #E
        var queryString = "?tz=" + Montric.get('selectedTimezone');
        if (Montric.get('showLiveCharts')) {
            queryString += "&ts=" + Montric.get('selectedChartTimespan');
        } else {
            queryString += "&chartFrom=" +
Montric.get('selectedChartFromMs');
            queryString += "&chartTo=" + Montric.get('selectedChartToMs');
        }
        queryString += "&rs=" + Montric.get('selectedChartResolution');

        return queryString;
    }
});
```

#A: Copy of default code from DS.RESTAdapter
#B: Overrides DS.RESTAdapter's buildURL function to append query string
#C: Builds queryString
#D: Returns URL and queryString as URL to use against server
#E: Custom function builds up query string

You don't have to be able to follow all the code in this listing. Most of the code is taken straight from the `DS.RESTAdapter` code. The only thing you add is a function that builds up the query string and appends it to the URL. Previously, the URLs to retrieve `Chart` models from the server looked like this:

```
/charts/_gs_:Montric%20Heap
```

Now, the URLs look like this:

```
/charts/_gs_:Montric%20Heap?tz=2&ts=10&rs=15
```

Now that you've seen how to implement a custom adapter to query the server with non-standard URLs, let's have a look at how to add a serializer to parse JSON that doesn't follow the `RESTAdapter`'s conventions.

5.4.2 Writing custom adapters and serializers

When a user logs in to Montric, the application issues a `find()` to the currently logged in user. An example of a nonstandard (in regards to the `RESTSerializer`) JSON hash is shown in this listing.

Listing 5.12 A nonstandard JSON hash for the `Montric.User` model

```
{
    "user_model": {                                                    #A
        "id": joachim@haagen-software.no,
        "user_name": "joachim@haagen-software.no",                     #A
        "account_name": "Haagen Software",                             #A
        "user_role": "root",                                           #A
        "firstname": "Joachim Haagen",
        "lastname": "Skeie",
        "company": "Haagen Software AS",
        "country": "Norway"
    }
}
```

#A: Nonstandard JSON keys

As you can see, the three keys `user_name`, `account_name`, and `user_role` are all nonstandard with regard to the `RESTSerializer`. In addition, the key for the user object, `user_model`, doesn't follow the `RESTSerializer` standard. You can sort this out by creating a new class, `Montric.UserSerializer`. When you write a custom serializer, you can override a few functions to customize how the serializer works with the JSON data (figure 5.14).

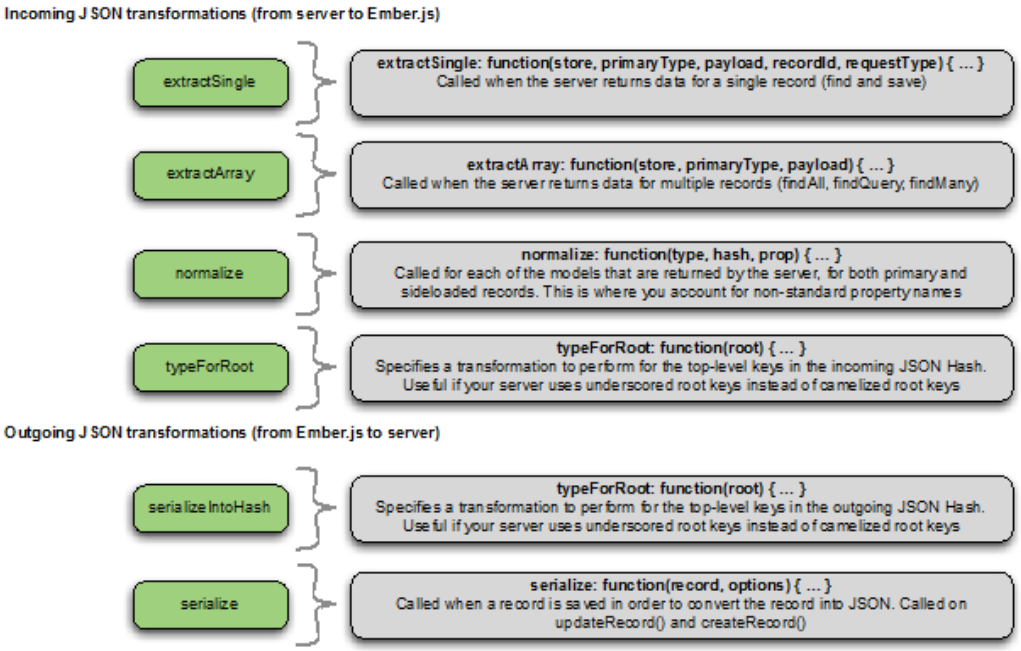


Figure 5.14 The methods you can override in order to build a custom serializer

To work with the JSON returned for each of the top-level keys, you override two functions: one that lets you specify the format of the top-level keys and one that lets you account for nonstandard property names. For this example, you need to override `typeForRoot` to tell the adapter to use `user_model` as the top-level key. You also need to implement `normalize` to support the three property names `user_name`, `account_name`, and `user_role`. The following listing shows the `Montric.UserSerializer`.

Listing 5.13 The `Montric.UserSerializer`

```
Montric.UserSerializer = DS.RESTSerializer.extend({
    typeForRoot: function(root) {
        return root.slice(root.length-6, root.length);                  #A
    },

    normalize: function(type, hash, property) {
        var json = {};                                                  #B
        for (var prop : hash) {                                         #C
            json[prop.camelize()] = hash[prop];                         #D
        }

        return this._super(type, json, property);                       #E
    }
});
```

#A: Strips away last six characters of top-level key
#B: Creates new object that you'll build up with correct property keys
#C: Iterates over each property in original hash
#D: Adds new camelize property with value from original hash
#E: Making sure that we call the normalize function in the super class

This code is extremely specific to this use case. You strip away the last six characters of each of the top-level keys, which works for this one use case, but if you find yourself needing to implement the `typeForRoot` function, you might be better off implementing something more sophisticated.

The `normalize` function, however, is more robust. It creates a new object (`json`) before it iterates over each of the properties for the current hash. For each property in the hash, it adds a new property to the `json` object, of which each key is camelized. The function ends by calling the `normalize` function of the super class `DS.RESTSerializer`. This is important for the rest of the serialization to work.

Now that you've seen how you can create both custom adapters and serializers, let's move on to see how to customize the URLs that your application uses when it contacts the server.

5.4.3 Custom URLs

By default, Ember Data expects to reach its data with a URL that lives at the root of the domain where the application runs. All the URLs are prefixed with a /, followed by the decamelized and underscored model name.

Some backends have special requirements that make this naming convention either inconvenient or impossible for different reasons. In these cases, you have two options: you can either specify a namespace to prepend a specific path to where the backend responds or you can specify a new URL altogether. Both approaches are shown here:

```
Montric.Adapter = DS.RESTAdapter.extend({
    defaultSerializer: "Montric/application",
    namespace: 'json/v1',                                    #A
    host: 'http://api.myapp.com'                             #B
});
```

#A: Prepends /json to URL by updating the namespace property
#B: Prepends URL by updating url property

Normally, when you call `this.store.find('mainMenu')` you issue an XHR GET to the URL /mainMenus. In this example, you add both a namespace and a host to our application default adapter. This causes that call to issue an XHR GET to the URL http://api.myapp.com/json/v1/mainMenus instead.

5.5 Summary

This chapter serves as an introduction to Ember Data and the built-in `RESTAdapter`. You started out learning how you can implement models that extend the Ember Data model object to represent the data that your application uses. You then looked at how Ember Data is structured as an identity map to ensure that your application is consistent by ensuring that only one copy of the data lives in Ember Data.

Models in Ember Data follow a strict lifecycle, and you looked at how that affects how you use the data in your application and how you can use this fact when you write your own application.

Ember Data provides powerful built-in relationships between model types. You looked at how you can use these relationships to build complex structures between your data. In addition, you saw how the lazy loading of relationships can have a negative effect on performance and how you can use sideloading to amend these issues by reducing the number of XHRs that are required to fetch data from the server.

Finally, this chapter wraps up by explaining the customization that the `RESTAdapter` supports and how you can build your own adapters and serializers.

Thus far, we've covered most of the features that are offered in Ember Data beta 2. Sometimes though, Ember Data is more than you need for your applications. In the next chapter, you'll look at how you can use Ember.js without the help of Ember Data.

6

Interfacing with the server side without using Ember Data

This chapter covers:

- Learning what Ember.js expects from your data layer
- Define a generic Model object that acts as the model layer for the application
- Fetching, persisting and deleting data using jQuery Ajax calls
- Learning where and how to integrate your model layer with Ember Router through the web application used for the Ember Fest conference

Ember Data will become a remarkable product; however, it's not ready for production use at the time I'm writing this. Even though you can interact with the server side in the same way you're used to with jQuery, Ember.js does require some extra thought before implementing a strategy for retrieving and persisting data between your Ember.js application and the server side.

In some situations Ember Data may not be suitable for your particular use case. For example, if you're dealing with a simple data structure, you may prefer to implement something less complex than Ember Data to fetch the data from the back end. Ember Router makes it easy to write your own integration layer. That being said, though, once your application grows you will most likely have to find a solution to a lot of issues that Ember Data solves out of the box for you. Regardless, learning how to utilize Ember Router in order to communicate efficiently with any backend you might have, or is building, comes in very handy in situations where you have a very non-standard API to interact with, or when you just want to implement something quick for some (or all) of you model objects.

This chapter examines the assumptions that Ember.js has about your client-to-server-side strategy, as well as an implementation strategy used in a real-world application.

Figure 6.1 shows the parts of Ember.js that we will be looking at in this chapter.

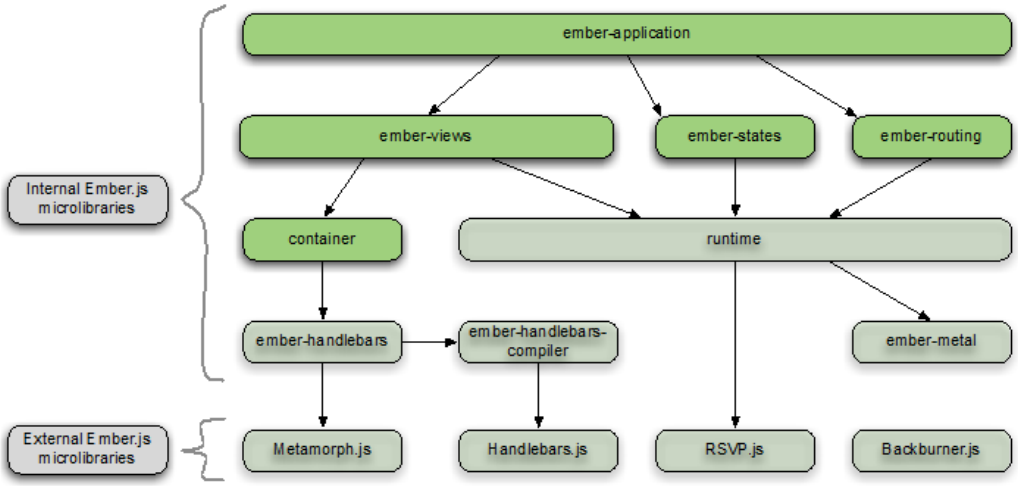


Figure 6.1 Parts of Ember.js you'll be working on in this chapter

Let's start by taking a quick look at the application you're developing parts of before diving into where you'll plug your data layer into your application.

6.1 Introducing Ember Fest

In this chapter you'll develop the data model layer of the web application for the European Ember.js Conference, Ember Fest. This application is fairly straightforward with limited functionality. Figure 6.2 shows the GUI of three of the routes within the Ember Fest application.

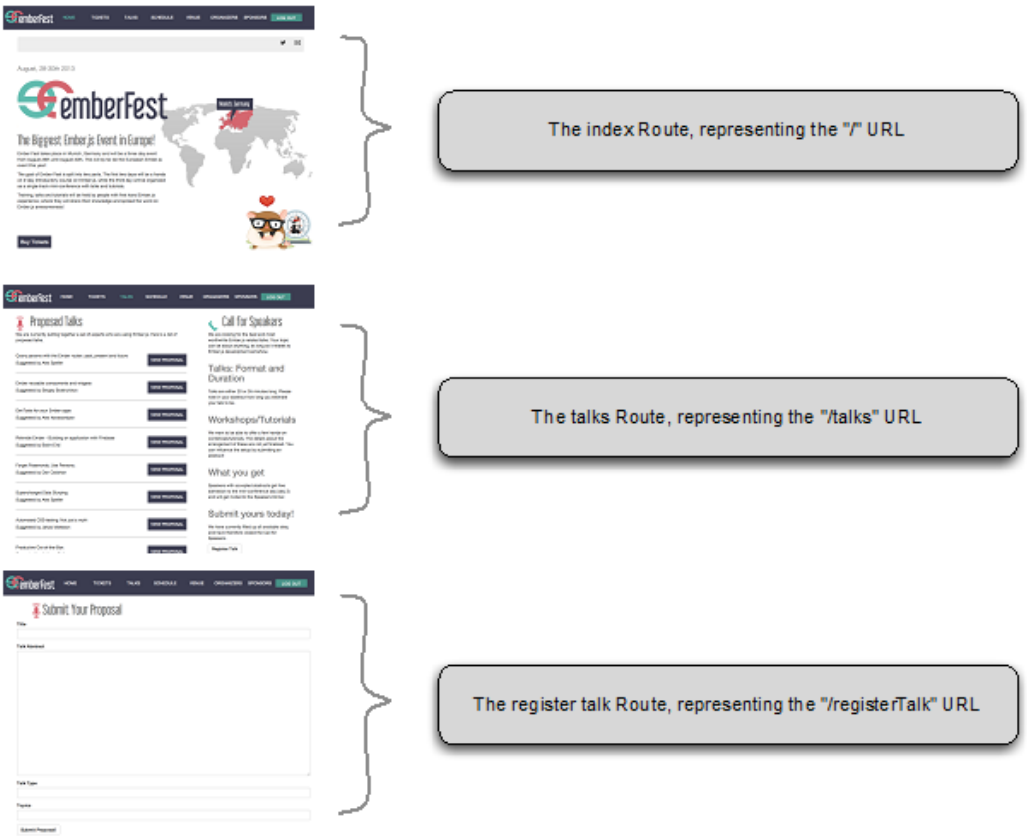


Figure 6.2 Three routes in the Ember Fest application

As you can see, the application is a set of pages, with each page representing a single route within the application. The application highlights the current route in the navigation bar as the user navigates through the application. At the top the user can log in or create a new account, which is provided via Mozilla Persona in this application. Authentication is covered in more detail in chapter 9.

We'll look at how you can use Ember.js Router as an integration point into the model layer, and then we'll look at the way you use each route's `model()` hook to fetch data into the Ember Fest controllers.

6.1.1 Understanding the application router

The router is the glue that holds your Ember.js application together. Having the important role that it has, it's not surprising that the router plays a significant role in your data layer strategy as well.

The router for this application is shown in the following listing.

Listing 6.1 Ember Fest router

```
Emberfest.Router = Ember.Router.extend({
    location: 'history'
});

Emberfest.Router.map(function() {

    this.route('tickets');
    this.resource('talks', function() {                                    #A
        this.route('talk', {path: "/talk/:talk_id"});                      #B
    });
    this.route('schedule');
    this.route('venue');
    this.route('organizers');
    this.route('sponsors');
    this.route("registerTalk");
});
```

#A: Talks route displays all submitted talks
#B: Talks.talk subroute displays current selected talk

As you can see from listing 6.1, each of the application's routes is defined in a flat structure with little hierarchy. Because each route effectively replaces the contents of the application (not including the header and footer), this is an appropriate and simple router definition. Note, though, that the `talks.talk` route is defined as a subroute of the `talks` route.

Throughout this chapter you'll concentrate on three routes. You'll take a closer look at both the `talks` route and the `talks.talk` route while also taking a closer look at the `registerTalk` route, which is responsible for allowing logged-in users to register new talks for the conference.

You'll use the `model()`hook for the `talks` and the `talks.talk` routes to tell the application's data layer to fetch data from the server.

Let's take a look at these routes before moving on to the data model implementation.

6.1.2 Using the model() hook to fetch data

You'll fetch all talks in the Ember Fest application and store them in the `TalksController`. You can accomplish this task in many ways; however, some of these approaches will lead you down the path of duplicated data, missing data, or missed updates.

Ember.js was built to support complex data with complex associations, and it can help you toward an efficient data layer implementation. The key to Ember Routers way of handing loading data from the backend into the Ember.js application is the `model()` function in each of your routes' definition.

The following listing shows how to use the `model()` function to fetch the application's talks and load them into the `Emberfest.TalksController`.

Listing 6.2 TalksRoute

```
Emberfest.TalksRoute = Ember.Route.extend({
```

```
    model: function() {                                          #A
        return EmberFest.Talk.findAll();                         #B
    }
});
```

#A: Hooks in data layer
#B: Returns result of fetching all talks from server

The code above looks pretty simple, right? As it should be! You return the result of fetching all the talks from the server from the `model()` function. Ember Router injects this data into the `content` property of the correct controller, the `TalksController` in this case.

Ember.js calls the `model()` function when the `talks` route is created. This prevents additional calls to the `findAll()` function and reduces the traffic between the client and the server.

To ensure that you don't end up with duplicate data for your application, you'll implement an identity map in the model layer. But first, let's take a look at the `TalksTalkRoute`, shown in the following listing.

Listing 6.3 `TalksTalkRoute`

```
Emberfest.TalksTalkRoute = Ember.Route.extend({
    model: function(id) {
        return Emberfest.Talk.find(id.talk_id);                  #A
    }
});
```

#A: Returns a single talk

The `TalksTalkRoute` is similar to the `TalksRoute`, but notice that you pass the `id` parameter to the `model()` function to fetch a single `Emberfest.Talk` object from the data layer.

You may remember from the discussion of the router in chapter 3 that when you enter the `TalksTalkRoute` via a direct URL, Ember.js sets up the `TalksRoute` before setting up the `TalksTalkRoute`. Both `Emberfest.Talk.findAll()` and `Emberfest.Talk.find(id)` are called one right after the other. To account for both methods being called, you'll design your data layer intelligently enough not to query the server twice because the communication between your client application and the server application is most likely be the slowest part of your system, by at least one order of magnitude.

6.1.3 Implementing an identity map

Several approaches are available at this point to avoid multiple server queries. For this application I chose an identity-map-like implementation. (For a discussion of identity maps, see chapter 5). You'll implement an in-browser cache ensuring that there is only one object of each type and id. After data is loaded into the identity map only one object of each of the talks will remain.

This, in turn, ensures that whenever you call `Emberfest.Talk.find(id)`, you retrieve the same instance of that object, and the cache contains no duplicate data. Figure 6.3 recaps the role of the identity map in this setting.

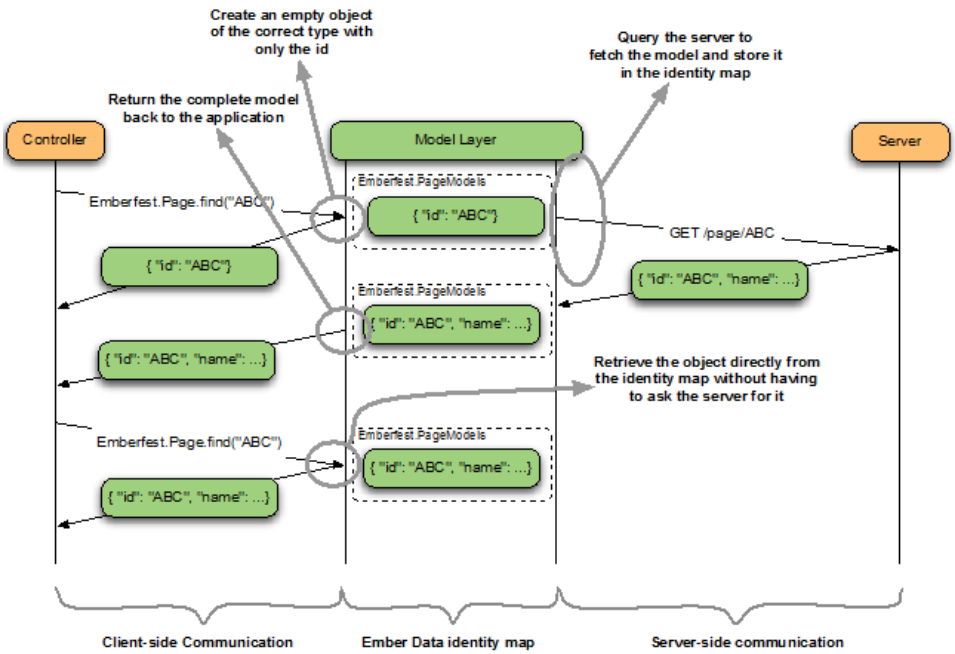


Figure 6.3 The implemented identity map

Now that you've seen the structure of Ember Fest and you have a better understanding of how an identity map works, let's take a closer look at the implementation.

6.2 Fetching Data

For the Ember Fest application, it's sufficient to always fetch all items from the server because you're storing a small amount of data for each individual data type. For instance, only a handful of submitted talk proposals are entered into the Ember Fest application. You only have to implement Ajax calls from within the `findAll()` function.

You first implement the `find()` function before you implement the data fetching in the `findAll()` function. After you see how you can fetch data both from the local identity map as well as from the server you'll look at an abstract implementation of a shared model class, which keeps code duplication to a minimum.

Let's take a closer look at the `find()` function.

6.2.1 Returning a specific task via the find() function

Because some of your controllers require only a single item, you support fetching single items from your identity map. When the user enters the `talks.talk` route, Ember.js calls the `find()` function for the `talks.talk` route to locate the specific talk that the user selected from the identity map. But because the `talks.talk` route is a subroute of the `talks` route,

Ember.js also calls the `findAll()` function of the `talks` route to load all talks. The object that's returned from the `find()` function must be the same as the one that will be included in the list of objects from the `findAll()` function. This is an important concept that's central to the implementation of an identity map.

The following listing shows the implementation of the `find()` function.

Listing 6.4 The `find()` function

```
Emberfest.Model = Ember.Object.extend();                                    #A

Emberfest.Model.reopenClass({                                               #B
    find: function(id, type) {                                              #C
        var foundItem = this.contentArrayContains(id, type);                #D

        if (!foundItem) {
            foundItem = type.create({ id: id, isLoaded: false});            #E
            Ember.get(type, 'collection').pushObject(foundItem);            #F
        }

        return foundItem;                                                   #G
    }
});
```

#A: Defines new class
#B: Reopens class definition to add class methods
#C: Implements find function, taking an id and a type parameter
#D: Checks whether object of this type and id is already loaded
#E: If an object of this type and id isn't yet loaded, creates a new object and sets id and isLoaded
#F: Pushes new object into identity map of this type
#G: Returns found item or item with only an id

The code defines a new class type called `Emberfest.Model`. This is the top-level model type for your application, and it's not intended to be instantiated into an object directly. Next, you reopen the class definition to add the `find()` class method. You add this method inside `reopenClass` to avoid having to instantiate the class into an object. This is similar to static methods from other languages such as Java and .Net.

As you can see, the `find()` function takes two parameters, an `id` and a `type`. The function calls the `contentArrayContains()` function to check whether an object of that type and id already exists in the identity map. If so, you return that item directly; if not, you create a new object of the correct type. In this case you create a new instance of the type you passed into the function. The only thing you know about the object at this point is the id because you passed this into the `find()` function. In addition to setting the id of the newly created object, you also set its `isLoaded` property to `false` so that the rest of the application can see whether the object is fully loaded or created locally. Finally, you add the newly created object to the collection property using `pushObject`.

Next, let's implement the `findAll()` function to fetch the data from the server.

6.2.2 Returning all talks via the findAll() function

The `findAll()` function calls the server side to fetch data and load it into your cache. To fetch data, you pass the function three parameters:

- a URL
- the data type that you expect to get in return
- the hash key that identifies the data retrieved from the server side

As you'll see later, you'll tell the `Emberfest.Talk.findAll()` function to fetch data through the /abstracts URL and get a collection of `Emberfest.Talk` objects in return.

The following listing shows the implementation of the `findAll()` function.

Listing 6.5 The `findAll()` function

```
findAll: function(url, type, key) {                                    #A

    var collection = this;                                             #B
    $.getJSON(url, function(data) {                                    #C
        $.each(data[key], function(i, row) {                           #D
            var item = collection.contentArrayContains(row.id, type);  #E
            if (!item) {                                               #F
                item =  type.create();                                 #F
                Ember.get(type, 'collection').pushObject(item);        #F
            }
            item.setProperties(row);                                   #G
            item.set('isLoaded', true);                                #H
        });
    });

    return Ember.get(type, 'collection');                              #I
}
```

#A: findAll() function takes a URL, the type, and the key used to fetch the data
#B: Creates a reference to this object used from within the callbacks
#C: Fetches data from the URL passed in
#D: Iterates over the result from the server
#E: Checks to see if the current object exists
#F: If it doesn't exist, creates new object and pushes it to collection
#G: Updates properties on page object
#H: Marks item as loaded
#I: Returns collection

The `findAll()` function calls the server with the URL provided. Here, you use the jQuery `$.getJSON` call, passing in the URL as well as a callback that is executed when the server responds.

Once the JSON has been successfully retrieved from the server, you use the key to get the data array and iterate over the contents of this array. Once inside the iteration, you see if this object is already loaded into the hash. If it's not you create a new instance and push it into the `collection` property.

Next, you pass the current row into the item's `setProperties()` function. This updates the item object and sets the properties retrieved from the server, before setting the `isLoaded`

property to true, telling the rest of the application that this model object has finished loading properly. Finally, you return the entire collection to the calling function.

Now that you've implemented a generic strategy to fetch data from the server, it's time to take a closer look at the `Emberfest.Talk` class.

6.2.3 Implementing the Emberfest.Talk model class

Each talk in the Ember Fest application shares a set of common properties. All talks have an id, a title, content (`talkText`), related topics, a type, who the talk was suggested by, as well as metadata that tells the Emberfest appliaction if this talks is suggested by the currently logged in user.

The following listing shows the data for the talks as it's returned from the server.

Listing 6.6 The JSON retrieved from the server

```
{"abstracts": [                                                          #A
        {                                                                #B
            "id": "05D5D5122DBA0C9E",                                    #C
            "talkTitle": "Query params …",                               #D
            "talkText": "An introduction to Ember Query…",               #D
            "talkTopics": "querystring, router, pushState",              #D
            "talkType": "20 or 35 minute talk",                          #D
            "talkByLoggedInUser": false,                                 #D
            "talkSuggestedBy": "Alex Speller"                            #D
        }
  ]
}
```

#A: Returns an array of abstracts
#B: Each talk is defined with key-values. Keys become properties on Model object
#C: The id of the talk, which will uniquely identify the talk in the application
#D: The rest of the properties are simple string values

The data format between the client and the server is standard JSON, as you would expect, with a list of talks specified as an array named `abstracts`.

Because you aren't implementing anything special in your model layer, the properties that you rEmberfestive from the server become real properties when the `Emberfest.Talk` model is instantiated by the `findAll()` function. It's the server side that mandates which properties are available to the Ember Fest application.

The data that the server returns is loaded into separate `Emberfest.Talk` models as shown in the following listing. Because you're doing most of the work in the generic `Emberfest.Model` class, the implementation for the `Emberfest.Talk` model is quite simple.

Listing 6.7 The `Emberfest.Talk` Model

```
Emberfest.Talk = Emberfest.Model.extend();                               #A

Emberfest.Talk.reopenClass({                                             #B
    collection: Ember.A(),                                               #C

    find: function(id) {                                                 #D
        return Emberfest.Model.find(id, Emberfest.Talk);                 #E
```

```
    },

    findAll: function() {
        return Emberfest.Model.findAll('/abstracts,
            Emberfest.Talk, 'abstract');                                    #F
    }
});
```

#A: Creates a model class
#B: Reopens the class definition to add the find and findAll functions
#C: Initializes collection of pages
#D: Fetches talk from cache
#E: Delegates call to Emberfest.find, including model type you expect to get returned
#F: Delegates call to Emberfest.Model, adds URL, model type you expect to get returned, and name of hash where data for this model type can be found

You define a new class called `Emberfest.Talk`, which extends from the `Emberfest.Model` class you defined previously (see listing 6.4). To add `find()` and `findAll()` as class methods, you reopen the class using the `reopenClass()` construct. To have one collection that's unique for each data type, you initialize a `collection` variable. Here, for simplicity, you specify that the `collection` property is an Ember.js array by using `Ember.A()`.

The `find()` function takes a single parameter, which is the `id` element of the model you wish to find. You delegate to the `Emberfest.Model.find()` function, but you tell it the type of class that you expect to get in return from `Emberfest.Model.find()`.

The `findAll()` function is similar, only here you specify the URL, '`/abstracts`', that `Emberfest.Model.findAll()`fetches the data from, as well as the type you expect in return (`Emberfest.Talk`) and the name of the array in which you expect to find data for the pages ('`abstracts`').

Using this approach it's easy to reuse the `Emberfest.Model` class for different model objects. The following listing shows the implementation of the `Emberfest.User` model.

Listing 6.8 The `EMBERFEST.User` Model

```
Emberfest.User.reopenClass({
    collection: Ember.A(),

    find: function(id) {
        return Emberfest.Model.find(id, Emberfest.User);                    #A
    },

    findAll: function() {
        return Emberefst.Model.findAll('/user, Emberfest.User, 'users');#B
    }
});
```

#A: Returns EMBERFEST.User objects
#B: Returns data based on the specified URL, hash key, and object type

As you can see, the `talk` and `user` model objects are quite similar, so it's easy to scale this approach out to support additional model objects that your application might need. You create a new `EMBERFEST.YourModel` class for each of the model objects that your application needs to support.

Because the model object implementations are so similar, you could take this one step further and standardize on the URL and hash keys, too.

Figure 6.4 shows the result of loading all talks from the server.

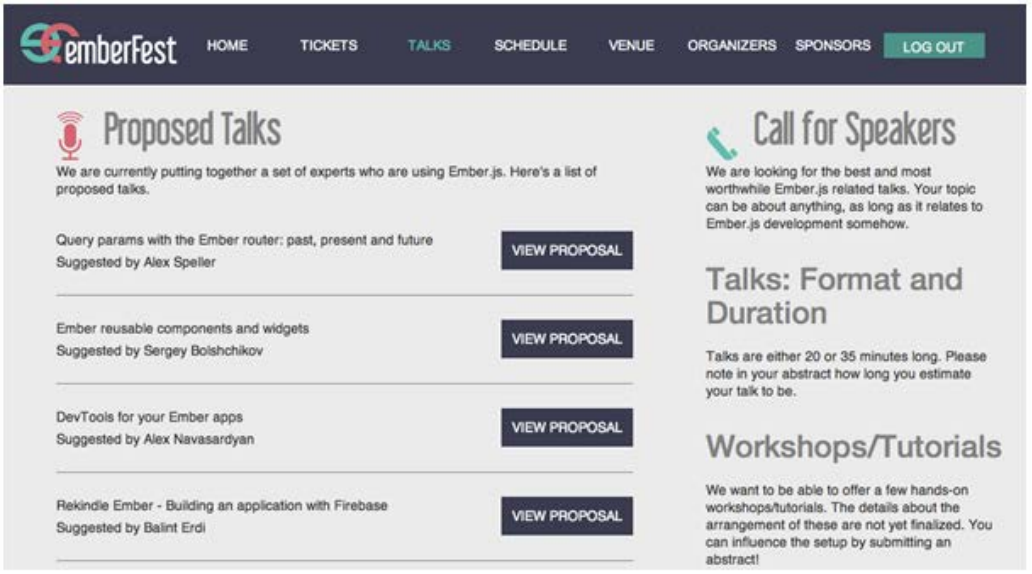


Figure 6.4 Showing all talks registered in the application in the `talks.index route`

Now that you've seen how you can create a strategy for fetching data, let's take a closer look at how you can implement data persistence via create and update.

6.3 Persisting data

To allow users to submit new talk proposals to the Ember Fest application, you implement persisting of data. You extend the strategy you've created by adding a few extra features to your setup.

Because the keyword `create` is reserved for creating new objects in Ember.js, you implement persistence via the two methods `createRecord()` and `updateRecord()`.

6.3.1 Submitting a new talk via the createRecord() function

To create a new `Emberfest.Talk` object on the client side (as opposed to the server side) you use the `createRecord()` function. This function instantiates a new model object of the given type and serializes the model to JSON before it sends the data to the server via an Ajax call. The following listing shows the contents of `Emberfest.Model.createRecord()`.

Listing 6.9 Implementing the `createRecord()` function

```
Emberfest.Model.reopenClass({
```

```
    createRecord: function(url, type, model) {                          #A
        var collection = this;                                          #B
        model.set('isSaving', true);                                    #C
        $.ajax({
            type: "POST",                                               #D
            url: url,
            data: JSON.stringify(model),                                #E
            success: function(res, status, xhr) {
                if (res.submitted) {
                    Ember.get(type, 'collection').pushObject(model);    #F
                    model.set('isSaving', false);                       #F
                } else {
                    model.set('isError', true);                         #G
                }
            },
            error: function(xhr, status, err) {
                model.set('isError', true);                             #G
            }
        });
    },
});
```

#A: Takes a URL, a type, and the model to persist
#B: Creates a local variable that you can refer to in the Ajax callback
#C: Indicates that the object is being saved currently
#D: Persists new model objects
#E: Sends the model's string representation as the data to the server
#F: Adds new talk to the collection array and resets isSaving to false
#G: If anything goes wrong, updates isError property to true

If you compare the `createRecord()` function to the `findAll()` function you notice that they have quite a few similarities. The function gets a local reference to `this`, which you'll use later inside the Ajax callback. To indicate to the users that the model they're currently watching has been sent to the server but still awaiting a response, you set the model's `isSaving` property to `true` before you issue the call to the server. You're creating a new object, so you call the server using the HTTP `POST` method. If the server responds with a successful response message, you push the newly created model onto the `collection` array and set the `isSaving` property back to `false`. If the Ajax call fails, or the server responds with an unsuccessful response, you update the model's `isError` property to `false`.

After this abstract function is implemented on the `Emberfest.Model` class, you'll add a to the `Emberfest.Talk` class, which is the class that the Ember Fest application uses, similar to the `find()` and `findAll()` functions you added previously. The following listing shows the result of adding this function to the `Emberfest.Talk` class. The `createRecord` method takes a model object as its only input parameter, and delegates to `Emberfest.Model`, adding the URL and the object type you're persisting.

Listing 6.10 Adding `createRecord` to `EMBERFEST.Talk`

```
Emberfest.Talk.reopenClass({
    collection: Ember.A(),

    find: function(id) {
```

```
        return Emberfest.Model.find(id, Emberfest.Talk);
    },

    findAll: function() {
        return Emberfest.Model.findAll('/abstracts',
            Emberfest.Talk, 'abstracts');
    },

    createRecord: function(model) {                                    #A
        Emberfest.Model.createRecord('/abstracts',
            Emberfest.Talk, model);
    }
});
```

#A: Adding the createRecord function

You delegate the call down to the `Emberfest.Model` class. In addition to the model you're persisting, you also pass in the type of model you're persisting, as well as the URL that `Emberfest.Model.createRecord()` will call.

When users of the Ember Fest website submit a talk to the system, they navigate to the `registerTalk` route, where they're presented with a form in which they enter the details of their talk and submit it to the system. After the talk is submitted, the users are forwarded to the `talks.talk` route where they can view all talks submitted to the system so far. Figure 6.5 shows the flow.

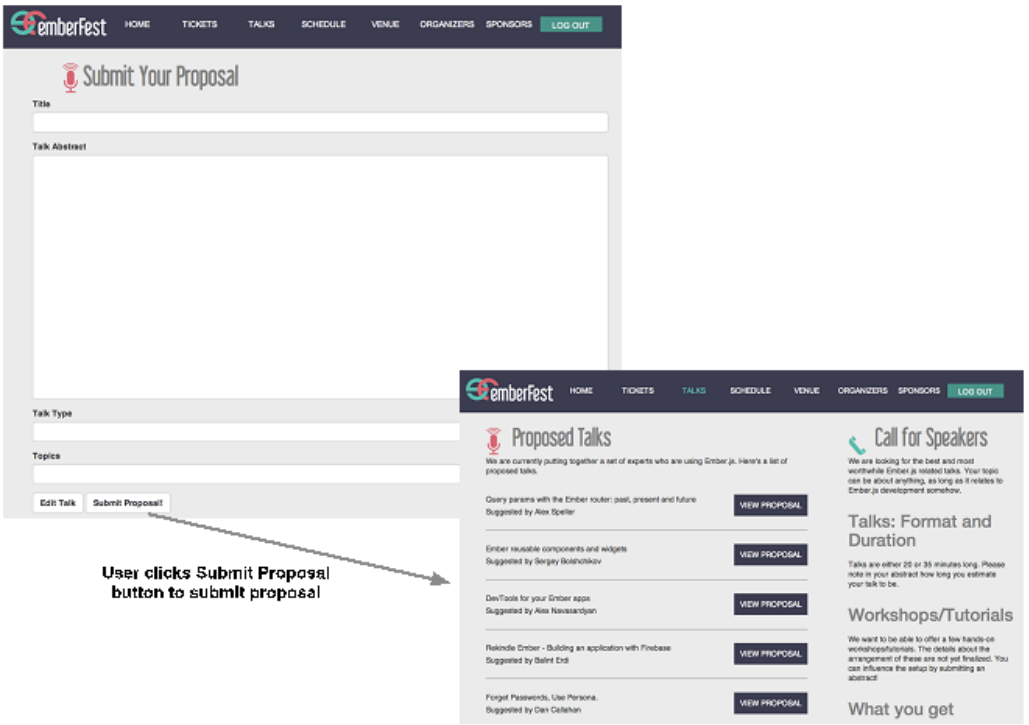


Figure 6.5 Submitting a new talk

At this point, everything is set up for you to create a new talk and submit it to the server side. The following listing shows an excerpt from the `Emberfest.RegisterTalkController.submitAbstract()` function that demonstrates how to use the newly created `createRecord()` function. You're expecting the validation of the user input to be `true`. For each failed validation, update `validated` to `false`.

Listing 6.11 Using the `createRecord()` function

```
submitAbstract: function() {
    var validated = true;

    //Validation ommitted from this listing

    if (validated) {                                                    #A
        var talkId = Math.uuid(16, 16);
        var talk = Emberfest.Talk.create({
            id: talkId,
            talkTitle: this.get('content.proposalTitle'),
            talkText: this.get('content.proposalText'),
            talkType: this.get('content.proposalType'),
            talkTopics: this.get('content.proposalTopics')
```

```
        });

        Emberfest.Talk.createRecord(talk);                          #B
        this.transitionToRoute('talks');                            #C
    }
}
```

#A: If user input is validated, persist talk to server
#B: Calls EMBERFEST.Talk.createRecord() to send model to server
#C: Redirects user to talks route

After the user input has been validated, you create a new `Emberfest.Talk` model object and initialize it with the input from the user. To persist the talk to the server, you call `Emberfest.Talk.createRecord(talk)`.

> **NOTE** Because you're dealing with user input, you must validate that the input adheres to the rules that you've defined for the system's talks/abstracts. The validation itself is omitted in this code listing but is available in the project's source code on GitHub.

After the proposed talk has been persisted, you transition the user to the talks route to show the complete list of talks submitted to the system. You redirect the user to the talks route via the transitionToRoute function.

Next, we'll take a closer look at the `updateRecord()` function.

6.3.2 Updating a talks data via the updateRecord() function

The `updateRecord()` function is implemented in a similar style as the `createRecord()` function. The difference between them is that you don't add the model to the `collection` array, as it should already be there when you issue an update. The following listing shows the implementation of the `updateRecord()` function.

Listing 6.12 Implementing `updateRecord` on `EMBERFEST.Model`

```
Emberfest.Model.reopenClass({
    updateRecord: function(url, type, model) {
        var collection = this;                                      #A
        model.set('isSaving', true);                                #B
        console.log(JSON.stringify(model));
        $.Ajax({
            type: "PUT",                                            #C
            url: url,
            data: JSON.stringify(model),                            #D
            success: function(res, status, xhr) {
                if (res.id) {
                    model.set('isSaving', false);                   #E
                    model.setProperties(res);                       #F
                } else {
                    model.set('isError', true);                 #G
                }
            },
            error: function(xhr, status, err) {
                model.set('isError', true);                     #G
            }
```

```
        })
    }
});
```

#A: Creates local variable to refer to within Ajax callback
#B: Sets model's isSaving to true
#C: Uses HTTP PUT method for updating new model object
#D: Sends Models String representation as data to server
#E: If Ajax call is successful, resets isSaving back to false
#F: Updates model object with response from server
#G: If anything goes wrong, updates isError property to true

As you can see, the `updateRecord()` function has many similarities to the `createRecord()` function. The `updateRecord()`function first finds a local variable reference to `this`, which you'll use inside the Ajax callback. Because you want to tell the user that you've sent the request to the server, but are awaiting a response, set the model's `isSaving` property to `true` before you issue the Ajax call.

The standard HTTP method to use for updating is PUT, so make sure to specify this method to the Ajax call. If the response from the server is successful and it contains the data of the updated model object, update the `isSaving` property by resetting it to `false`, before you update the model via the `setProperties()` function.

If the Ajax call fails or the server doesn't return the updated model object, update the `isError` property to `true` to indicate to the rest of the application that something went wrong while updating the model on the server. This alerts the user that you were unable to persist the proposed talk.

Once this function is implemented on the `EMBERFEST.Model` class, you add a function to the `EMBERFEST.Talk` class, the way you did with the `createRecord()` function you added earlier, as shown in the following listing. The `updateRecord` method takes a model object as its only input parameter, and delegates to `Emberfest.Model` adding the URL and the object type you're persisting.

Listing 6.13 Adding `updateRecord` on `Emberfest.Talk`

```
Emberfest.Talk.reopenClass({
    collection: Ember.A(),

    find: function(id) {
        return EMBERFEST.Model.find(id, EMBERFEST.Talk);
    },

    findAll: function() {
        return EMBERFEST.Model.findAll('/abstracts', Emberfest.Talk,
'abstracts');
    },

    createRecord: function(model) {
        EMBERFEST.Model.createRecord('/abstracts', Emberfest.Talk, model);
    },

    updateRecord: function(model) {
        EMBERFEST.Model.updateRecord("/abstracts", Emberfest.Talk, model);
```

```
    }
});
```

Here you're delegating the call down to the `Emberfest.Model` class. In addition to the model you're persisting, you also pass in the type of model you're persisting, as well as the URL `Emberfest.Model.updateRecord()` will call.

The users can edit a talk that they've submitted. Once inside the `talks.talk` route, they can click an edit button that will present a talk edit form. When the users update the talk they are forwarded to the `talks` route, showing all the talks submitted to the application. Figure 6.5 shows the flow of this action.

Figure 6.6 shows the process the user follows to edit the proposed talk, while listing 6.14, shows the code you use to call the `updateRecord()` function on `Emberfest.Talk`.

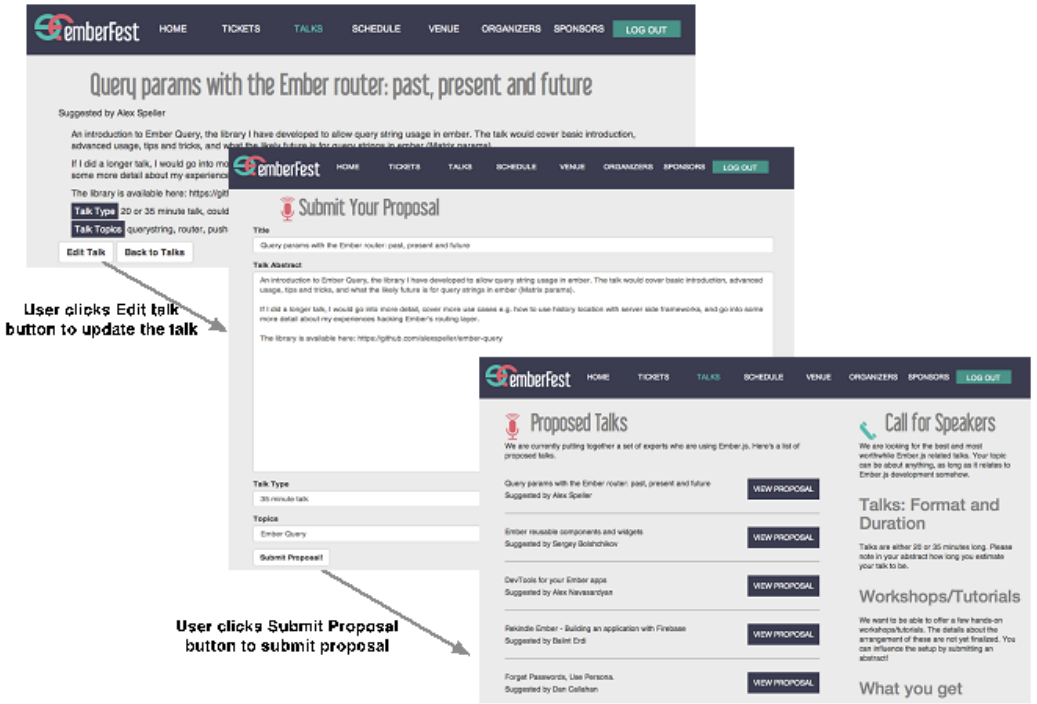


Figure 6.6 Editing a talk

At this point, everything should be ready for the user to update a talk and submit it to the server side. The following listing shows an excerpt from the `Emberfest.TalksTalkController's submitTalk()` function that shows how you can use the newly created `Record()` function.

You expect the validation of the user input to be `true`. For each failed validation you update validated to `false`. If the user input is validated, you persist the talk to the server. When the new `Emberfest.Talk` model is instantiated, you call `Emberfest.Talk.updateRecord()`to send the model to the server.

Listing 6.14 Using the `updateRecord()` function

```
submitTalk: function() {
    var validated = true;

    //Ommiting the validation from the code listing

    if (validated) {
        var talk = this.get('content');
        EMBERFEST.Talk.updateRecord(talk);

        this.transitionToRoute('talks');                                    #A
    }
}
```

#A: Redirect the user to the talks route

When the user input has been validated, you get the `content` property of the `Emberfest.TalksTalkController` (which contains a single `Emberfest.Talk` instance) and pass this along to `Emberfest.Talk.updateRecord()`.

> **NOTE** Again, because you're dealing with user input, you must validate that the input adheres to the rules that you defined for the system's talks/abstracts. The validation itself is omitted from the code listing but is available in the project's source code on GitHub.

To display the complete list of talks submitted to the system you transition the user to the `talks` route via the `transitionToRoute` function.

Now that you've seen how you can create, update, and read data from the server, there is only one operation missing. The next section shows how you can implement deletion.

6.3.3 Deleting a talk via the delete () function

The site administrator of the Ember Fest website can delete proposed talks. These could be talks that were submitted in error or any type of spam. The administrator is presented with a Delete button alongside the list of talks. When the Delete button is clicked, the Ember Fest application calls on the server to delete the talk.

Implementing deletion is similar to the functions you created previously in this chapter. You use the standard HTTP DELETE method when sending the Ajax call to the server and you remove the deleted object from the collection array once the server has indicated that the item has been successfully deleted, as shown in the following listing.

Listing 6.15 Implementing `delete()` on `EMBERFEST.Model`

```
delete: function(url, type, id) {
    var collection = this;                                                  #A
```

```
    $.ajax({
        type: 'DELETE',                                                    #B
        url: url + "/" + id,                                               #C
        success: function(res, status, xhr) {
            if(res.deleted) {
                var item = collection.contentArrayContains(id, type);      #D
                if (item) {
                   Ember.get(type, 'collection').removeObject(item);       #E
                }
            }
        },
        error: function(xhr, status, err) {
            alert('Unable to delete: ' + status + " :: " + err);           #F
        }
    });
}
```

#A: Creates a local variable that you can refer to in Ajax callback
#B: Uses HTTP DELETE method
#C: Formats URL, adds ID of deleted model
#D: Fetches object from collection array
#E: Removes deleted item from collection array
#F: If anything goes wrong, displays alert to user

As you can see the `deleteRecord()` function has many similarities to the `updateRecord()` function. The function finds a local variable reference to `this`, which you use inside the Ajax callback.

The standard HTTP method to use for deletion is DELETE, so you'll specify this method to the Ajax call. If the response from the server is successful, you fetch the item from the `collection` array and remove it. If the Ajax call fails, display an alert to the user.

After this function is implemented on the `Emberfest.Model` class, you add a function to the `Emberfest.Talk` class, as you did with the `updateRecord()` function. The following listing shows the result of adding this function to the `Emberfest.Talk` class. The `delete` method takes a model id as its only input parameter and delegates to `Emberfest.Model`, adding the URL and the object type you're persisting.

Listing 6.16 Adding `delete()` on `Emberfest.Talk`

```
Emberfest.Talk.reopenClass({
    collection: Ember.A(),


    delete: function(id) {
        EMBERFEST.Model.delete('/abstracts', Emberfest.Talk, id);
    }
});
```

Here you can see that you're delegating the call down to the `Emberfest.Model` class. In addition to the model `id` you're deleting, you also pass in the type of model you're persisting, as well as the URL `Emberfest.Model.delete()` will call.

At this point, everything should be set up for an admin user to delete a talk and submit it to the server side. The following listing is an excerpt from the

`Emberfest.TalksIndexController`'s `deleteTalk()` function that shows how to use the newly created `delete()` function.

Listing 6.17 Using the `delete()` function

```
deleteTalk: function(a) {
      Emberfest.Talk.delete(a.get('id'));
}
```

#A: Deletes model

The `deleteTalk()` function responds to the user clicking a Delete button in the application. This action function gets the model object that the user clicked as its only parameter and it passes the `id` property of this model object along to `Emberfest.Talk.delete()`.

This concludes the implementation of the server-side communication for the Ember Fest website.

6.4 Summary

This chapter has gone through one strategy for creating Create, Read, Update, and Delete (CRUD) functionality in your Ember.js application. This approach is similar to the CRUD operations of any rich web application, but it's adapted to fit into the Ember.js life cycle. Because Ember.js might issue calls to the `find()` method before `findAll()`, depending on the route through which the user enters the application, it helps to implement an identity map inside the application to minimize the number of Ajax calls that you issue to the server.

You've also seen how you can create and update models on the client side and how you can issue Ajax calls to the server to persist them. The approach taken here will be familiar to you if you have experience with writing Ajax-bound web applications. The deletion of models is also similar to what you might expect.

In this chapter you've created a simple approach to data persistence. There are many improvements that could be made to this implementation to make the implementation both more robust and easier to use. But this approach does work well for smaller applications in which a large-scale framework such as Ember Data might bring in too much overhead. Overall, though, the chapter has gone through the expectations that Ember.js has for your data layer implementation. In addition, after implementing a model-layer yourself, you should have a clearer understanding of where data-layer frameworks like Ember Data, Ember Persistence Framework and Ember Model is coming from and what kind of scenarios they're built to solve.

In the next chapter we'll look at how you can create custom components for your application.

7

Writing custom components

This chapter covers

- An introduction to writing custom components
- Implementing a selectable-list component
- Implementing a tree-view component
- Integrating Ember.js with Twitter Bootstrap

The ability to write custom components is a key factor in most GUI frameworks because it allows you to build parts that can be reused in the same application as well as across applications. Most applications have components that share a similar functionality throughout. Features such as selectable lists, buttons that integrate with Twitter, or tree-based components are a few examples of situations in which implementing custom components can make sense in your application.

Ember.js's use of Handlebars.js templates, easy integration of third-party JavaScript libraries, and strong binding system makes it an excellent framework for building custom components. This chapter presents a few of the custom components that have been written for the Montric project and discusses how they're structured to achieve different goals in your application. When combined, some of these components work together to form complex functionality. You'll structure your components to be as small and specific as possible so that you can reuse them individually as well as combine them as building blocks to create more complex components.

Figure 7.1 shows the parts of the Ember.js ecosystem this chapter examines—ember-application, ember-views, container, ember-handlebars, ember-handlebars-compiler, and Handlebars.js.

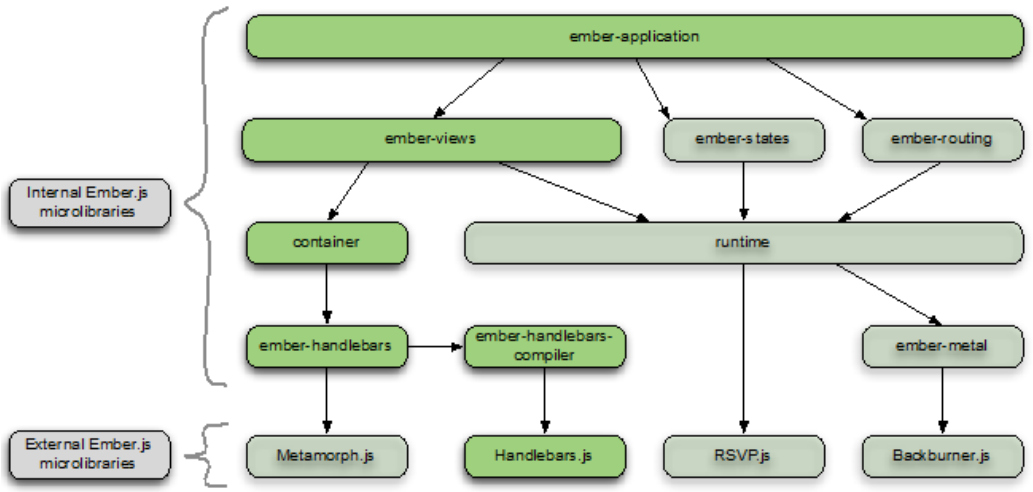


Figure 7.1 The parts of Ember.js you'll be working on in this chapter.

You'll begin by creating a `selectable-list` component, which is similar to the selectable list you implemented in chapter 1 for the Notes application. This time, however, you'll split the functionality of the selectable list in three different self-contained components. Next, you'll learn how to create a hierarchical tree-based component in which the leaf nodes (the nodes without subnodes) are selectable via a check box.

7.1 About Ember custom components

The technical description of a component would be something along the lines of "an identifiable part of a larger program that provides a particular function or a group of related functions." You can think of a component as an independent part of your application that you can reuse in multiple places of your application without modification. If you do build your components generally enough, they may also serve a purpose outside of your initial application.

When Ember components were built in the later release candidates of Ember.js, it consolidated a group of functions that would otherwise be directed at custom view, handlebars templates, and custom handlebars expressions. Now, your components generally consist of two items, a handlebars template and a component class. In fact, only a template is necessary for the simplest of components.

If you're used to large, server-side frameworks like JavaServer Faces or Microsoft ASP.NET MVC, you'll be surprised that your Ember.js custom components are built with relatively little code and with relatively few moving parts. This is a true testament to the power that Ember.js provides you as a web developer!

Let's get started with your first custom component, the selectable list.

7.2 Implementing a selectable list

The selectable-list component works as a list from which the user can select an item. In addition to selecting an item, the the user can delete an item via a handy Delete button.

The component displays a list of items, and each item appears in a separate row. The component, with no item selected, is shown in figure 7.2.

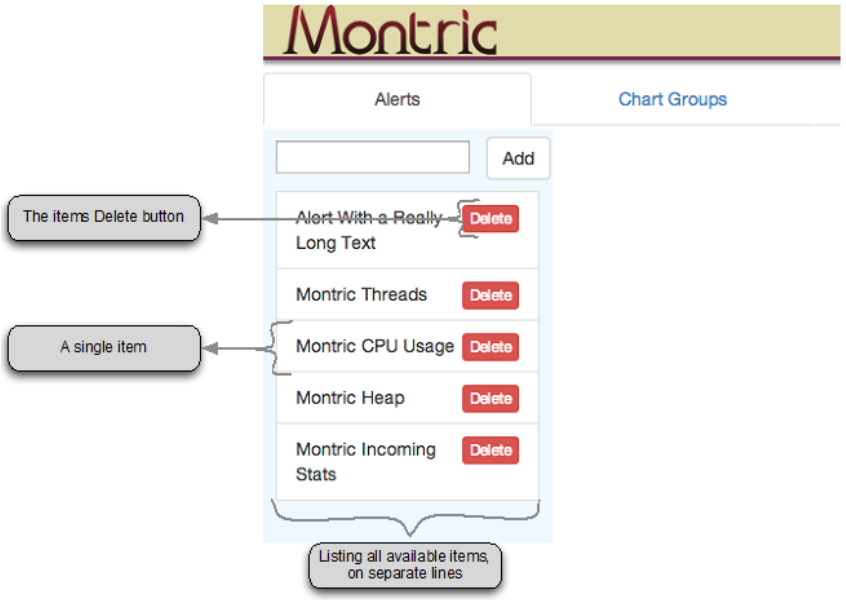


Figure 7.2 The resulting view of the selectable list component with no item selected

When the user clicks a row with the mouse, you want to highlight the selected row. The component with an item selected is shown in figure 7.3.

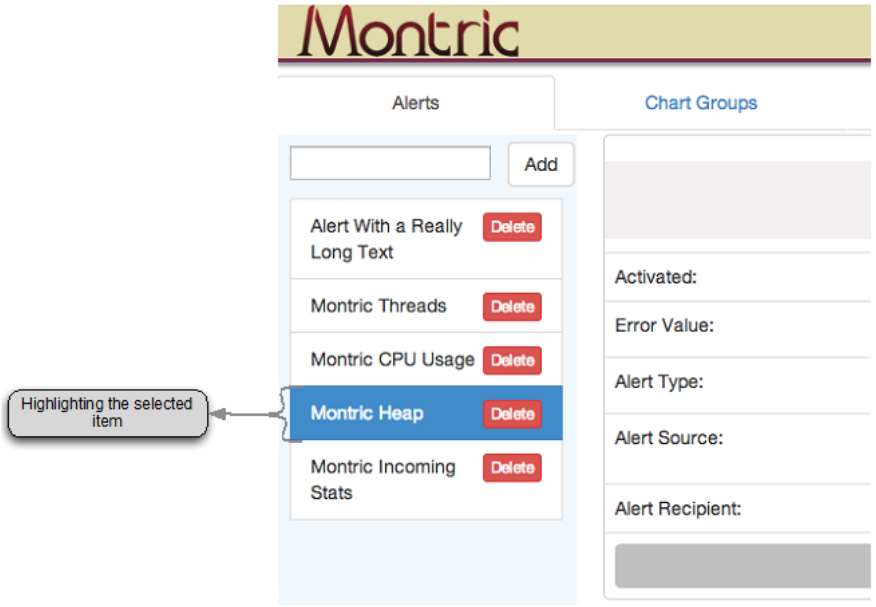


Figure 7.3 Selecting a row in the selectable list highlights the item

This component has one more part. When the user clicks the Delete button, you want to show a modal panel to the user to prompt the user to confirm deletion. The `delete-modal` panel is shown in figure 7.4.

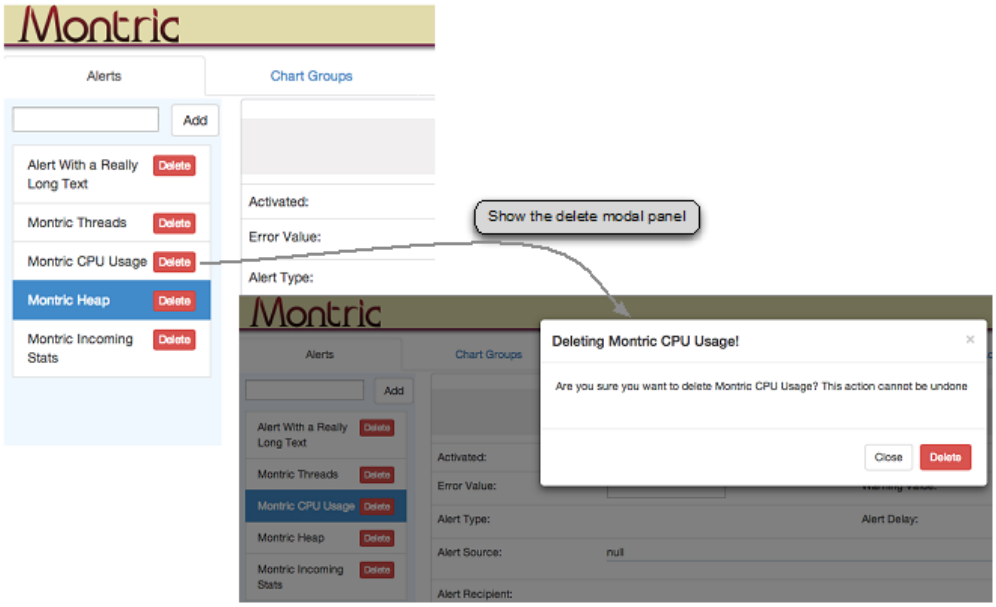


Figure 7.4 Showing the delete modal panel when the user clicks on the Delete button

As I mentioned previously, you build your components as small and specific as possible. In fact, the functionality that you've seen in the previous three figures consists of a total of three components:

- *A selectable-list component*—Displays the list of items from which the user can select and is rendered using Twitter Bootstraps List Group CSS markup
- *A selectable-list-item component*—Displays each individual item in the list and is also rendered using Twitter Bootstraps List Group CSS markup
- *A delete-modal component*—Displays the modal panel that prompts the user to confirm the item's deletion

Now that you have a clearer picture of the components you're building, let's begin by implementing the selectable-list component. But before moving on, it's useful to review the Montric router definition in the following listing to note which route is involved.

Listing 7.1 The Montric router definition

```
Montric.Router.map(function () {
    this.resource("main", {path: "/"}, function () {
        this.resource("login", {path: "/login"}, function () {

        });
        this.route('charts');
        this.resource("admin", {path: "/admin"}, function() {
```

```
            this.resource('alerts', {path: "/alerts"}, function() {      #A
                this.route('alert', {path: "/:alert_id"});               #B
            });
            this.route('chartGroups');
            this.route('mainMenu');
            this.route('accessTokens');
            this.route('accounts');
            this.route('alertRecipients');
        });
    });
});
```

#A: Adds selectable-list component to alerts resource
#B: Transitions to alerts.alert route

You add your selectable-list component to the `alerts` route, which means that you add it to the alerts.hbs file. When the user selects a node from the list (an alert), the user transitions to the `alerts.alert` route. The URL is updated, allowing the user to bookmark and get direct access to a selected alert. Now that you know where your component will be used inside your application, let's begin the implementation.

7.2.1 Defining the selectable-list component

An initial implementation of the selectable-list component is shown in the following listing.

Listing 7.2 The components/selectable-list template

```
<div class="list-group mediumTopPadding" style="width: 95%;">           #A
    {{#each node in nodes}}                                             #B
        <div class="list-group-item">{{node.id}}</div>                  #C

    {{/each}}
</div>
```

#A: Creates list-group div element
#B: Renders selectable-list-item component for each node
#C: Prints list of node ids

The first thing to notice in this listing is the path and the name of the template. Any components must start with components/. In addition, the name of the component must contain a dash (-) character.

> **Why the strange naming requirement?**
>
> The use of the dash has a logical explanation. To prevent a name clash with the future and final WebComponents specification (to be ratified by the TC39), the Ember.js team decided to require that Ember.js components contain a dash in their names.

You may have noticed that the code in this listing looks like a standard Handlebars.js template, and you're absolutely right. This is one of the key strengths of Ember components. But, whereas a standard Handlebars.js template is backed by an Ember.js view that has access to the current context in the application (like the controller and the route), a component can be considered a special kind of view that has no access to the context in which it lives. Ember.js won't inject the current controller into a component. This is what helps make Ember components behave like standalone, complete pieces of reusable functionality.

Currently, your component doesn't do much other than iterate over each of the elements in the `node` property and print out each node's `id` property. Nevertheless, let's examine how to use this brand-new component in your application. When your application initializes, Ember.js finds all components in the `components/`directive and sets up custom Handlebars.js expressions for them. In this case, the component is accessible via the `{{selectable-list}}` expression, as shown here for your component from the alerts.hbs file:

```
{{selectable-list nodes=controller.model}}
```

That was easy enough. To use your selectable-list component, you use the `{{selectable-list}}` Handlebars.js expression. But because the component won't have access to the current context, you need to pass any data to it manually. In this example, you pass the controller's `model` property to the component's `nodes` property. That's all you need to create a component.

Currently, you've implemented the first part of your component, and you can now list each of your nodes. Figure 7.5 shows your progress so far.

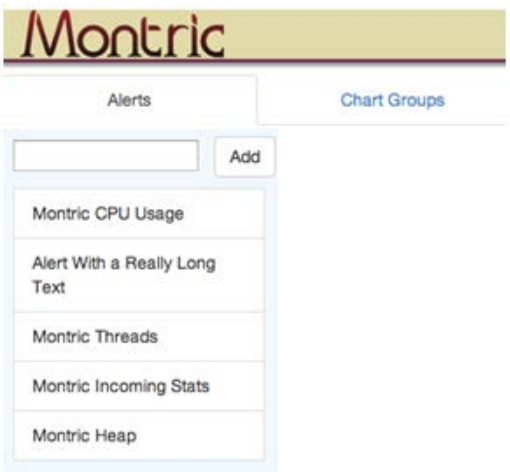


Figure 7.5 Listing each of the alerts registered for the logged in account

Next up, you'll add the functionality to select an item in the list and transition from the `alerts` route to the `alerts.alert` route.

7.2.2 The selectable-list-item component

To separate the concerns between the selectable list and each of the items in the list, create a new component whose sole responsibility is to render a single list item. The following listing shows the updated selectable-list component template.

Listing 7.3 Selecting a node and transitioning to the `alerts.alert` route

```
<div class="list-group mediumTopPadding" style="width: 95%;">
    {{#each node in nodes}}
        {{#if linkTo}}                                              #A
            {{#linkTo linkTo node tagName=div                       #B
              classNames="list-group-item"}}
                {{selectable-list-item node=node action="showDeleteModal"
                  param=node textWidth=textWidth}}                  #C
            {{/linkTo}}
        {{/if}}
    {{/each}}
</div>
```

#A: Checks whether linkTo property is defined
#B: Links to route specified by linkTo, and makes link a div element
#C: Prints each node via selectable-list-item

This selectable-list component template has a few new concepts. First, if the component has a `linkTo` property, you want to link to the route that this property specifies. You use the standard `{{#linkTo}}` Handlebars.js expression to achieve this. In addition, you use the `{{#linkTo}}` expression's `tagName` property to specify that you want the link to be rendered as a `div` element and the `classNames` property to specify the Twitter Bootstrap `list-group-item` CSS property.

The most important point is that you've moved the rendering of each of the nodes to a second component named selectable-list-item. The reason for this will become apparent soon, but let's go ahead and take a look at the implementation of this component template. The following listing shows the selectable-list-item template code.

Listing 7.4 The components/selectable-list-item component

```
<div {{bind-attr width=textWidth}} {{bind-attr maxWidht=textWidth}}>    #A
    {{node.id}}                                                        #B
</div>
```

#A: Adds div element with attributes width and maxWidth
#B: Prints id

You print out a `div` element with the attributes `width` and `maxWidth` set, and you print out the `id` property of the `node` element.

For your component to work as intended, you need to tell the selectable-list component the width of the text inside the component as well as which route to link to, as shown here:

```
{{selectable-list nodes=controller.model textWidth=75
  linkTo="alerts.alert"}}
```

After you add these two new properties, you can now click an item in the list to select it. When an item is selected, the user transitions to the `alerts.alert` route and the application displays the selected alert at the right of the selectable list.

Figure 7.6 shows the progress so far.

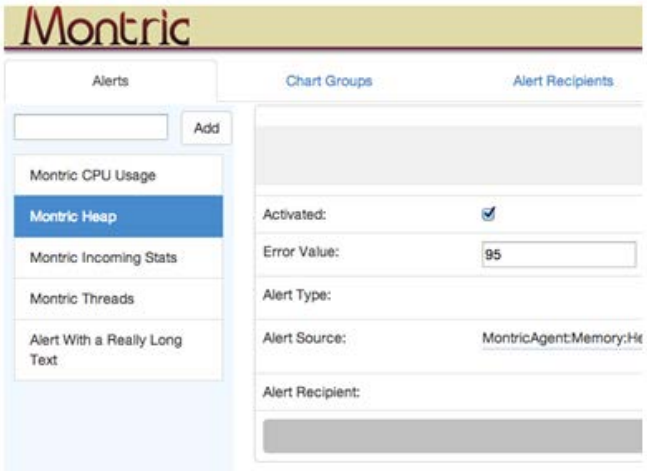


Figure 7.6 Selecting an item to transition to the `alerts.alert` route

You've created a component that has no actions added to it. Next, you want to add a Delete button to your selectable-list-item component that you can click to delete the item. To achieve this, you create a new component called a delete-modal.

7.2.3 The delete-modal component

The delete-modal component is responsible for showing a Twitter Bootstrap modal panel. This panel prompts users to confirm that they're, in fact, interested in deleting the node. As you would expect, the modal panel has two buttons: the Close button cancels the deletion, and the Confirm button confirms the deletion.

The delete-modal component template is shown in the following listing.

Listing 7.5 The delete-modal component template

```
<div class="modal-dialog">
    <div class="modal-content">
        <div class="modal-header">
            <button type="button" class="close" data-dismiss="modal"
                aria-hidden="true">&times;</button>
            <h4 class="modal-title">Deleting {{item.id}}!</h4>          #A
        </div>
        <div class="modal-body">
            <p>Are you sure you want to delete {{item.id}}?
                This action cannot be undone</p>
```

```
                    #B
        </div>
        <div class="modal-footer">
            <button type="button" class="btn btn-default"
                data-dismiss="modal">Close</button>                    #C
            <button type="button" class="btn btn-danger"
                {{action "deleteItem"}}>Delete</button>                #D
        </div>
    </div><!-- /.modal-content -->
</div><!-- /.modal-dialog -->
```

#A: Displays id of item being deleted in header of modal panel
#B: Displays id of item being deleted in body of modal panel
#C: Adds Close button to cancel deletion
#D: Adds Delete button to confirm deletion

Most of the code here is standard Twitter Bootstrap code. If you're unfamiliar with the structure presented here, head over to the Twitter Bootstrap project website to get an explanation of the HTML markup.

> **NOTE** Twitter Bootstrap is a common Graphical User Interface library found on lots of websites across the world. You can read more, or download Bootstrap, via the projects website http://getbootstrap.com

The `delete-modal` component has one property named `item` that triggers the action `deleteItem` when the user clicks the Delete button. But because the component won't have access to the outer context that it's part of, you may be wondering where you can catch this action to perform the deletion.

For each of your component templates, Ember.js instantiates a default `Component` object for you automatically. To catch the `deleteItem` action, you need to override the default `DeleteModalComponent` class. The code for the `Montric.DeleteModalComponent` is shown in the following listing.

Listing 7.6 The `Montric.DeleteModalComponent` class

```
Montric.DeleteModalComponent = Ember.Component.extend({                #A
    classNames: ["modal", "fade"],                                     #B

    actions: {                                                         #C
        deleteItem: function() {                                       #D
            var item = this.get('item');
            if (item) {                                                #E
                item.deleteRecord();                                   #F
                item.save();                                           #F
                $("#" + this.get('elementId')).modal('hide');          #G
            }
        }
    }
});
```

#A: Creates new DeleteModalComponent that extends Ember.Component
#B: Adds Twitter Bootstrap modal and fade CSS class names
#C: Implements action's hash to catch deleteItem action

#D: Called when user clicks Delete button
#E: If item is defined, deletes it
#F: Deletes record via Ember Data
#G: Closes modal panel

The `Component` class follows the naming convention you've become accustomed to in Ember.js. The name of the class is the same as the component template, but each of the dashes in the template name is removed. The name is then camelized, and the string "Component" is added to the end of the name. It's also important to note that any component extends the `Ember.Component` class, which ensures that the component won't get passed in the outer scope from the application.

After the component class is created, you can catch the component action as you would inside a controller or a route, via the action's hash. Here, you implement a function named `deleteItem` that's triggered whenever the Delete button is clicked and the `deleteItem` action is fired. Inside the `deleteItem` function, you ensure that the component has an `item` property and that it contains a `non-null` value before deleting it. After the item is deleted, you close the modal panel.

To wrap up the functionality of your three components, you still need to complete the following:

- Add a Delete button to the selectable-list-item component template
- Tell the delete-modal component which item to delete

7.2.4 Deleting an item using the three components

Before you can delete an alert from Montric, you need to add a Delete button to the selectable-list-item component template. The updated template is shown in the following listing.

Listing 7.7 Adding a Delete button to the selectable-list-item component template

```
<button class="…" {{action "showDeleteModal"}}>Delete</button>          #A
<div {{bind-attr width=textWidth}} {{bind-attr maxWidht=textWidth}}>
    {{node.id}}
</div>
```

#A: Adds Delete button

You've added an action to the selectable-list-item component template. Now, you need to create a `Montric.SelectableListItemComponent`, as shown in the following listing.

Listing 7.8 The `Montric.SelectableListItemComponent`

```
Montric.SelectableListItemComponent = Ember.Component.extend({          #A
    actions: {                                                          #B
        showDeleteModal: function() {                                   #C
            $('#deleteAlertModal').modal('show');                       #D
            this.sendAction('action', this.get(node));                  #E
        }
    }
});
```

#A: Creates new SelectableListItemComponent, extending Ember.Component

#B: Implements action's hash to catch any actions the component fires
#C: Implements showDeleteModal action
#D: Shows modal panel
#E: Sends action with context

The code for this component is similar to the delete-modal component, but one important concept has been added. Notice that you get the `node` property and pass that to the `this.sendAction()` function. The `sendAction()` function is how a component sends actions out of the component and to the outside application. You use this so the `selectable-list-item` component can send an action to the outside application. Because the selectable-list-item component is defined inside the selectable-list component, this is where your action is sent. But before you can catch the `showDeleteModal` action in the selectable-list component, you need to update the selectable-list template slightly. The updated template is shown in the following listing.

Listing 7.9 The updated selectable-list component template

```
<div class="list-group mediumTopPadding" style="width: 95%;">
    {{#each node in nodes}}
        {{#if linkTo}}
            {{#linkTo linkTo node tagName=div
              classNames="list-group-item"}}
                {{selectable-list-item node=node action="showDeleteModal"
                  textWidth=textWidth}}                          #A
            {{/linkTo}}
        {{/if}}

    {{/each}}
</div>

{{delete-modal id="deleteAlertModal" item=nodeForDelete}}        #B
```

#A: Adds action to selectable-list-item expression
#B: Adds delete-modal expression to selectable-list component

You've added two new things to the selectable-list component template: an `action` property to the selectable-list-item expression and a delete-modal expression that renders the delete-modal panel.

Inside the `action` property, you refer to the action that the selectable-list-item fires whenever the `sendAction('action')` function is called in that component.

Note that you give the delete-modal an `id` as well as an `item`. You haven't seen the `nodeForDelete` property yet, but when you implement the action `showDeleteModal` for the selectable-list component, you'll see where this property comes from. The following listing shows the new `Montric.SelectableListComponent`.

Listing 7.10 The `Montric.SelectableListComponent`

```
Montric.SelectableListComponent = Ember.Component.extend({              #A
    nodeForDelete: null,

    actions: {                                                          #B
```

```
        showDeleteModal: function(node) {                              #C
            if (node) {
                this.set('nodeForDelete', node);                       #D
            }
        }
    }
});
```

#A: Creates SelectableListComponent that extends Ember.Component
#B: Implements action's hash to catch any actions the component fires
#C: Passes node selected by user
#D: Assigns node to component's nodeForDelete property

You now see where the `nodeForDelete` property comes from. Whenever the user clicks the Delete button of a node, the selectable-list-item's `showDeleteModal` action is triggered. This, in turn, fires the `showDeleteModal` action on the selectable-list component, setting the node the user wanted to delete in the selectable-list component's `nodeForDelete` property. Because the `nodeForDelete` property is passed in and bound to the delete-modal component's `item` property, the modal panel can show the user which item will be deleted when the Delete button is clicked. The `deleteItem` action of the delete-modal component deletes the item and closes the modal panel.

This is a good time to recap what you've implemented. Figure 7.7 shows the relationships and the actions that are called for each of your three components.

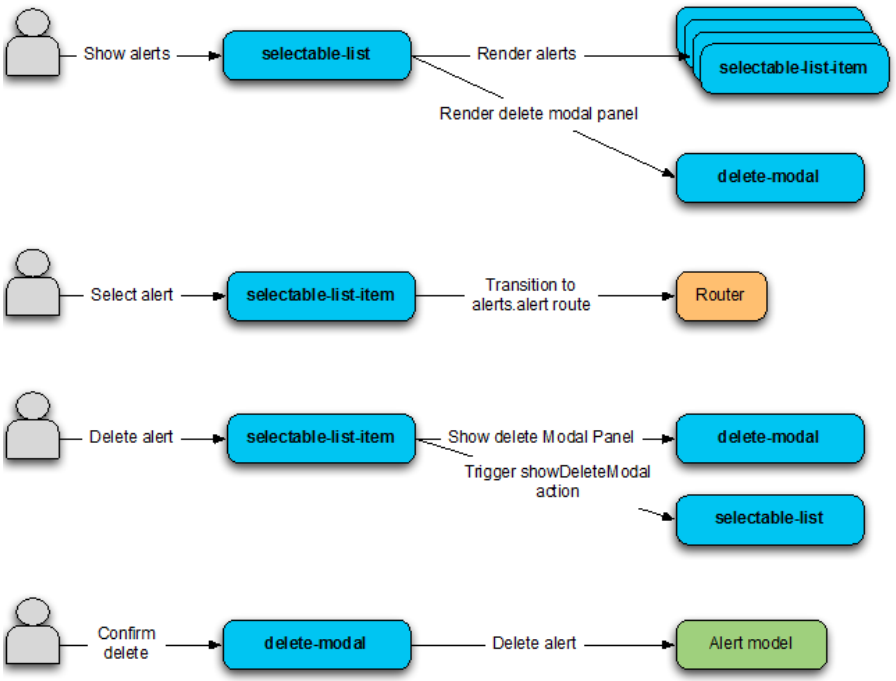


Figure 7.7 An overview of the three components, their relationships, and actions triggered

Now that you've seen how to create three components and combine them to create complex functionality, let's see how you can implement a hierarchical component.

7.3 Implementing a tree menu

Implementing a hierarchical component is slightly more complex than the list example you created. When finished, your tree menu will have the following functionality:

- The ability to expand and collapse nodes that have children so you can navigate in the hierarchical structure
- Indentation at each level to visually indicate the hierarchical structure
- A disclosure triangle at each node to visually indicate whether the node is expanded or collapsed
- The ability to support a single selection or multiple selections from the menu
- The ability to add an icon to the leaf nodes

Figure 7.8 shows how the tree structure looks.

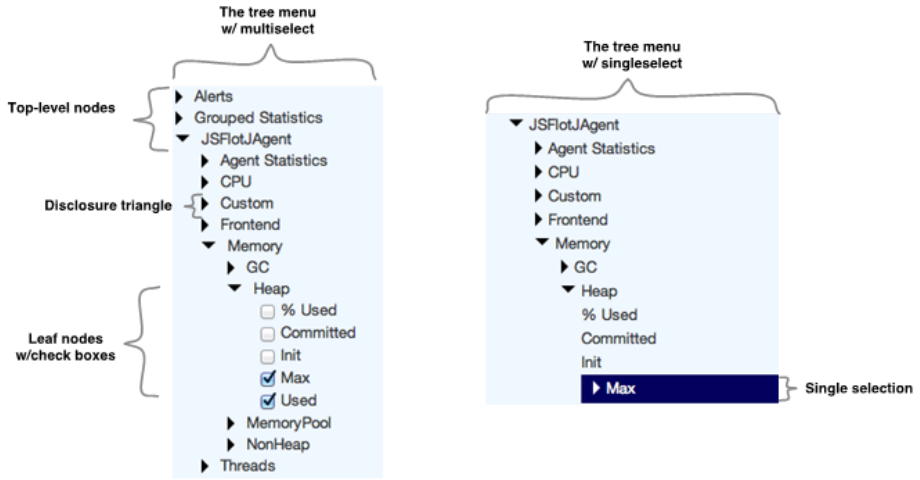


Figure 7.8. The tree menu with multiple selections (at left) and a single selection (at right).

In this section, you'll look at the data model used for the tree model as well as the components required for the tree model to render and function properly.

7.3.1 The tree-menu data model

Before you get going on the code that makes up the tree menu, let's take a look at the underlying data model that the component uses, as shown in the next listing.

Listing 7.11 The tree-menu data model

```
EurekaJ.MainMenuModel = DS.Model.extend({                              #A
    name: DS.attr('string'),
    nodeType: DS.attr('string'),
    parent: DS.belongsTo('mainMenu'),                    #B
    children: DS.hasMany('mainMenu'),                    #C
    chart: DS.belongsTo('chart'),

    isSelected: false,                                                 #D
    isExpanded: false,                                                 #D

    hasChildren: function() {                                          #E
        return this.get('children').get('length') > 0;
    }.property('children'),

    isLeaf: function() {                                               #E
        return this.get('children').get('length') == 0;
    }.property('children')
});
```

#A: Based on Ember Data model object
#B: Single parent, with one-to-one binding
#C: Zero or more children, with one-to-many binding
#D: Used internally in tree menu
#E: Helper property for template

You use an Ember Data model object as the data model you send to the tree-menu component. A few things are important in the model. First, the server sets up the links between the nodes via the `parent` and `children` properties. Ember Data ensures that the model objects are connected as expected after the data has been loaded from the server. In addition, you've defined two helper properties, `hasChildren` and `isLeaf`, that you use inside the component to make the template shorter and simpler. The component is built up from two subcomponents, a tree-menu component and a tree-menu-item component. You begin by implementing the multiselect functionality, which you'll extend later to also support single selection.

7.3.2 Defining the tree-menu component

The tree-menu component consists of a single template, components/tree-menu.hbs. This component's only feature is to render each of the top-level nodes. The following listing shows the tree-menu component template.

Listing 7.12 The components/tree-menu.hbs component template

```
{{#each node in rootNodes}}                                            #A
    {{tree-menu-node node=node}}                                       #B
{{/each}}
```

#A: Iterates over each root node
#B: Renders component for each node

The implementation of the tree-menu component template is simple and should be self-explanatory. You don't have to implement a component class for this component because, for now, no actions are triggered; the default implementation that Ember.js provides is sufficient.

With that out of the way, let's move on to the tree-menu-item component.

7.3.3 Defining the tree-menu-item and tree-menu-node components

The tree-menu-item component is a bit more complex because it needs to support the rendering of the correct disclosure triangle as well as set both the `isExpanded` and `isSelected` properties of the node it's representing. The following listing shows the tree-menu-item component template.

Listing 7.13 The components/tree-menu-item.hbb component template

```
{{#if node.hasChildren}}                                          #A
    {{#if node.isExpanded}}                                       #B
        <span class="downarrow" {{action "toggleExpanded"}}></span> #B, C
    {{else}}
        <span class="rightarrow" {{action "toggleExpanded"}}></span> #A, C
    {{/if}}

    <span {{action "toggleExpanded"}}>{{node.name}}</span>        #C
{{else}}
    {{view Ember.Checkbox checkedBinding="node.isSelected"}}      #D

    <span {{action "toggleSelected"}}>{{node.name}}</span>        #D
{{/if}}

{{#if node.isExpanded}}
    {{#each child in node.children}}                              #E
        <div style="margin-left: 22px;">                          #E
            {{tree-menu-node node=child}}                         #E
        </div>
    {{/each}}
{{/if}}
```

#A: Renders right-pointing disclosure triangle (collapsed)
#B: Renders down-pointing disclosure triangle (expanded)
#C: Fires toggleExpanded
#D: Renders selectable check box, binds checked property to isSelected property, and fires toggleSelected
#E: Renders each node child as new tree-menu-node component with left-margin space added

This component template has a lot going on. You fire two actions, `toggleExpanded` and `toggleSelected`, and you, therefore, need to override the default `Montric.TreeMenuNodeComponent` to catch these actions (see listing 7.14).

Next, you check whether the node has children. If so, you render a disclosure triangle and the name of the node. Both the disclosure triangle and the node name are clickable and fire the `toggleExpanded` action.

If the node doesn't have children, the node is a leaf node and may be selected by the user. In this case, you render a check box and the name of the node. Both the check box and the name of the node are clickable and fire the `toggleSelected` action.

A node is expanded if the `isExpanded` property is `true`. You need to render the node's children in a similar manner. You iterate over each of the nodes in the `children` property and

render them as new tree-menu-item components. This is what makes the component hierarchical in nature.

Let's take a look at the component's class definition. The following listing shows the `Montric.TreeMenuNodeComponent`.

Listing 7.14 The `Montric.TreeMenuNodeComponent` class

```
Montric.TreeMenuNodeComponent = Ember.Component.extend({                #A
    classNames: ['pointer'],

    actions: {                                                          #B
        toggleExpanded: function() {                                    #C
            this.toggleProperty('node.isExpanded');
        },

        toggleSelected: function() {                                    #D
            this.toggleProperty('node.isSelected');
        }
    }
});
```

#A: Creates SelectableListComponent that extends Ember.Component
#B: Implements action's hash to catch any actions the component fires
#C: Implements toggleExpanded action
#D: Implements toggleSelected action

The implementation of the `TreeMenuNodeComponent` should be familiar to you. As you've seen before, you implement the action's hash, which contains functions for each of the actions that you want to catch. The `toggleExpanded` function does what it advertises and toggles the node's `isExpanded` property between `true` and `false`. Similarly, `toggleSelected` toggles the node's `isSelected` property between `true` and `false`. Now that you have the component working for multiselection, let's build in the single selection functionality.

7.3.4 Supporting single selections

You've now implemented the necessary functionality for a multiselect tree component. Figure 7.9 shows the progress and the component structure so far.

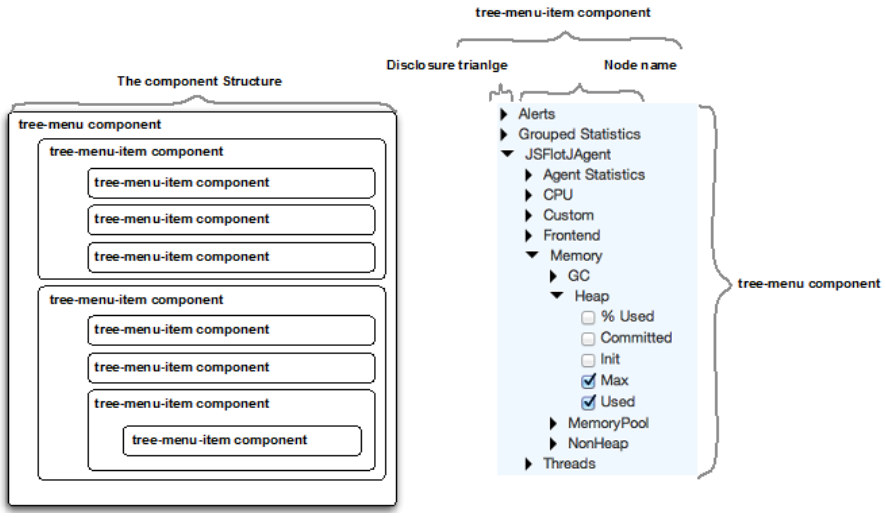


Figure 7.9 The relationships between the tree-menu components

To support single selection, you first need to add a flag that tells the component whether you want to allow multiple selections or only a single selection. You also need to make sure that this property (`allowMultipleSelections`) gets passed down to the tree-menu-node component. Although we were able to rely on simply toggling the underlying model's `isSelected` property when you want to implement multiple selections, in the case of single selections, we want the component to assign the selected item to the property that we assign to the `tree-menu` component's `selectedNode` property. As with the `allowMultipleSelection` property, you need to send the `selectedNode` property down to each of the tree-menu-node components as well. The following listing shows the updated tree-menu component template.

Listing 7.15 The updated components/tree-menu.hbs component template

```
{{#each node in rootNodes}}
    {{tree-menu-node node=node
      allowMultipleSelections=allowMultipleSelections action="selectNode"
      selectedNode=selectedNode}}                                    #A
{{/each}}
```

#A: Passes two new properties and an action to the tree-menu-node component

You pass the `allowMultipleSelections` and `selectedNode` properties to the tree-menu-node component. In addition, you tell the tree-menu-node component that you expect that the `selectNode` action may be fired from the tree-menu component.

Next, you need to update the tree-menu-node component template to make use of these new properties. The following listing shows the updated component template.

Listing 7.16 The updated components/tree-menu-node.hbs component template

```
{{#if node.hasChildren}}
    {{#if node.isExpanded}}
        <span class="downarrow" {{action "toggleExpanded"}}></span>
    {{else}}
        <span class="rightarrow" {{action "toggleExpanded"}}></span>
    {{/if}}

    <span {{action "toggleExpanded"}}>{{node.name}}</span>
{{else}}
    {{#if allowMultipleSelections}}                                    #A
        {{view Ember.Checkbox checkedBinding="node.isSelected"}}

        <span {{action "toggleSelected"}}>{{node.name}}</span>
    {{else}}                                                           #B
        <span {{action "selectNode" node}}
          {{bind-attr class=isSelected}}>{{node.name}}</span>          #C
    {{/if}}
{{/if}}

{{#if node.isExpanded}}
    {{#each child in node.children}}
        <div style="margin-left: 22px;">
            {{tree-menu-node node=child
              action="selectNode" selectedNode=selectedNode}}          #D
        </div>
    {{/each}}
{{/if}}
```

#A: If multiple selections allowed, continues as before
#B: If only single selection allowed, doesn't render check box
#C: When user clicks leaf node, fires selectNode action
#D: Passes selectedNode property and selectNode action to child nodes

You add a check that makes sure you render the template as before whenever `allowMultipleSelections` is `true`. If, however, `allowMultipleSelections` is `false`, you fire the `selectNode` action when the user clicks a leaf node. In addition, if the current leaf node is the node that's currently selected, you mark that node as blue by appending the CSS class `is-selected`.

Finally, you need to pass the `selectNode` action and the `selectedNode` property to any child nodes that are shown in the template.

Note, though, that you now need a function named `isSelected` in the `Montric.TreeMenuNodeComponent` class, and you catch the `selectNode` action in the tree-menu component. To do this, you'll override the default `Montric.TreeMenuComponent` and expand the `Montric.TreeMenuNodeComponent`. The following listing shows the updated `Montric.TreeMenuNodeComponent`.

Listing 7.17 The updated `Montric.TreeMenuNodeComponent`

```
Montric.TreeMenuNodeComponent = Ember.Component.extend({
    classNames: ['pointer'],
```

```
    actions: {
        toggleExpanded: function() {
            this.toggleProperty('node.isExpanded');
        },

        toggleSelected: function() {
            this.toggleProperty('node.isSelected');
        },

        selectNode: function(node) {                                    #A
            this.sendAction('action', node);
        }
    },

    isSelected: function() {                                            #B
        return this.get('selectedNode') === this.get('node.id');
    }.property('selectedNode', 'node.id')
});
```

#A: Fires action out of component
#B: Returns boolean indicating if node is selected node

So far, so good. You haven't added any code you haven't seen before, so let's move on to the new `Montric.TreeMenuComponent` definition, shown in the following listing.

Listing 7.18 The new `Montric.TreeMenuComponent`

```
Montric.TreeMenuComponent = Ember.Component.extend({                    #A
    classNames: ['selectableList'],

    actions: {                                                          #B
        selectNode: function(node) {                                    #C
            this.set('selectedNode', node.get('id'));                   #D
        }
    }
});
```

#A: Creates TreeMenuComponent that extends Ember.Component
#B: Implements hash to catch any actions the component fires
#C: Takes node selected by user as input
#D: Updates selectedNode property with id of node selected by user

Here, you catch the `selectNode` action, which receives, as input, the node that the user clicked. You then assign that node's `id` to the `selectedNode` property. This property is bound to the value that you gave the tree-menu component when you created it, so you can update the user interface directly to tell the user which node is selected.

The only thing left is to add the updated tree-menu component to the alert.hbs template. The following code shows how you can update the Handlebars.js expression to specify single-node selection and to map the selected node to the `alertSource` property on the currently selected Alert model:

```
{{tree-menu rootNodes=controllers.admin.rootNodes
  allowMultipleSelections=false selectedNode=alertSource}}
```

Figure 7.10 shows the single-selection tree menu in action.

Figure 7.10 The updated single-selection tree menu

As you can see we have now implemented a single tree menu component that allows us to specify whether or not the user will be able to only select a single node, or it the user should be allowed to select multiple nodes. If the user is able to select multiple items, then when the user selects an item, that item's `isSelected` property is set to true. If the user is only able to select a single item, then the selected item will be assigned to the the `selectedNode` property that the user passed in to the tree-menu component.

7.4 Summary

The ability to write customized and specialized components that you can easily use in multiple places in your application is an important feature of any frontend framework. The fact that Ember.js makes it easy to include almost any third-party widget library, combined with the ability to use the standard Handlebars.js template functionality, makes Ember.js a suitable framework for writing both simple and complex standalone components that encapsulate the logic and templates that are required.

This chapter has shown how you can create standalone custom components that are built with views and templates, as well as custom components that use the third-party frontend library Twitter Bootstrap. You created both list and tree components and saw how it's trivial to implement simple custom components that use both the Bootstrap CSS class specifications and Bootstrap's jQuery plugins.

In addition, you saw how it's possible, with a few lines of code, to customize a component to transition the user from one route to another in your application. I hope you have seen how powerful and helpful it is to define custom components, not as part of the overall context into which they're placed, but rather as pieces of functionality that can be reused in multiple places both across your application and across multiple applications.

In the next chapter, you'll take a look at how you can test your application to ensure that you're building the functionality that you intended, while also ensuring that future changes don't break existing functionality.

8

Ember.js – Testing your Ember.js Application

This chapter covers:

- Testing strategies available for JavaScript applications
- Using QUnit and PhanotmJS for unit and integration testing
- Combining these tools can be combined in order to build up a complete testing strategy
- An introduction to Integration Testing
- Using Ember Instrumentation for quick performance measurements

Even though JavaScript has matured significantly over the last five years, there are still a couple of areas where you will most definitely notice that you are working with a project that still has some maturing to do. Testing is definitely one of these areas, leaving a lot of the decisions required up to the application developers. This chapter will attempt to outline how you can successfully test your own Ember.js applications while looking at a real world implementation of one possible test harness.

As with applications written in other languages, there are multiple way to test your application, including:

- Unit Testing
- Integration Testing
- Performance Testing
- Regression Testing
- Black Box Testing
- Continuous Integration

Naturally, you might not need to implement a solution for all of these testing types in your own application. Chances are, though, that you might need several of them, even though unit testing and integration testing are the most common types of test harnesses you will find for JavaScript applications.

Depending on what other languages and tools you have previous experience with, chances are also that you will find your JavaScript test harnesses to be significantly more involved and specialized for your application and your environment. One of the reasons for this is that the tools available to perform JavaScript Testing are rather young, another is the fact that JavaScript has changed so rapidly from a scripting language to a full featured application framework, meaning that building standardized tools to perform testing in this environment is all that much more difficult.

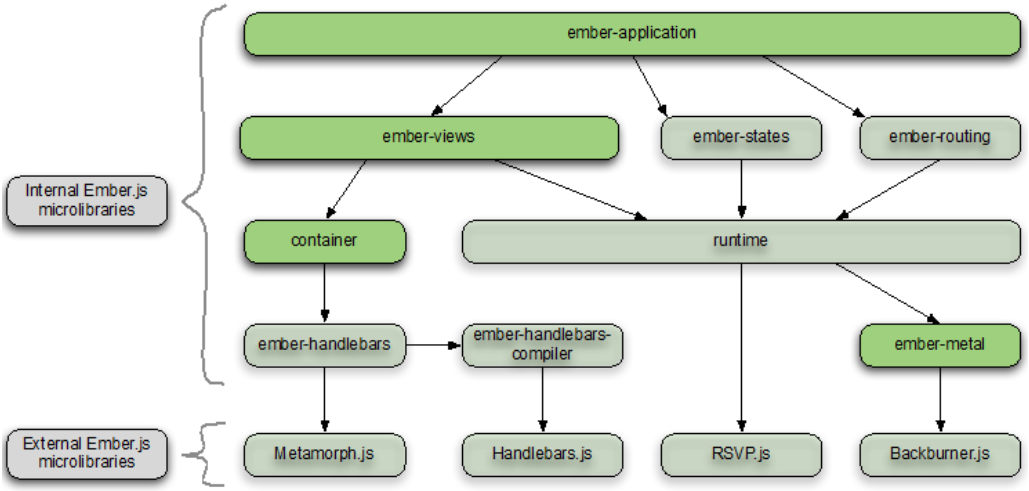


Figure 8.1 – The parts of Ember.js we will be working on in this chapter

That said, there is definitely a light at the end of the tunnel, and we are seeing an increased focus on good testing tools for JavaScript application emerging.

In this chapter we will build up a complete testing strategy. Throughout the chapter we will see examples taken from the Montric project. Specifically we will look at using QUnit and PhantomJS, and we will see one possible in which we can integrate these tools to perform both unit- and integration testing. We will learn how we can use PhantomJS to execute our tests in a headless mode, meaning that the both the test and the code will execute in an environment without requiring an actual browser. Finally we will see how we can use Ember Instrumentation in order to gain a quick insight into the performance of your Ember.js application.

8.1 Integrating QUnit and PhantomJS

Before we get started on writing some real test for the Montric application, I think we need to get a clear understanding of the tools that we will be using, as well as their purpose and how we are going to utilize them.

You can download QUnit from http://qunitjs.com and PhantomJS from http://phantomjs.org. Please note that PhantomJS needs to be installed onto your system before you can use it.

More specifically this chapter will look into the following tools:

- QUnit for Unit Testing
- PhantomJS for both Unit Testing and Integration Testing
- QUnit for Integration Testing

The first types of test that you will be likely to want to setup are unit tests, so we will start out by looking more closely at QUnit.

8.1.1 Writing Unit Tests with QUnit

QUnit is a framework that lets you write Unit test or your application. Because running tests that require an actual browser makes testing in a continuous integration environment particularily difficult, we will utilize PhanotmJS in order to execute the tests in a headless mode. Because PhantomJS allows us to run the tests without an actual browser, structuring your tests to work with PhantomJS will make testing in a continuous integration environment easier to set up.

QUnit itself is used, amongst others, by jQuery, jQuery UI and jQuery Mobile. In fact, the unit tests for the Ember.js framework are mostly written in QUnit. In order to get started you need to download both the QUnit JavaScript file, as well as the QUnit CSS file from QUnits website, http://qunitjs.com/.

As your application grows and the number of developers working on your project increase, having a logical, sane and reproducible set of Unit tests become vital for the continued development of your product. This chapter shows how QUnit and PhantomJS can be combined in order to both write and execute your projects unit tests.

Next, we will create a small unit test in order to verify that QUnit is in fact setup and is working properly. Create a file named firstTest.html and place it in a new directory somewhere on your harddrive. The contents of firstTest.html is shown below in listing 8.1.

Listing 8.1 – Our Very First QUnit Test

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">

<html lang="en">
<head>
     <title>First Test</title>
     <link rel="stylesheet" href="qunit-1.11.0.css" type="text/css"
charset="utf-8">                                                    #A
     <script src="qunit-1.11.0.js" type="text/javascript" charset="utf-
```

```
8"></script>                                                          #B

</head>
<body bgcolor="#ffffff">
     <div id="qunit" style="z-index: 100;"></div>          #C

     <script type="text/javascript" charset="utf-8"
src="firstTest.js"></script>                               #D
</body>
</html>
```

#A: Include the QUnit CSS file in the header of your HTML page
#B: Include the QUnit JavaScript file in the header of your HTML page
#C: Create a div with id qunit inside your body-tag
#D: Include the firstTest.js file which contains the test we are going to run

As you can see QUnit uses a HTML page to set up the test. Inside this page we need to include both the QUnit CSS and JavaScript files along with a div element that QUnit will use to display the test results. All of the tests that you want to run via the same HTML file, needs to be specified inside the body-tag of your HTML page. Note that this HTML expects to be able to find both the QUnit CSS and JavaScript files in the same directory. If this is not the case, please change this HTML page accordingly.

Next, we need to provide an implementation for the firstTest.js file. Here we are simply creating a new test that tests that the integer value of 1 is equal to the string value of "1". The full test script is shown below in listing 8.2.

Listing 8.2 – Creating a Simple Unit Test

```
test("Test that QUnit is working as expected", function() {          #A
    ok( 1 == "1", "QUnit Test Passed!" );                            #B
});
```

#A: Provide a name and a callback function to test via QUnits test-function
#B: Add an assertion to the callback function, which will act as the actual test

As you can see, each test is specified within QUnits `test()` function. The first argument to the `test()` function is the name of your test, while the second argument is a callback function that includes this tests assertions. QUnit will use the name of your test when it reports back the result of executing your test.

Any assertions that the test requires are created inside the callback function, passed in as parameter 2 to the `test()` function. In this case we are testing if the integer value of 1 is the same as the string value of "1", which in the land of JavaScript it strangely is.

In order to execute this test you can simply drag-an-drop the `firstTest.html` file into the browser of your choice. This will lead to a similar result as shown below in figure 8.2.

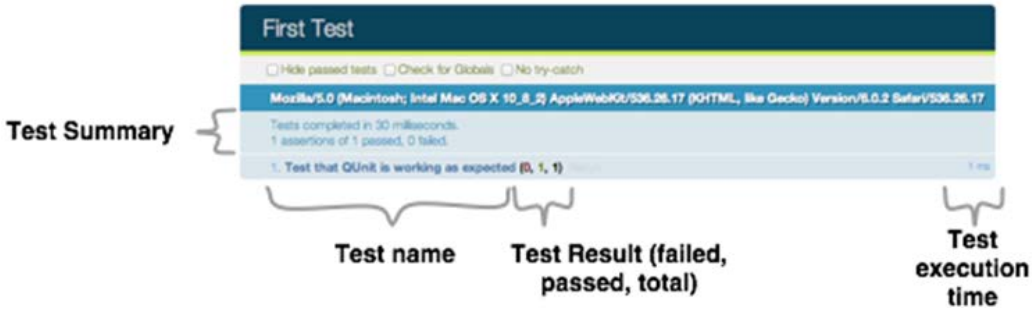


Figure 8.2 – Executing the firstTest.html file

As you can see from the figure above the test results consists of a few different elements

- A header displaying the name of the test. This is the same as the title-tag of your HTML document
- Options that lets you hide passes tests (only showing the failed tests), check for globals or disable the try-catch feature of QUnit
- The current browsers user-agent string
- The time it took to execute all the tests
- The total, passed and failed number of assertions
- The name of each executed test, as well as the total, passed and failed number of assertions for each test.

This is a convenient way to get up-and-running quickly with unit testing, but there are a number of key features missing from this setup

- Relying on a browser to execute your tests are cumbersome when testing. This is especially true when it comes to setting up and operating a Continuous Integration (CI) environment for your application
- There is no built-in way to execute this test from the command line
- Feedback is given via updates to the browsers DOM, which is hard to extract automatically

The points above all mean that this approach is not well suited for environments that require Continuous Integration to work seamlessly whenever there is a commit to the Source Code Management system. In order to be able to fix these issues we are going to bring in an additional tool, PhantomJS.

8.1.2 Executing from the Command line with PhantomJS

PhantomJS is a headless WebKit installation that is highly scriptable via a JavaScript API. PhantomJS works well via the command line, which means that it will also work equally as well executed through a CI-server. The fact that it serves as a headless WebKit installation opens

up many interesting approaches when it comes to testing. Using PhantomJS you can achieve the following:

- Execute your tests from the command line and get feedback on test results
- Execute your tests against a real WebKit installation
- Capture screenshots of your web application before, during or after a test case execution
- Chain together multiple test scripts and scenarios using third party tools
- Build up a testing pipeline that can be reused for different testing purposes (unit-, integration- and performance testing)

PhantomJS is widely deployed within the test strategies of companies around the world. The Ember.js project itself uses it, as well as Bootstrap, Modernizr and CodeMirror.

To get started, we are going to create a test that navigates to http://emberjs.com, verifies that the page's title is as we expect. If so, we will take a screenshot of the webpage and exit the test.

Listing 8.3 – Creating a Screen Capture PhantomJS Test

```
var page = require('webpage').create();                              #A
var before = Date.now();
page.open('http://emberjs.com/', function () {                       #B
    var title = page.evaluate(function () {                          #C
        return document.title;
    });
    if (title === "Ember.js - About") {                              #D
        console.log('Title as expected. Rendering screenshot!');
        page.render('emberjs.png');                                  #E
    } else {
        console.log("Title not as expected!")
    }

    console.log("Test took: " + (Date.now() - before) + " ms.");     #F
    phantom.exit();                                                  #G
});
```

#A: Require the webpage module from PhantomJS in order to load a web page
#B: Open up http://emberjs.com
#C: Fetch the page's title via the page.evaluate function
#D: Verify that the title is what we expect it to be
#E: Render a PNG file with the screenshot to the current directory, named emberjs.png
#F: Log the amount of milliseconds that the test took to execute
#G: Exit PhantomJS when the test is completed

There are a couple of important things to note from the above code listing. The first is the fact that we need to require the modules that we intend to use from PhantomJS before using them. There are a number of modules available, including

- `webpage` – Makes it possible for your test to interact with a single webpage.
- `system` – Exposes system-level functionality to your test

- `fs` – Exposes file system functionality, as well as access to files and directories to your tests
- `webserver` – An Experimental module that uses an embedded webserver that your PhantomJS scripts can start.

Once we have created an instance of the webpage module, we are able to load the Ember.js website in order to execute our test. The test starts out by fetching the webpage's title via the `page.evaluate()` function. If the title is what we expect it to be, `"Ember.js – About"`, the script takes a screenshot of the complete website and stores it in the current directory as emberjs.png. Before the scripts finishes it prints out the amount of milliseconds that the test took, making sure to end the script with the `phantom.exit()` function.

Save the file as testEmberHomepage.js. In order to execute the test you need to install PhantomJS onto your computer. There are binaries available for the most common operating systems, including Windows, Mac OS X and Linux. Once PhantomJS is installed, you can execute the script by issuing the command `phantomjs testEmberHomepage.js`. Figure 8.3 below shows the result from running the test on my laptop, with MacOS X installed.

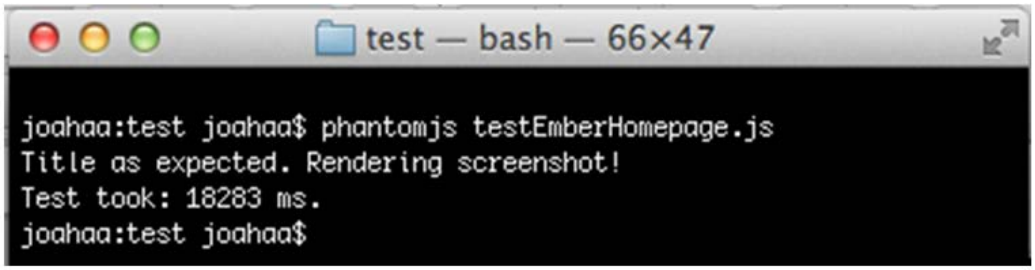


Figure 8.3 – Executing the testEmberHomepage.js script

After running the above test, you should be able to locate a file named `emberjs.png` alongside the `testEmberHomepage.js` file.

Now that we have made sure that we are able to execute tests via PhantomJS we can move on and integrate QUnit in order to run unit tests.

8.2 Combining QUnit and PhantomJS

There are two major issues with using QUnit by itself in order to write unit tests that can be executed by your CI server on each build of your application. First, QUnit requires its own HTML file in which you need to set everything up that the test needs. Second, QUnit will print its output straight into the DOM, making it hard for a CI server to figure out if any tests failed along the way.

By combining QUnit with PhantomJS, we can alleviate both of these issues, but in order to do so, we need to build a test harness where we can execute our test both while writing our application as well as via the CI server build.

Here, we will start out by showing one possible solution for integrating PhantomJS and QUnit, before moving on to writing actual unit tests in QUnit.

8.2.1 Running QUnit From PhantomJS

To be able to run QUnit from within PhanotmJS, we are going to adapt the PhantomJS QUnit integration script from the Ember.js project. Because the run-qunit.js script is rather lengthy, I have broken it down into parts. I am also only showing the most relevant parts. You can view the complete `run-qunit.js` script in Montrics source code on GitHub.

The first thing we will be looking at is the bootstrapping of the run-qunit.js script, where we declare what input parameters the script is expecting, as well as loading our test HTML file. This file serves as the integration point between PhantomJS and QUnit and in essence is what will allow us to execute the unit tests in a headless manner. The code is shown below in listing 8.4.

Listing 8.4 – Integrating PhantomJS and QUnit - Bootstrapping

```
var interval = null;                                                    #A
var start = null;                                                       #B
var args = phantom.args;                                                #C
if (args.length != 1) {
    console.log("Usage: " + phantom.scriptName + " <URL>");
    phantom.exit(1);                                                    #D
}

var page = require('webpage').create();
page.open(args[0], function(status) {                                   #E
    if (status !== 'success') {                                         #F
        console.error("Unable to access network");
        phantom.exit(1);
    } else {
        page.evaluate(logQUnit);                                        #G
        start = Date.now();                                             #H
        interval = setInterval(qunitTimeout, 500);                      #I
    }
});
```

#A: The interval parameter will be used later in the script to stop PhantomJS when all tests pass
#B: The start parameter is used to exit PhantomJS if the unit tests takes too long to execute
#C: Fetching the arguments to PhantomJS via the phantom.args function
#D: If the number of arguments is not 1, print out the correct usage and exit
#E: Open the webpage given as the first argument to the the script
#F: If PhantomJS cannot open the webpage requested, exit the script
#G: If PhantomJS can open the webpage, call the logQUnit function
#H: Record the current timestamp in the start variable
#I: Set an interval to be able to exit PhantomJS later, and store it in the interval variable

We start out by defining a variable called `interval` into which we will store a JavaScript `interval` object. We will be using this interval to exit PhantomJS once every test has completed successfully or tests times out. The script then fetches the arguments that are given into the scripts and prints out an error message if the number of arguments is not equal to

one. In the next section, we are simply going to open the URL that run-qunit.js is passed in via its first argument. If PhantomJS is unable to load this URL, we will print out an error message and exit PhantomJS, otherwise we are going to call the function `logQUnit()` via the `page.evaluate()` function. The last thing this part of the script does is to register an interval that will execute the `qunitTimeout()` function every 500 milliseconds. As we will see later we are using this interval to exit PhantomJS once every test has completed or if any of the tests times out.

Before we delve into the execution of QUnit itself, lets take a closer look at the `qunitTimeout()` function which is shown below in listing 8.5.

Listing 8.5 - Integrating PhantomJS and QUnit – qunitTimeout()

```
function qunitTimeout() {
    var timeout = 60000;                                          #A
    if (Date.now() > start + timeout) {                           #B
        console.error("Tests timed out");
        phantom.exit(124);
    } else {
        var qunitDone = page.evaluate(function() {
            return window.qunitDone;                              #C
        });

        if (qunitDone) {
            clearInterval(interval);                              #D
            if (qunitDone.failed > 0) {
                phantom.exit(1);                                  #E
            } else {
                phantom.exit();                                   #F
            }
        }
    }
}
```

#A: All test must finish within 60 seconds
#B: If more than 60 seconds have passed, halt test execution and exit PhantomJS
#C: Check to see if QUnit is done
#D: If QUnit is done, clear the interval timer
#E: If any tests failed, exit PhantomJS with an error state
#F: If no tests failes, exit PhantomJS normally

The `qunitTimeout()` function is executed every 500ms via the JavaScript `setInterval()` function declared in listing 8.4 above. The script starts out by defining the maximum number of milliseconds that all of the unit tests are allowed to take, combined. It then periodically checks to see if this threshold has been breached and exits PhantomJS immediately if it has. This mechanism is added to ensure that test execution wont hang or otherwise bring down a continuous integration system.

Within the allowed execution time, the script checks to see if QUnit is done via `window.qunitDone`. If QUnit have finished, the script clears out the interval that executes

the timeout checking. If QUnit reports that any of the tests have failed, the script will exit PhantomJS with an error state. Otherwise it will exist PhantomJS normally.

So far so good. Next we will be looking at the `logQUnit()` function. This function will register a number of callbacks into the QUnit runtime in order to gather the data it needs to be able to print out the results of executing the tests. The most important metric for any test harness is the number of actual tests that are left in either of the passed, failed or not-executed state.

Here, we will only look at a skeleton of the `logQUnit()` function, please have a look at the run-runit.js script from Montric for the full contents.

Listing 8.6 - Integrating PhantomJS and QUnit – logQUnit()

```
function logQUnit() {
    var moduleErrors = [];                                          #A
    var testErrors = [];                                            #A
    var assertionErrors = [];                                       #A

    QUnit.moduleDone(function(context) {                            #B
        //Log any failures to the moduleErrors Array
        //Print Module status to the console
        …
    });

    QUnit.testDone(function(context) {                              #C
        //Log any failures to the testError and
        //asertionErrors Arrays
        …
    });

    QUnit.log(function(context) {                                   #D
        //Print Assertion error messages to the console
        …
    });

    QUnit.done(function(context) {                                  #E
        //Print out any moduleErrors and testErrors
        //Print Stats that show the total, successful and failed
        //tests
        …
        window.qunitDone = context;                                 #F
    });
}
```

#A: Creating arrays to hold the error messages from the modules, tests and assertions
#B: This callback will be executed each time a module finishes. We can then log any module failures into the moduleErrors array
#C: This callback will be executed each time a test finishes. We can then log any test failures to the testErrorsArray
#D: This callback will be executed each time QUnit wants to log output, typically each time an assert fails.
#E: When QUnit is done executing all of the tests, it will execute this callback. We can now log the final stats about our tests. We finish by setting the window.qunitDone to true.

The `runQUnit()` function registers four callbacks into QUnit in order to be notified when QUnit executes the tests.

The script book keeps the test results for tests contained within a module, for assertions within a single test, as well as for single assertions. The script will report back these results after each module, as well as a summary after all of the tests have executed.

8.2.2 Writing A Simple Ember.js Unit tests with QUnit

Writing unit tests for JavaScript applications is different than what you might be used to with other, statically typed languages. Because there is no standard way to set up a JavaScript application runtime you need to provide a complete application in your QUnit setup. QUnit is setup via an html-file, as we saw earlier in this chapter. Inside this HTML file you need to bootstrap your entire application, remembering to include any third-party JavaScript libraries that your application is using. Once that is setup, we can start writing unit tests for our application.

THE FUNCTION-UNDER-TEST

Montric.ApplicationController is used, among other things, to generate readable strings, generated from JavaScript date objects, which will be displayed on a live-updating chart in the application. In order to format these strings it is possible to provide a dateString that will tell Montric.ApplicationController how to correctly format the date-string.

When a user of the Montric application requests a chart, the client side will issue an XHR request to the server in order to fetch the data for the chart. Montric draws the chart with the values along the y-axis and the time along the x-axis. Depending on the user's preferences the format of the dates along the x-axis will be formatted accordingly. Figure 8.4 below shows one possible way to format the chart.

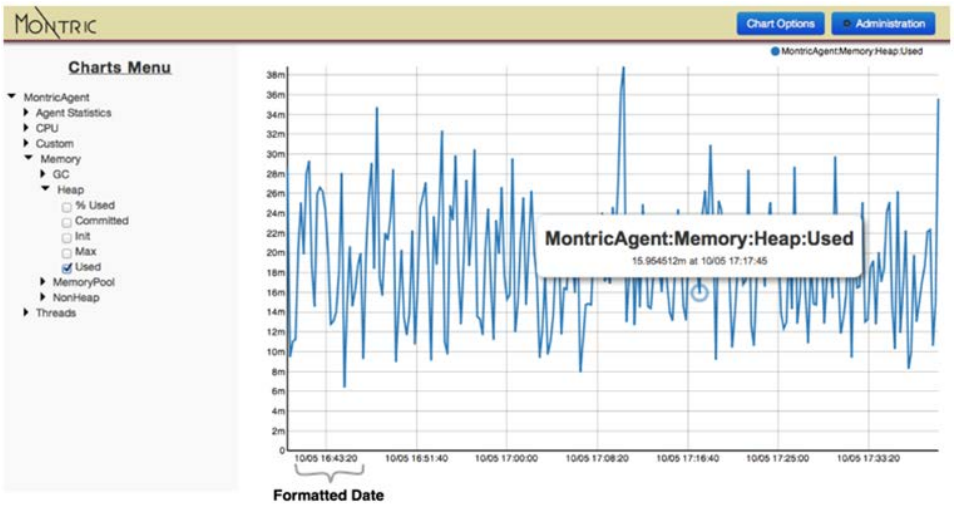


Figure 8.4 – The result of specifying the date format as dd/mm hh24:MM.ss

The code that converts the JavaScript date into this human readable date is shown below in listing 8.7.

Listing 8.7 – The Montric.ApplicationController generateChartString() function

```
Montric.ApplicationController = Ember.Controller.extend({
    dateFormat: 'dd mmmm yyyy HH:MM',                          #A

    generateChartString: function (date) {                     #B
        var fmt = this.get('dateFormat') || 'dd.mm.yy';        #C

        var dateString = date ? dateFormat(date, fmt) : "";    #D
        return dateString;
    }
});
```

#A: Specifying a default date format string
#B: The function we are going to write a unit test for
#C: Getting the date format. Defaults to dd.mm.yy if no dateFormat is specified
#D: Generating a date string if a date-input was given, an empty string otherwise

Here, the ApplicationController specifies a default `dateFormat` that it will use to format dates into human readable form for displaying on a chart. In order to provide consistency of the dates displayed on each chart throughout the application, the `dateFormat` controls the dates that are presented to the user throughout the application.

Our unit test will be testing the functionality of the `generateChartString()` function. This function takes a single `date`-argument, which it will use to return a valid string representation from. If `dateFormat` is not defined or is null at this point in time, the default format "`dd.mm.yy`" date format is used.

If the date variable is null or undefined, we are simply returning an empty string, otherwise we are going to return the result of the `dateFormat()` function.

Bootstrapping the Unit Test

Now that we know what the function we are testing does, lets get started with writing a unit test using QUnit. Listing 8.8 shows the test-setup, while listing 8.9 shows the actual unit tests.

Listing 8.8 – Testing the generateChartString() function

```
var appController;                                             #A
var inputDate = new Date(2013,2,27,11,15,00);                  #B

module("Montric.AppController", {                              #C
    setup: function() {                                        #D
        Ember.run(function() {
            appController =
Montric.__container__.lookup("controller:application");        #E

        });
    },

    teardown: function() {                                     #F
```

```
    }
});
```

#A: Creating a variable to hold the Ember.js-instantiated ApplicationController
#B: Creating a variable to hold the date we will base the unit tests around
#C: Creating a QUnit module to house similar unit tests
#D: We are going to use the modules setup() function to fetch the ApplicationController
#E: Fetching the Ember.js-instantiated ApplicationController and storing it in appController
#F: The module also have a teardown() function, included here for completeness

We start out by defining two variables, one to hold the Ember.js instantiated `Montric.ApplicationContoller`, and one to hold a date object that we will be using throughout our unit tests. The `module()` function allows us to perform some common setup that all tests will require, as well as any teardown functionality that our tests need. As we will see later, Ember.js provides a nice and simple way to reset our application between each test, meaning that you will most likely not have any special teardown functionality for your Ember.js-based unit tests.

Inside the `module()` function I have included both a `setup()` and a `teardown()` function. Inside the `setup()` function we are simply fetching the `Montric.ApplicationController` and assigning it to the `appController` variable. Note that it is OK to use the `Montric.__container__.lookup()` function for our unit tests, but you should never, and I mean **never**, use this private function in your production code! Ember.js does a lot of work trying its best to enforce you to develop your application with a clear MVC-structure. If you find yourself wanting to use the __container__ within your own application, you should rather go back and find out where your application logic is combining concerns that ought to be separated.

CREATING THE UNIT TESTS

Next, we are going to create five unit tests that test the generateChartString() function in order to verify that:

- We were able to get hold of the Montric.ApplicationController instance
- That the default date-format formats the date according to the default specification given in the class initialization
- That providing a custom date-format formats the date according to the date-fomat pattern given
- That a null date-format formatted the date accorind to “dd.mm.yyyy”
- That a null date returns an empty string

The code for all five tests is shown below in listing 8.9.

Listing 8.9 – Creating Five Unit Tests to Test the generateChartString() function

```
test("Verify appController", function() {                    #A
    Montric.reset();                                         #B
    ok(appController, "Expecting non-null appController");   #C
});
```

```
test("Testing the default dateFormat", function() {                    #D
    Montric.reset();                                                   #B
    strictEqual("27 March 2013 11:15",
appController.generateChartString(inputDate), "Default Chart String
Generation OK");                                                       #D
});

test("Testing custom dateFormat", function() {                         #E
    Montric.reset();                                                   #B
    appController.set('dateFormat', 'dd.mm.yyyy');
    strictEqual("27.03.2013", appController.generateChartString(inputDate),
"Custom Chart String Generation OK");                                  #E
});

test("Testing null dateFormat", function() {                           #F
    Montric.reset();                                                   #B
    appController.set('dateFormat', null);
    strictEqual("27.03.13", appController.generateChartString(inputDate),
"Null Chart String Generation OK");
     #F
});

test("Testing null date", function() {                                 #G
    Montric.reset();                                                   #B
    strictEqual("", appController.generateChartString(null), "Null Date OK");
     #G
});
```

#A: Writing a test to verify that appController is not null nor undefined
#B: Making sure to reset the Montric application before each test
#C: Asserting that the appController is OK via the ok-assert
#D: Writing a test to verify the default date format
#E: Writing a test to verify that a custom date format works as expected
#F: Writing a test to verify that a null date format generates the expected date string
#G: Writing a test to verify that a null date generates an empty string.

As you can see, each of our tests starts out by calling `Montric.reset()`. This will ensure that Ember.js will reset our application to a freshly loaded copy. This makes unit testing a lot easier, because we do not have to keep track of what each test might have changed in order to successfully write the code for the modules `teardown()` function.

There is nothing-special going on in any of the five tests, and the code should be readable and easy to understand. In the first test we are using the `ok`-assert, which will pass if the first argument is *truthy*, meaning that it is neither null, undefined nor false. Here, we are testing to see if we were able to retrieve a valid non-null instance of Montric.ApplicationController. This test is an important addition to your test module, because it will make it a lot easier to debug the cause of your tests failing if you know that it is caused by not being able to retrieve the controller upon which the rest of the tests are built.

For the rest of the tests, the goal is to assert that the `generateChartStrings()` function returns a correct string representation given four different circumstances. For instance, in the second test we are verifying that the controllers default date-format works as intended. In this

case we use the `strictEquals`-assert in order to check if the formatted date equals to "27 March 2013 11:15".

strictEquals() vs. equals()

The stricktEquals() function verifies that the first two arguments are equal using the strict equality operator (===). We could have used the `equals`-assert to check for non-strict equality, which would have used the equality operator (==)instead).

The last argument that we pass in to each assert is a String that will be displayed in case of the unit test failing.

Figure 8.5 shows the result of running the unit tests.

```
joahaa:qunit joahaa$ phantomjs run-qunit.js http://localhost:8081/index-test.html
Module EurekaJ.AppController Finished. Failed: 0, Passed: 5, Total: 5


Time: 174ms, Total: 5, Passed: 5, Failed: 0
```

Figure 8.5 – Executing the unit tests from PhantomJS

As you can see, we are simply executing the command `phantomjs run-qunit.js http://localhost:8081/index-test.html` in order to execute the unit tests. The first argument to PhantomJS is the script that it will execute. The rest of the arguments given will be used as input-arguments to this test-script itself. In our case, we are giving the run-qunit.js script the URL that we wish to use for executing the QUnit scripts. The run-qunit.js script will load the URL that it gets passed into it and set up the listeners and hooks into QUnit that it needs in order to be able to report back the progress of the unit tests as well as to be able to report back any test failures that occur while executing the tests.

Now that we have seen how we can use QUnit in order to test single-functions and single-classes, the next section will explain how we can leverage the same technologies in order to implement integration testing as well.

8.3 Integration Testing

While Unit testing are concerned with testing single units of work, broken down to single functions and single classes, integration testing is concerned with integrating different layers of your application and performing wider tests on your application. Depending on your requirements and your test setup you will be able to isolate particular parts of your application in order to test their isolated requirements, or you will be able to test certain features all the way through your application.

There are many tools available that lets you do integration testing at different layers within your JavaScript Application. Some of the more popular choices include Mocha, Capybara, Selenium-Webdriver and Casper.js. For our purpose, since we are already using QUnit for our Unit tests, we will stay with QUnit and PhantomJS, and rather bring in a third library, sinon.js, which can be downloaded from http://sinonjs.org.

Sinon.js is a library that will help us to fake the parts of the application that you do not want to test for your particular test. There are a number of ways to achive this, based on the intended result, including:

- Stubs – Objects that provide a valid, but static result. Regardless of what you pass into the stubs you will always get the same response
- Mocks – Objects that also provide a valid result, but these also provide you with input on which methods that were executed and their invocation count
- Spies – Objects that allow report back the same information as mocks. While mocks also provide pre-programmed functionality, spies only lets you implement fake methods without functionality.
- Fake Objects – Objects that act like the real object, but in a simpler way. A common example is a data-access-object that stores its data in memory instead of in a real database

Sinon.js provides all of the above features, which we need in order to implement a maintainable integration testing strategy.

8.3.1 Introduction to Sinon.js

In this chapter we will focus our integration testing on the client-side application. Because we are definitely not interested in the communication with the server side while performing these types of integration tests, we will introduce a mocking framework, sinon.js, which will allow us to effectively mock out the server-side communications, letting each test specify a fake server-side response. The first thing we need to do is to download sinon.js, sinon-server.js and sinon-qunit.js and include them in an index-integration-test.html file.

Sinon.js is a library that provides testing spies, stubs and mocks for any of the JavaScript test frameworks available. We will be using Sinon.js mainly to mock out the server-side communication by letting each test specify a fake return value in place of the real server-side request and response.

For this example we will be writing an integration test that tests the Alert Administration feature of Montric. In short, Montric lets its users set up custom alert thresholds for the gathered metrics. The GUI provides the user with the ability to add, alter and delete alerts from the system. Figure 8.6 below shows the point in the application that we will be testing.

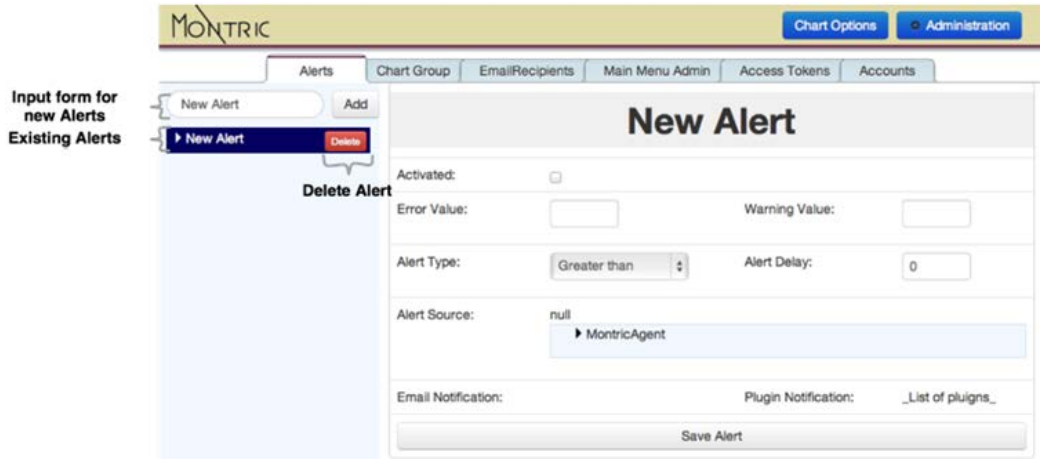


Figure 8.6 - The Alert Administration View

As you can see from the screenshot in figure 8.4 the user will be able to input a name for the new alert before clicking on the Add-button to create the alert. Once there are alerts registered in the system, the user is then able to select the alert by clicking on it. The selected alert is editable via the form to the right of the list of alerts, and can be deleted by clicking on the delete button.

We will create an integration test that will allow us to test that we are able to add new alerts to the system.

8.3.2 Integration Test for Adding a New Alert

We will get started by writing a test that will verify that we are able to add a new test to the system by filling out the name for a new test and clicking on the add-button. We are going to verify that this alert is added to the `content`-array of the `Montric.AdministartionAlertsController`. The first thing we need to do, however, is to enable the Sinon fake server in order to stub out any of the AJAX calls. Because I need to make sure that the fake server is setup before the Montric application is set up, I include the script shown in listing 8.10 below inside the index-integration-test.html file, just before I load up the Montric application.

Listing 8.10 – Initializing Sinon.js' fakeServer

```
<script type="text/javascript">
    Montric.server = sinon.fakeServer.create();                    #A
</script>
```

#A: Initializing the Sinon Fake Server inside index-integration-test.html

In the listing above, we are creating a new instance of `sinon.fakeServer`, which we are storing into the `Montric.server` property, in order for our tests to be able to retrieve the `fakeServer` instance.

Once we have ensured that the application won't issue any XHR request to the server, we can start implementing the actual integration test. Listing 8.11, below, shows the module definition of the test.

Listing 8.11 – The alertAdminIntegrationTest.js Module Setup

```
var alertAdminController;

module("Montric.AdministrationAlertsController", {
    setup: function() {
        console.log('Admin Alerts Controller Module setup');
        Montric.server.autoRespond = true;                              #A
        Montric.server.respondWith("GET", "/alert_models",              #B
                [200, { "Content-Type": "text/json" },
                 '{"alert_models":[]}'
                 ]);


        Montric.server.respondWith("POST", "/alert_models",             #C
                [200, { "Content-Type": "text/json" },
                 '{"alert_model":{"alert_source":"null","id":"New
Alert","alert_delay":0,"alert_plugin_ids":[],"alert_notifications":"","alert_
activated":false,"alert_type":"greater_than"}}'
                 ]);

        Ember.run(function() {
            alertAdminController =
Montric.__container__.lookup("controller:administrationAlerts");        #D

        });
    },

    teardown: function() {
    }
});
```

#A: Tell Sinon.js' fakeServer to autorespond to AJAX requests
#B: Provide Sinon.js' fakeServer with the intended JSON response to the first AJAX request
#C: Provide Sinon.js' fakeServer with the intended JSON response to the second AJAX request
#D: Getting hold of the instantiated Montric.AdministrationAlertsController

We start out configuring the fake server to automatically respond to AJAX requests, before we create the two AJAX requests that Montric will issue. The first response will be issued when the application is loaded and Montric attempts to load any alerts that might already be stored for that user account. In this case we are returning an empty array, because this test doesn't require the system to have any pre-defined alerts. The second response is a fake response that simulates what the server would reply with when a new alert with the name "new alert" is created and sent to the server.

In the final part of the module setup, we are fetching the application instantiated `Montric.AdministrationAlertsController`, which we will store in the `alertAdminController` variable, so that it can be used from within our tests.

Next, we will add a test to verify that we are able to create a new alert and add it to the controllers content-array. The code for this test is shown below in listing 8.12.

Listing 8.12 – Testing Adding New Alerts

```
var testCallbacks = {                                                    #A
    verifyContentLength: function() {                                    #B
        Montric.reset();
        if (alertAdminController.get('content.length') > 0 ) {
            strictEqual(1, alertAdminController.get('content.length'),
                    "Expecting one alert. Got: " +
alertAdminController.get('content.length'));                             #C
            QUnit.start();                                               #D
        }
    }
};

asyncTest("Create a new Alert and verify that it is shown", 2,
        function() {                                                     #E
    ok(alertAdminController, "Exepcting a non-null
AdministrationAlertsController");                                        #F

    alertAdminController.get('content').addObserver('length', testCallbacks,
'verifyContentLength');                                          #G

    alertAdminController.set('newAlertName', 'New Alert');               #H
    alertAdminController.createNewAlert();                               #I
});
```

#A: Creating a container that we can put the tests callbacks on via Ember.js' addObserver
#B: The verifyContentLength callback will be used to verify that the number of alerts have increased, and to end the test
#C: Asserting that an alert was added
#D: Restarting the test after the asynchronous call
#E: Because we are running a test with asynchronous calls, we are using QUnits asyncTest
#F: Asserting that we have a valid instance of the controller
#G: Adding an observer inside which we can check to see the result of the async-part of the test
#H: Specifying the name of the new alert
#I: Simulating clicking the Add-button

Because we are testing asynchronous code here, we need to use QUnits `asyncTest` function. This function will execute everything inside the test function, but it wont exit the test until the `start()` function is called. This is handy in order to be able to test Ember.js applications, which is, by nature, very asynchronous.

The test starts out, as always, by resetting the application before it verifies that it was able to retrieve a valid controller from the Ember.js `Container`. Next, we are updating the alert name input box with the text `"New Alert"`, before triggering the controllers `createNewAlert()` function. This is a simple way of simulating a user clicking on the Add

button, which will call the same `createNewAlert()` function. This function will create a new `Montric.AlertModel` object with the `id` given in the new alert input box.

The next part is a bit tricky to test. We want to assert that the alert we created does indeed exist in the list of alerts after the server-side have responded. In order to do so, we must add an observer on the `alertAdminController's content.length` property. When the `content.length` property changes, the test will invoke the `verifyContentLength` callback.

Once the `verifyContentLength` callback is invoked, we are able to assert that there is exactly one item in the content array, and that its id is the one we provided it with, `"New Alert"`. Once we have asserted that everything is as we expect, we can safely call the QUnits `start()` function in order to progress to the next test.

The above test demonstrates one way in which we can integrate QUnit, PhanotomJS and Sinon.js in order to create an integration test harness for your Ember.js based applications. Most of your integration tests will have a similar setup, depending on how much traffic there is between the client and the server side.

Before concluding this chapter, we are going to take a quick look at how we can performance test Ember.js applications via the built-in `Ember.Instrumentation` implementation.

8.4 Performance Testing with Ember.Instrumentation

There are multiple ways to performance test your Ember.js applications. One rather simple way is to use Ember.js' built in instrumentation API. In order to use this API you simply register which event you would like to get measurements for, and implement a before and after function. In our case, we want to retireve measurements of the time it takes to render the views for the Montric application.

Listing 8.12 below shows how you can retrieve these performance metrics by subscribing to the `render.view` event.

Listing 8.12 – Measuring render.view

```
Ember.subscribe('render.view', {                                    #A
    ts: null,                                                       #B
    before: function(name, timestamp, payload) {
        ts = timestamp;
    },
    after: function(name, timestamp, payload, beforeRet) {          #C
        console.log('instrument: ' + name +
                " " + JSON.stringify(payload) +
                " took:" + (timestamp - ts));
    }
});
```

#A: Subscribing to the render.view event
#B: Variable to recording the timestamp just before the views render
#C: Generating a log-message showing the time each view takes to render

As you can see, the above code is rather simple, starting out by subscribing to the `render.view` event. An event will get triggered just before the view render, as well as just after the view has finished rendering. In this example we are simply keeping track of the timestamp in the `before()` function, and using this to print out a log message that shows how long the view took to render in the `after()` function. An example of the result is shown below in listing 8.13.

Listing 8.13 - The Instrumentation Result

```
instrument: render.view {"template":null,"object":"<LinkView:ember427>"}
took:6.5119999926537275

instrument: render.view
{"template":null,"object":"<Montric.BootstrapButton:ember434>"}
took:6.587000010767952

instrument: render.view
{"template":null,"object":"<Montric.HeaderView:ember424>"}
took:9.111000021221116

instrument: render.view
{"template":"adminAlertLeftMenu","object":"<Ember.View:ember471>"}
took:6.40800001565367
```

As you can see from the listing above, the output is fairly simple. In this example we are simply logging the results to the console, but because you do, in fact, have this information available inside your application you can implement interesting strategies detect possible errors and bottlenecks on the client-side. Although this book wont cover this, it should be fairly easy to write code that aggregates the information you need from your users, and periodically sending this metric back to you server-side for analysis.

There are quite a few events that you can subscribe to, including:

- render.view
- render.render.boundHandlebars
- render.render.metamorph
- render.render.container
- * - (all)

Even though the functionality offered by Ember.Instrumentation is limited at this point, it still provides a useful and quick way to gather statistics on the render process of your applications. There are other, more involved tools, available in order to perform performance testing, but depending on your needs and requirements Ember.Instrumentation might be just what you need.

8.5 Summary

There are multiple approaches available in order to test your JavaScript application. Because the underlying ideas and goals of many of these tools are the same, I have tried to keep the

number introduced libraries and frameworks to a minimum. While most of the examples in this chapter are based on QUnit, there are multiple testing libraries available that cater for different requirements and different test styles.

Because JavaScript as a platform has grown so rapidly over the last few years, the tools that are available often feel lacking, incomplete or poorly integrated. This makes implementing strategies for testing your applications at different layers more difficult than it should be, and more difficult than what programmers have become used to from other, more mature languages. That said, the JavaScript tooling business is blossoming these days, with new and powerful tools emerging at an increasing pace.

Hopefully this chapter has given you some ideas as to how you can utilize some of the more popular testing libraries and tools in order to implement the testing strategies that you require for your Ember.js applications. In particular, PhantomJS is quickly becoming the de-facto standard for preforming headless testing, which means that most of the other libraries and tools have implemented support for it. As time progress, it will become more clear which of the tools available that gains popularity, which means that the tools will integrate easier moving forward.

This chapter concludes part two of this book. At this point you should be able to design, write and test your Ember.js based applications, as well as having a thorough understanding of the Ember.js core concepts and features. The next part of the book will discuss more advanced topics, as well as packaging, deployment and interacting with the cloud.

10

The Ember.js Run Loop – Backburner.js

This chapter covers:

- The internal structure of the Ember.js Run Loop
- How Ember.js use to Run Loop to propagate events through your application
- How the Run Loop is great for your applications performance
- Executing code within a specific Run Loop
- Executing repeated tasks within the Run Loop

The Ember.js Run Loop is quite a unique concept to Ember.js and is one of the distinguishing features over similar frameworks like Angular or Backbone.js. Even though the name might indicate that the Run Loop is implemented as a continuous loop, its actually not. During the final release candidates to Ember.js, the Run Loop was extracted into its own sub-library called Backburner.js. Even though Ember.js now uses Backburner.js internally, the APIs internal to Ember.js remain the same as before. Figure 10.1 shows how Backburner.js fits in with the rest of the Ember.js application framework.

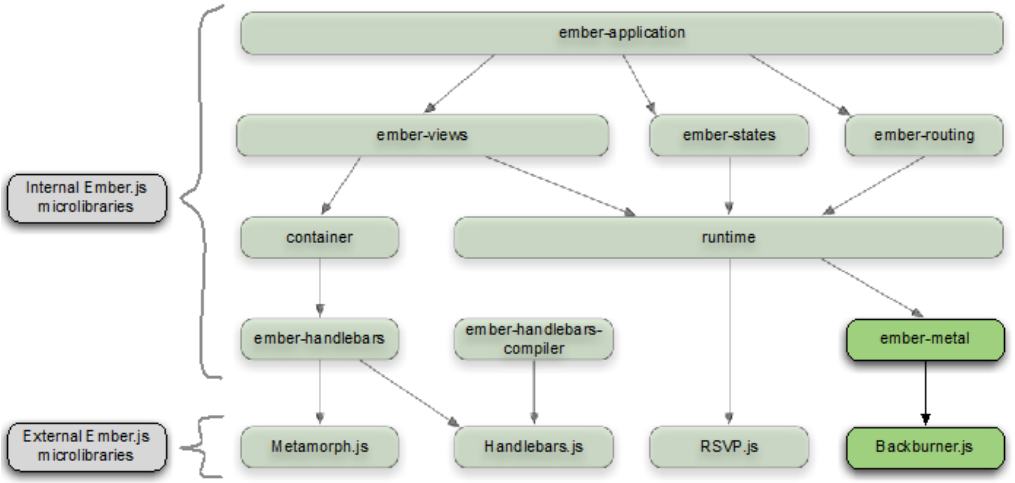


Figure 10.1 – How Backburner.js fits in with the rest of the Ember.js framework

This chapter explores what Backburner.js is and how Ember.js utilizes it in order to keep your Ember.js based application responsive, and your data bindings up-to-date throughout your application.

When I refer to the Ember.js Run Loop in this chapter I am referring to both the Ember.js specific API as well as the underlying Backburner.js library.

We will start by explaining what the run loop is, before we take a look at a sample application in order to take a deep dive into the run loop itself. Once that's out of the way we will take a closer look at how we can interact with the run loop and how we can issue code to execute within the constraints of a run loop.

10.1 What is the Run Loop?

In short, the Run Loop, is a mechanism that Ember.js uses in order to group, coordinate and execute events, key-value notifications and timers within your application. The run loop will remain dormant until a valid event occurs within your application or until you start one manually via the API.

In order to explain what the Run Loop is, and what role it plays within your application, lets try to define the run loop in the context of a simple application, the TodoMVC application. Specifically we will look at the Ember.js version of the TODO MVC example at http://todomvc.com. So before we get into how the Run Loop works, lets quickly introduce the application.

10.1.1 Introducing the Ember.js Todo MVC Application

The Todo MVC project is a great project where you will be able to explore and compare the different JavaScript MVC libraries and frameworks. Each of the MVC frameworks present implements the very same Todo application and you will be able to go in and look at the source code developed for each of the frameworks. Figure 10.2 below, shows how the Todo application looks.

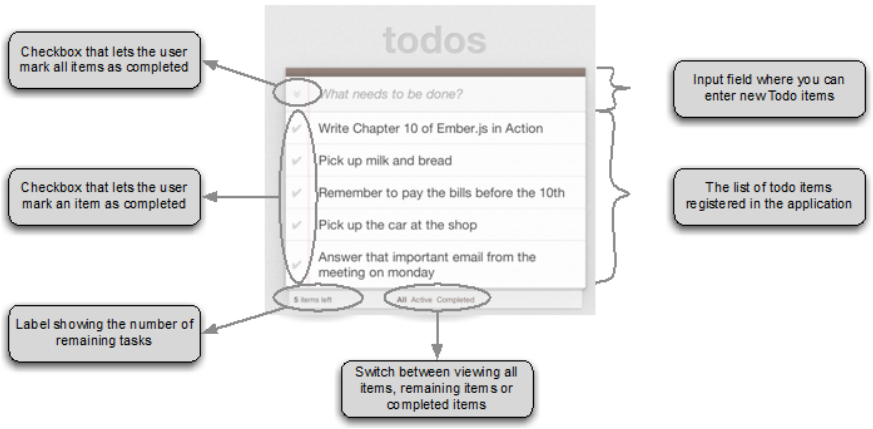


Figure 10.2 – The Todo MVC Application

As you can see, the Todo MVC application has a number of features. Using the text-field at the top of the application the user is able to type in and enter new items to the todo list. Once the user has added at least one item, the items will be displayed in a list below the text input field. To the left of each of the items in the todo list, there is a single check box that lets the user mark that item as completed, and to the left of the text input field there is a checkbox that will allow the user to mark all of the remaining tasks as completed.

Finally, at the bottom left of the Todo MVC application, there is a label that will show the number of tasks that have yet to be completed.

Consider the scenario where the user has added 100 items to his todo list. Later, the user has finished his chores and wish to mark all of the tasks as completed. What happens when the user clicks on the “mark all” checkbox next to the text input field? What you definitely do not want to happen, is for the application to update the DOM for each of the 100 tasks, as this is going to be a particularly time consuming, slow and expensive task to carry out. This is where Ember.js’ Run Loop kicks in and helps your application structure the events and carry them out in a sane and efficient order.

With this scenario in mind, lets take a closer look at the Ember.js Run Loop.

10.1.2 The Ember.js Run Loop Explained

Despite its name, the Run Loop isn't a loop that executes continuously, but rather it consists of a set of predefined queues, which are placed in an array defining the sequence of execution. Figure 10.3 below shows the default queues that are defined in Ember.js' Run Loop.

Figure 10.3 –The Ember Run Loop Queues

By default, Ember.js includes five queues. Unless otherwise specified any events that are added to the run loop will become scheduled into the **action** queue. We will discuss these queues, their relationship as well as their responsibilities as we progress through the rest of the chapter.

Ember.js does a good job at implementing listeners that will fill in these queues whenever appropriate events occur. In rare cases where Ember.js won't add an event to one of the run loop queues, Ember.js provides an API that lets you schedule your own events into one of the queues. If you happen to do something sufficiently unique or complex, like implementing your own custom listeners, working with new technologies like WebRTC or WebSockets, you are even allowed to create new queues.

In the Ember.js TodoMVC application, when the user clicks on the "mark all tasks as done" checkbox, the event will start up a new Run Loop, starting its work on the `Sync` queue, which contains all of your applications actions that involve propagating bound data. In this queue you will, at this point find at least 400 events in the `Sync` queue:

- There are a total of 100 events that will update each of the tasks in the todo list, marking the todo as done by setting the `isCompleted` property to true.
- There are 100 checkboxes, one next to each todo item. These checkboxes have their checked-property bound to the todo item's `isCompleted` property. These binding events account for another 100 entries into the Sync queue.
- At the bottom left of the application there is a label indicating the number of remaining todo items. This label is bound to the `TodosController's remainingFormatted` computed property. This property is, in turn, bound to each of the todo items `isCompleted` properties, accounting for another 100 events in the Sync queue.
- Similarly the "mark all as done" checkbox at the top left of the application will toggle on or off depending on whether or not all of the items in the todo list are marked as completed or not. These events account for another 100 events in the Sync queue.

Once the Run Loop has finished working through this queue, all of your bindings will have been propagated out through your application.

Depending on how your application's bindings are set up, the effect of carrying out one binding might lead to new events being added to one of the Run Loop Queues. The Ember.js

Run Loop will therefore ensure that the current queue, as well as any previous queues are completely exhausted before moving on to the next queue. Figure 10.4 shows the state of our Run Loop just after it has been triggered, just before it has started working on exhausting the sync queue.

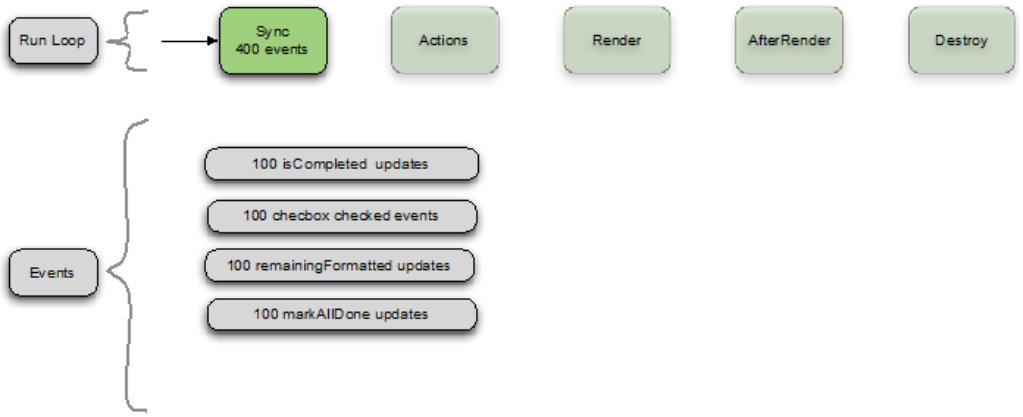


Figure 10.4 – The start of the Ember.js Run Loop

Once the `Sync` queue is empty the Run Loop moves on to the `Action` queue. This queue is the default queue in Ember.js and contains any events that needs to be carried out after all of the bindings have been propagated, but before the views get rendered into the DOM. The only two events that are specifically specified to go into the `Action` queue are the application initialization and RSVP events (RSVP is the library Ember.js uses internally to interact with promises).

During the processing of the events in the Sync queue have completed, Ember.js will have propagated out the bindings, and will have added events to the Render queue, telling Ember.js to redraw all 101 checkboxes and to cross out the completed tasks with a strikethrough. In addition, the counter at the bottom left of the application will have to be updated in order to show that there are zero todo items remaining. This will account for another event in the Render queue. Once the Sync queue is empty, the Run Loop will move on to the next queue, the Actions queue. Figure 10.5 shows the state of the Run Loop as it starts working on the Actions queue.

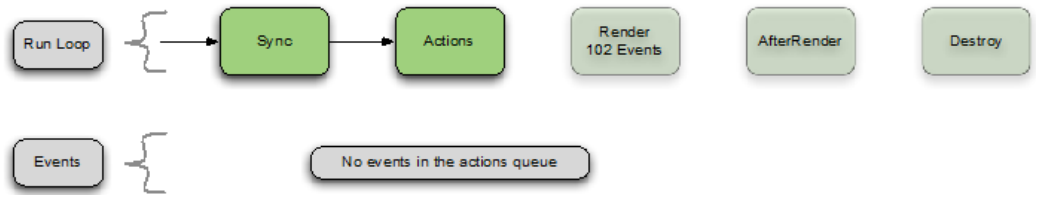


Figure 10.5 – After the Sync queue has been exhausted

Because there are no items in the Actions queue, and the Sync queue is still empty, The Run Loop will simply move along to the next queue.

The view packages add the two queues `Render` and `AfterRender` to the list of queues in the Run Loop. Once the Actions queue is exhausted, it is guaranteed that all of your applications bindings have been propagated throughout your application before it reaches the Render queue. This is important because interacting with and manipulating the browsers DOM tree is amongst the most time consuming tasks that Ember.js perform. Having a system implemented that ensures that your application will perform a minimum amount of DOM manipulation is critical for your applications performance, especially on mobile devices, which have limited processing capabilities. Figure 10.6 shows the state of the Run Loop as it starts working on the Actions queue.

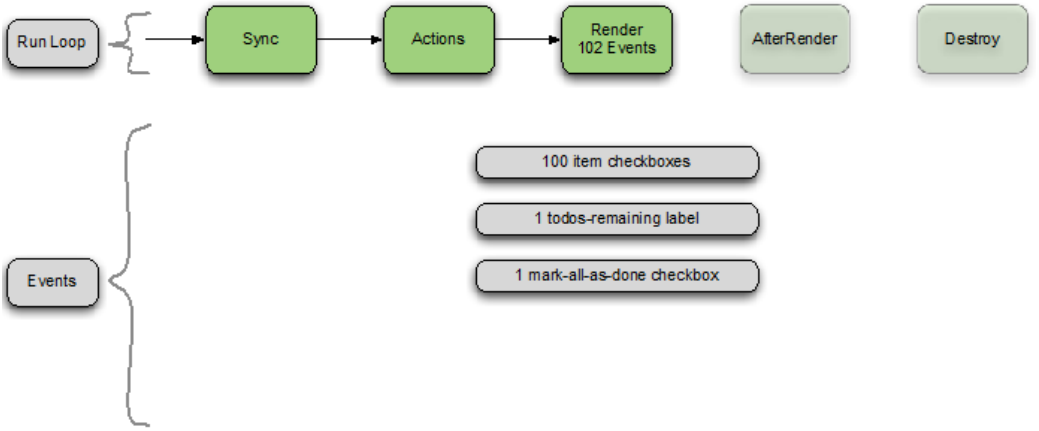


Figure 10.6 – After the Sync and Actions queues have been exhausted

The `Render` queue contains events related to view rendering into the DOM, as its name suggests. It is critical that this queue comes after the sync- and actions queues, as Ember.js uses this fact in order to keep the number of DOM manipulations to a minimum during the run loop. Once all of the bindings have been propagated in the sync queue, and all of the pending

callbacks and promises have been executed in the actions queue, everything is ready for Ember.js to start manipulating the user interface.

The `render` queue contains DOM-related events, and each of the items in this queue will generally result in one or more DOM manipulations. Most of your views will put its render events in this queue. Experience will show you, though, that some views' events will need to be scheduled *after* the `render` queue, and this is where – you guessed it – the `afterRender` queue comes in handy. Figure 10.7 shows the state of the Run Loop after the Render queue has been exhausted.

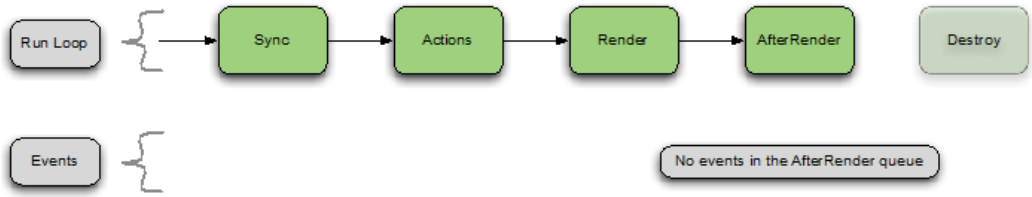


Figure 10.7 – After the Render queue have been exhausted

Summarized, the `afterRender` queue contains view-related events that need to occur after the normal render events have finished executing. One example of such a view is a view that needs to access the resulting DOM, as it will be after the `render` queue is exhausted. Typically this is used in views that need to append or alter the contents of a HTML-element that is rendered or updated as part of the `render` queue. Figure 10.8 shows the state of the Run Loop after the AfterRender queue has been exhausted.

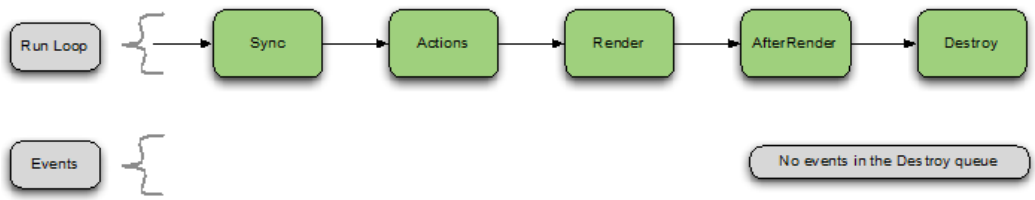


Figure 10.8 – After the Render queue have been exhausted

Once the Run Loop has finished emptying the AfterRender queue, it will move on to the final Destroy queue. In this queue, events for any objects that need to be destroyed will be added. You might be wondering what goes into the Destroy queue. Most often, any views that have been removed from the DOM and are not needed any more are added to this queue. This will happen when the user navigates from one route to another inside your application, or when the state changes so that different parts of a template get rendered into the DOM. Figure 10.9 shows the state of the Run Loop after the Destroy queue have been exhausted.

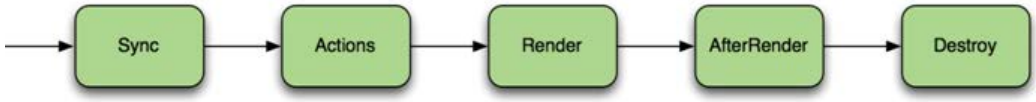


Figure 10.9 – After the Destroy queue has been exhausted

The Run Loop has one more trick up its sleeve though. Any events in the run queue might cause other events to be scheduled in any of the Run Loops queues. This means that events in the Actions queue might, for instance, schedule new events inside the Sync queue. If an event in the actions queue manipulates data within your application, that data might have its own observers or computed properties bound to it. If that is the case, new events will be added to the Sync queue in order to ensure that these changes are also taken care of before the Run Loop moves on to the next queue.

This is an important concept that is critical for the Run Loop in order to keep the user interface consistent with the changes while also minimizing the amount of DOM manipulations that your application needs to perform, the Ember.js Run Loop will ensure that all of the previous queues are completely exhausted before it moves on to the next queue.

Now that we have a more clear understanding, lets revise our conceptual overview of the Run Loop, as shown below in figure 10.10.

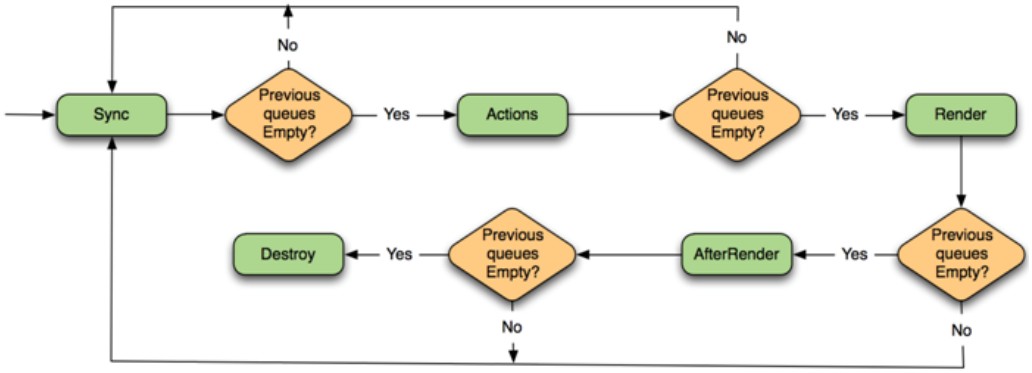


Figure 10.10 – The revised Ember.js Run Loop

Knowing how the Run Loop is implemented comes in handy once you get past the initial learning curve that Ember.js has, and your applications complexity grows. Understanding the run loop, you will be able to take advantage of how the run loop works in order to optimize the performance of your application.

I mentioned, however, that the run loop has an API that you can use to interact with it. Through this API you will be able to execute, schedule and repeat your applications code within the constraints of the run loop. The next section dives into the run loop API.

10.2 Executing Code Within the Constraints of the Run Loop

The run loop has a number of ways in which you can schedule code to be executed inside of a run loop. Using the run loops API it is possible to execute code immediately, after a set amount of time, during the next run loop. You might find this useful in situations where you need to manipulate the DOM elements after the didInsertElement function of your views have completed, or in situations where you need to rely on performing actions a specific amount of time after your view have rendered, etc. We will take a look at examples of each of these scenarios.

If you want to execute code immediately you need to wrap your code inside `Ember.run(callback)`. Calling this function will ensure that your code is executed within a run loop. If no run loop is currently executing, Ember.js will automatically start one. After a run loop has been started, the code within the callback is executed before the run loop is ended. Once the run loop has finished, the run loop has ensured that all of the queues are flushed properly.

This means that, as a developer you can be certain that any events and any bindings that your code affects have been properly executed and that the DOM is updated once the run loop reached its end.

10.2.1 Executing Code Inside the Current Run Loop

Executing code in the current run loop will be the most common run loop scheduling task you will write. Using Ember.run, Ember.js will place the given callback into the default actions queue.

Listing 10.1 below shows an example of attaching a piece of code to current run loop. In this specific example we are implementing part of a view that draws a line chart. Specifically we are looking at an observer that listens for changes to the views chart.series property. Because the server side won't always return a specific color for each of the series in the line chart, we need to tell the charting library (Rickshaw) to pick a new color from its palette and attach that color to each of the series where a color is not already defined. Once a new color is added to the series, we need to redraw the view.

Listing 10.1 - Executing Code in the Run Loop

```
Montric.ChartView = Ember.View.extend({
    contentObserver: function() {                                    #A
        var series = this.get('chart.series');
        if (series) {
            var palette = new Rickshaw.Color.Palette({scheme: "munin"});

            series.forEach(function(serie) {                         #B
                if (!serie.color) {
                    serie.color = palette.color()
```

```
                }
            });

            var view = this;
            Ember.run(function() {                                          #C
                view.rerender();
            });
        }
    }.observes('chart.series')                                              #A
});
```

#A: An observer function that observes the chart.series property
#B: Update the color for each of the series in the chart
#C: After the chart seires are updated, ensure that the view is rerendered

The view shown above is the main chart view from the Montric application. This view has an observer that listens for changes to the to the `chart.series` property. If `chart.series` has a value, we ensure that the chart series have a color defined, and if not we use a new value from the chart library's color palette. Once each chart series have been updated, we need to `rerender` the view in order to update chart present in the DOM. Because we want to ensure that the view is updated properly, we have wrapped the `rerender` function inside a callback function, passed into `Ember.run`.

But there are other options, which you can use to gain better control of how and when your code gets executed by the run loop.

10.2.2 Executing Code Inside the Next Run Loop

Sometimes you want to make sure that your code will be executed just after the current run loop has finished. There are two ways in which you can do this. You can either use the special `Ember.run.next()` function call, or you can use schedule the code to run in 1ms using the `Ember.run.later()` function call. In Listing 10.2 , we are revisiting the chart series observer from listing 10.1, only updated to use `Ember.run.next()`.

Listing 10.2 – Executing Code Inside the Next Run Loop Using Ember.run.next()

```
Montric.ChartView = Ember.View.extend({
    contentObserver: function() {
        //Code ommited, but same as listing 10.1

        var view = this;
        Ember.run.next(function() {                                         #A
            view.rerender();
        });
    }.observes('chart.series')
});
```

#A: Using Ember.run.next() to execute code inside the next run loop

As you can see, the code in the listing above is the same as the one we saw earlier in listing 10.1, with the exception that we have replaced the call to `Ember.run()` to

`Ember.run.next()`. The signatures for both of these functions are the same. The effect of this change is that the rerendering of the view will wait until all bindings and observers have been properly propagated and until all DOM updates that are scheduled in the current run loop have been properly carried out. This might be useful if you want to apply animations to a DOM element after it has been rendered, and if you need to schedule an event to occur after the events in the Render, AfterRender and Destroy queues have been carried out.

10.2.3 Executing Code Inside a Future Run Loop

I mentioned that it is possible to use `Ember.run.later()` in order to schedule code to be executed within the next run loop. In Montric, the application is receiving updated information every 15 seconds. Because we want to be able to keep the user interface updated without the user having to trigger an "update" action, we need to be able to update visible charts every 15 seconds automatically. Using Ember.run.later, will allow us to implement this type of functionality. Listing 10.3 shows an example of how `Ember.run.later()` is used.

Listing 10.3 – Executing Code Inside the Next Run Loop Using Ember.run.later()

```
Montric.ChartView = Ember.View.extend({
    contentObserver: function() {
        //Code ommited, but same as listing 10.1

        var view = this;
        Ember.run.later(function() {                                    #A
            view.rerender();
        }, 1);                                                          #B
    }.observes('chart.series')
});
```

#A: Using Ember.run.later() to execute code inside the next run loop
#B: Scheduling the task to be added to a run loop starting 1 millisecond after the current run loop have ended

As you can see from the code above we have replaced the call to `Ember.run.next()` with a call to `Ember.run.later()` while adding an additional parameter which specifies the number of milliseconds that the code will be scheduled to execute in. In this case we are scheduling the code to execute in the run loop starting 1 milliseconds after the current run loop finishes, which will most likely be the next run loop.

You might have guessed by now, that it is also possible to schedule tasks to be executed at a specific time in the future using `Ember.run.later()`. By changing the number passed in as the final parameter to 500, the code given will be executed in the first run loop starting after 500 milliseconds have passed.

In fact, there are many ways in which you can schedule a task at a specified time in the future, and so far we have only been scheduling code into the `actions` queue. You might be wondering how it's possible to schedule code into one of the other queues, or how it is possible to schedule a repeated task. This is where `Ember.run.schedule()` comes in.

10.2.4 Executing Code Inside a Specific Queue

Most of the time you will want to schedule code into the `actions` queue. At that point all bindings have been propagated and everything should be set up for most of the code that you would want to schedule to execute inside a run loop.

Once in a while though, you want to have fine-grained control of which queue you want to schedule your code in. In the Ember Fest application that we looked at in Chapter 6 we need to schedule code into the `afterRender` queue. The reason for this is that the Ember Fest landing page is built up from 6 sub-routes that are organized as a single page where the user can scroll down in order to reveal the contents of the sub routes. As the user scrolls down, the route changes and the URL updates to reflect where on the page the user is at that moment. In addition, if the user enters the application at one of the visible sub routes, the application needs to scroll the page to the correct spot on the page. Figure 10.11 shows how a few of the routes from the Ember Fest application, and which Routes each part of the website is associated with.

Figure 10.11 – The Ember Fest Application showing its “scrollable routes”

This scrolling-functionality is added into each of the sub routes using `document.getElementById(id).scrollIntoView()`. Consider the code in listing 10.4.

Listing 10.4 – Scrolling to a Specific Element in the DOM

```
Emberfest.IndexVenueRoute = Ember.Route.extend({                          #A
    setupController: function(controller, model) {
        this._super(controller, model);
        _gaq.push(['_trackPageview', "/venue"]);                          #B

        document.title = 'Venue - Ember Fest';
    },

    renderTemplate: function() {                                          #C
        this._super();
        document.getElementById('venue').scrollIntoView();               #D
}
});
```

#A: The Route for the /venue URL
#B: Integrating the route with Google Analytics
#C: Adding the scrolling functionality to the renderTemplate function
#D: A non-working example of scrolling to the correct position in the DOM

As you can see from the above code we are attempting to scroll the HTML element that corresponds to the ID "venue" by using the `scrollIntoView()` function. The problem though with this code, is that the template in which the venue element is part of wont be rendered into the DOM when the `renderTemplate` function is executed. The code from listing 10.4 will result in the following error message:

```
TypeError: 'null' is not an object (evaluating
'document.getElementById('venue').scrollIntoView')
```

This is obviously not what we want. Instead we want to scroll the venue element into view just after it has been rendered in the DOM. If you remember back, the element will be drawn to the DOM inside the render queue, so lets try to schedule the code into the `afterRender` queue using `Ember.run.schedule()`. Listing 10.5 shows the updated code.

Listing 10.5 – Scheulding Code Into the afterRender Queue using Ember.run.schedule()

```
Emberfest.IndexVenueRoute = Ember.Route.extend({
    renderTemplate: function() {
        this._super();
        Ember.run.schedule('afterRender', this, function(){              #A
            document.getElementById('venue').scrollIntoView();
        });
    }
});
```

#A: Using Ember.run.schedule to schedule code into the afterRender queue

We have omitted the `setupController` code in the above example, but it remains the same as before. The only difference is that we have now wrapped the `document.getElementById('venue').scrollIntoView()` inside `Ember.run.schedule()`. `Ember.run.schedule()` takes three arguments. The first one specifies which queue to schedule the task in, while the second takes the context in which the callback will be executed in.

Now, the above code works as expected. When the user enters the `IndexVenueRoute`, the routes `setup()` function is called, which in turn triggers the `renderTemplate()` function. Ember.js will internally defer any rendering logic to the render queue, which include rendering the template that belongs to the `IndexVenueRoute`. This means that, once the run loop enters the `afterRender` queue, that the venue element that we are looking for will already have been added to the DOM. This means, that at this point in time, we are able to scroll the page down to the correct element.

We have looked at how we can schedule single tasks into the run loop in different ways, but so far we haven't looked at how we can use the run loop to implement repeated tasks.

10.2.5 Executing Repeated Tasks With the Run Loop

Sadly, Ember.js does not implement an `Ember.run.interval()` function. This makes it harder to implement repeated functionality in your Ember.js applications, which means that you have to choose to either

- Fall back to the standard JavaScript `setInterval()` function, wrapping its contents in `Ember.run()` or `Ember.run.schedule()`.
- Use `Ember.run.later()` and make sure that the callback you pass in will recursively add another call to `Ember.run.later()`.

Listing 10.6 shows how the Montric application implements updating the loaded charts every 15 seconds. The ChartsController have implemented both a startTimer() and a stopTimer() function that will create a new interval, store it on the controller and clear it. In addition to the code shown here, the controller will call the startTimer() and stopTimer() functions based on the current state of the ChartsController, as well as the state the application is in.

Listing 10.6 – How to execute repeated tasks inside the Run Loop

```
Montric.ChartsController = Ember.ArrayController.extend({
     startTimer: function() {                                          #A
         var controller = this;
        var intervalId = setInterval(function () {                     #B
            Ember.run(function() {                                     #C
                    if (controller
                       .get('controllers.application.showLiveCharts')) {
                     controller.reloadCharts();
                }
            });
        }, 15000);

        this.set('chartTimerId', intervalId);
    } ,
```

```
    stopTimer: function() {                                          #D
        if (this.get('chartTimerId') != null) {                      #E
            clearInterval(this.get('chartTimerId'));
            this.set('chartTimerId', null);
        }
    }
});
```

#A: The startTimer function will start an interval and store it in the controller
#B: Start an interval, executing every 15000 milliseconds
#C: Execute the contents of the interval inside the current run loop
#D: The stopTImer function will terminate the interval
#E: If an interval is stored on the controller, clear it and reset it to null

As you can see from the listing above, we have wrapped the contents of the interval function inside `Ember.run()` in order to ensure that that piece of code is executed within the current run loop. This is important in order to ensure that any changes that occur within your interval function will trigger a Run Loop and keep your applications user interface up to date.

10.3 Summary

Throughout this chapter we have looked at how the Run Loop and backburner.js are used throughout the Ember.js framework in order to ensure that your Ember.js based applications is as fast as possible, while staying out of your way as much as it can. In most cases, Ember.js will schedule your code into the correct run loop queue without you even knowing that the run loop exists.

That being said, there are a few edge cases where knowing how the run loop works, and how you can utilize its queue in order to serve your applications needs properly comes in handy. In these cases, scheduling tasks into the right run loop queue, or scheduling code to be run in a future run loop will most likely lead to more readable code.

We have seen that by using the run loops API, you will be able to schedule tasks into both into the current run loop, or into a future run loop. You will also be able to schedule tasks into a specific run loop queue.

Now that we have seen most of what Ember.js has to offer, it is time to take a closer look at how we can package and deploy our Ember.js applications.

11

Packaging and Deployment

This chapter covers:

- The concepts behind JavaScript application packaging and assembly
- A discussion about project structure
- A detailed look at the steps involved, file minimizing, concatenation and template compilation
- A look at using grunt.js

The state of JavaScript build tools are, sadly, at its infancy still. There are a range of different products that all compete to solve the tasks of application assembly; running unit tests, performing linting (ensuring that you JavaScript code is clean), source code minification and application packaging.

The problem with the tools that exist, is that they tend to be somewhat hard to use and give error messages that is hard to understand, even for developers. In addition, the JavaScript community has yet to agree on a standard for managing application dependency and how these dependencies should be bundled with your application. If you are entering the JavaScript world from more mature server-side programming languages like Java and .Net, the JavaScript build tools and the assembly pipeline will leave a lot to desire.

This chapter will be split into two parts. The first part will describe the steps that are necessary in order to assemble and package your Ember.js application into a format that is suitable for deployment and for sending out to your users' browsers. The second part will explain how to achieve these tasks using the tool that is most popular at the time of publication, Grunt.js.

Before we move on, though, lets review the parts of Ember.js we will be working on in this chapter.

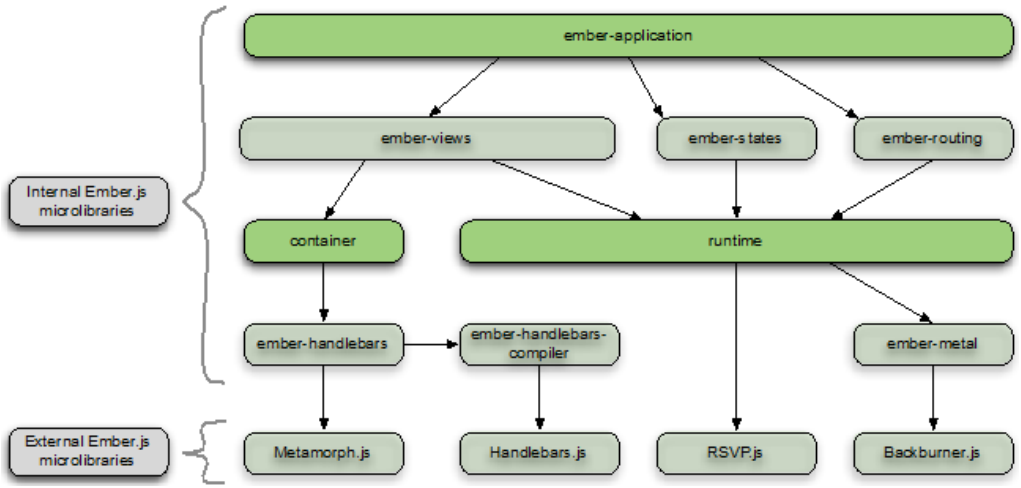


Figure 11.1 The parts of Ember.js we will be working on in this chapter

11.1 The Concepts Behind JavaScript Application Packaging and Assembly

In order to be able to use any build tool, you need to structure your applications source code files properly. There are many ways in which you can build up this structure, but most importantly, you will need to keep your different source code file types in separate directories. This means that all of your JavaScript files need to go either directly into, or into a subdirectory of a single directory inside your project. The same is true for CSS files, HTML files and any other assets that you wish to include in your application. So before we move on, let's have a look at how we can structure an Ember.js application in order to prepare it for packaging and assembly later.

11.1.1 Preparing the directory structure

As I mentioned, the JavaScript community hasn't agreed on a standard directory structure for JavaScript applications. This means that you are free to name your directories whatever your want. As long as you keep your different assets types separate from each other, you will be able to tell your build tool where to find them later.

On the bright side, however, the JavaScript community is slowly, but surely, building up a consensus that eventually will lead to a standard way to define your JavaScript projects directory structure.

Until then, it's wise to attempt to follow a few sane guidelines. For my Ember.js applications I like to separate my assets in the following directories:

- "js/app" – Any self-written JavaScript files that is part of the project
- "js/lib" – Any third party JavaScript files. Some of these files might already be minimized

- "js/test" – Any JavaScript unit tests that are part of the project
- "css" – All of your projects CSS files go directly into this folder
- "images" – All of your projects images.

> **NOTE** Some build tools require a specific directory structure, that it either prefers or requires your code to adhere to. In our case, because we are using Grunt.js, we will tell Grunt.js where to locate our different files. This means that you are completely free to use your own directory-structure style.

By structuring your application in this way, it will be easy for your application, your developers and your build tool to be able to find your assets later on. Most of the directories above are flat, meaning that they don't have any sub-directories. This is, however, most likely not the case for the "js/app" directory, as we will see below.

STRUCTURING YOUR CUSTOM WRITTEN SOURCE CODE

Having a sane structure for your source code is extremely important. Because I cannot stress this fact enough, I will repeat it: Having a sane structure for your source code is extremely important! You have probably seen a lot of Ember.js application examples where all of the JavaScript application code resides inside a single app.js file.

Writing applications this way is great and very efficient when you are **writing small, proof of concept or example** applications. If you intend to be able to maintain your application later, and if you are serious about ever being able to deploy your awesome Ember.js application into production, you need to separate your source code out into different files.

Other, statically typed languages employ a structure where you can only define a single entity inside a single file. In my opinion this is the only sane strategy to use for JavaScript applications too. This means that you end up with a lot of files inside your application. There are generally two approaches to bring a sense of structure to this chaos: Keeping objects of the same type together, or keeping your features together. Which of these two strategies you choose is ultimately up to you, but lets take a look at how to organize your code both ways, and why I would recommend keeping your features together.

KEEPING OBJECTS OF THE SAME TYPE TOGETHER

Keeping object of the same type together, means that you will create directories inside "js/app" where you collect, for example, all controllers, all views, etc. This will ultimately lead to a structure somewhat similar to figure 11.2 below.

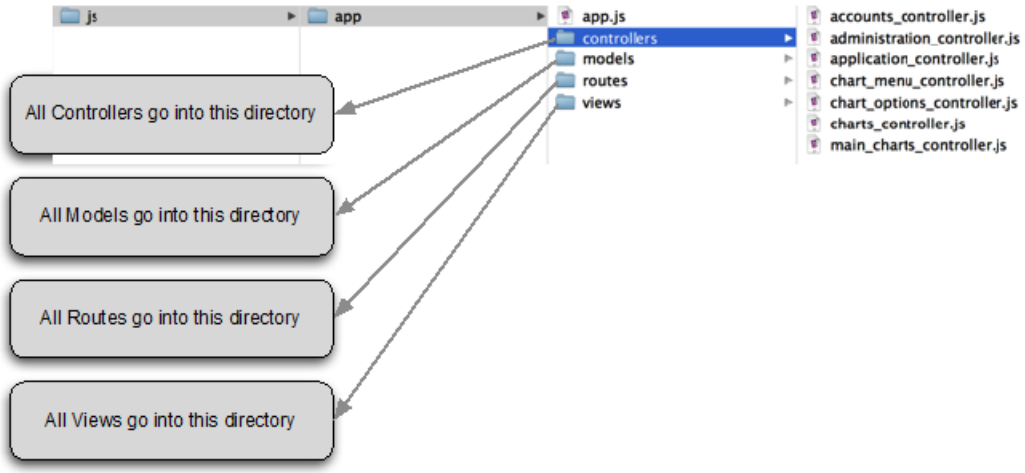


Figure 11.2 – Keeping objects of the same type together

As you can see, we have directories labeled “controller”, “models”, “routes” and “views”. As long as your application is small, this approach is manageable. But even in the example shown in figure 11.1, you can probably see where this approach will be hard to maintain in the long run. If you look inside the “controllers” directory, you will quickly see that there is no quick-and-easy way to distinguish between your files and their responsibilities within your application. You can guess that the accounts_controller.js file is responsible for administrating user accounts, but because this directory structure does not give you any context for each of your controllers you won’t be able to know until you examine the contents of the file.

KEEPING YOUR FEATURES TOGETHER

The other approach is to define your directory structure bundled into the features that your application provides. For an Ember.js application this means that any code that is part of a single route within your application will live in the same folder. You will then end up with folders named “administration/accounts” or “chart” that contain separate files defining that routes’ route, controller and view. Figure 11.3 shows an example of this approach.

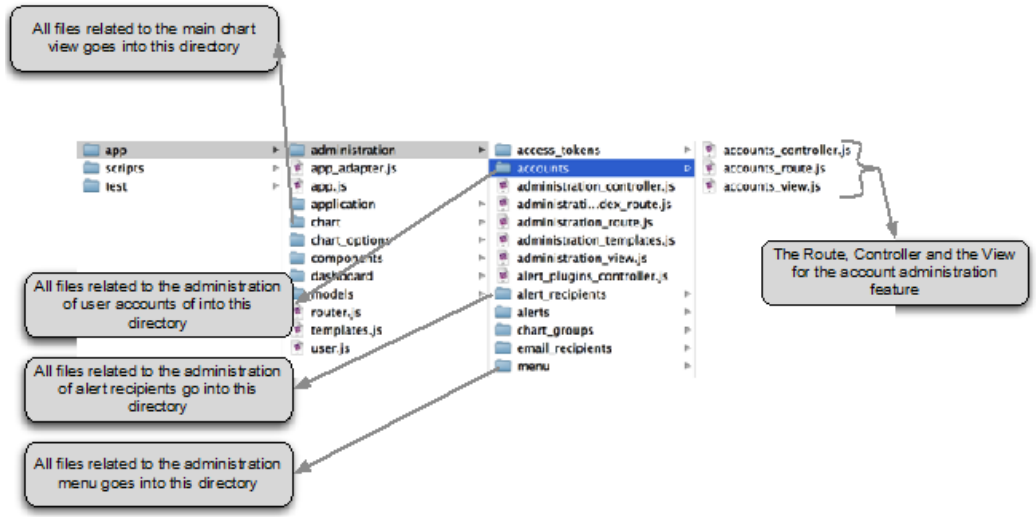


Figure 11.3 – Keeping Your Features Together

Here you will see that the "js/app/administration/accounts" directory contains all of the files that together define the functionality for the "administration/accounts" route within the Montric application. In addition, you can see that there are directories for "administration/alert_recipients" and "administration /menu", etc.

This directory structure takes care of three important aspects of your applications structure:

- Separating your logic into small, maintainable files
- Each file have a predictable location within you "js/app" directory
- Your directory structure will reflect the main features that your application provides, making your application easier to maintain in the future while also being easier for new developers on your team to understand.

THE "JS/TEST" DIRECTORY

You might wonder what to do with the "js/test" directory. In my opinion you should structure this directory in the same way as you structure your "js/app" directory. This means that you will end up with directories like "js/test/administration/account" and "js/test/administration/alerts". As with your "js/app" directory, structuring your tests in this manner takes care of three aspects:

- Whenever a test fails your test runner will give you feedback about what feature that has failing tests. If a test is failing inside the "js/test/administration/account" directory, you can quickly without any detective work know that you have introduced a bug that affects your account administration
- Each file have a predictable location within your "js/test" directory

- Glancing over the directory structure of your tests will quickly give you a sense of which parts of your applications that are well tested and which parts you should look closer at

The Top-Level Directory of Your Project

This only leaves the top-level directory of your project. In my opinion this directory is reserved for files that define something about your application. This includes, but is not limited to

- index.html – Defining the overall structure of your application
- A file describing any third party dependencies that your application has
- A build file telling a build tool how to assemble and test your application

There is, however, one more directory structure that we haven't talked about – templates.

Structuring Your Templates

I recommend that you put each of your templates into their own separate files, suffixed with the custom .hbs extension. In fact, you can add any suffix you want for these files, as we will specify this when we are assembling the application later. These files should be named after the routes within your application, which will serve three purposes:

- Separating each template into its own file
- Grouping related templates together into a sensible directory structure that resembles your applications features
- The directory name and file name combination will tell your build tool how to compile your templates into JavaScript code, and, more importantly, what to call your templates automatically.

Figure 11.4 shows the templates structure for Montric.

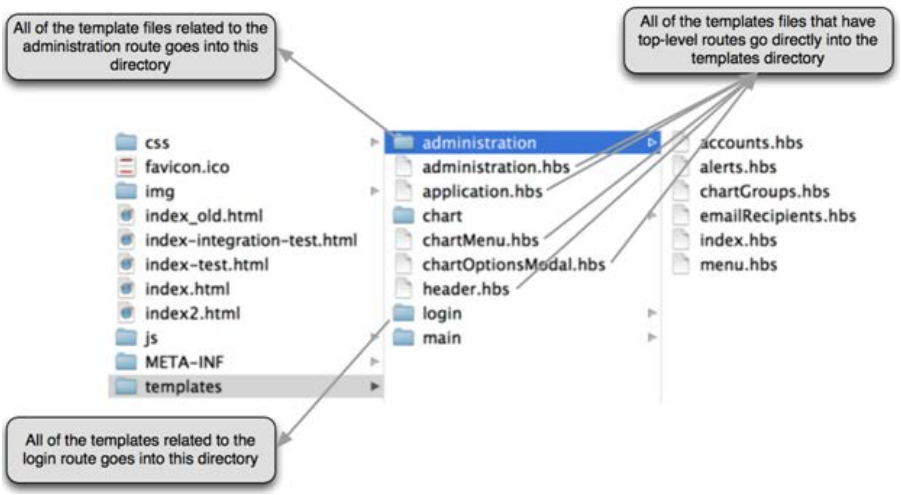


Figure 11.4 – The templates directory structure

As you can see, this will ensure that you end up with a directory “templates” inside which you have a hierarchical directory structure that exactly maps your applications routes. This is both predictable for the developer and easy for a build tool to make use of when pre-compiling your templates into JavaScript code.

Now that we have had a look at how you should structure your Ember.js application, lets move on and explore the Ember.js application assembly process.

11.1.2 The Ember.js Application Assembly Process

Once you have structured your application into separate files to make development easier and your application easier to maintain, the assembly process involves combining all of your applications assets into single files. This means that all of your JavaScript code needs to be combined into one file, while all of your CSS code needs to be combined into another file. In addition, each of these files needs to be minimized in order to cut down on the amount of data that you will send across the wire from the server to the client.

Summarized, the JavaScript application assembly process includes the following steps. You don’t have to know what each step does, or what they mean at this point. We will discuss each concept in detail as we work our way through the rest of this chapter.

- Find each file inside the “js/app” directory
- Lint the file for common JavaScript errors
- Minify the file
- Concatenate with previous files

Because Ember.js also have templates spread across multiple files, you will also need to perform the following tasks on your templates.

- Find each *.hbs file
- Take the contents of each file and assign it to Ember.TEMPLATES[dirname/filename] = Ember.Handlebars.compile(fileContents)
- Concatenate with previous templates

In addition to the JavaScript and templates, the CSS files also need to be combined and minified. This process is a bit simple and involved the following tasks.

- Find each *.css
- Minify the contents
- Concatenate with previous CSS

There are multiple steps involved in this process, so lets take a closer look at how they are related. The Ember.js application assembly process is shown in figure 11.5 below.

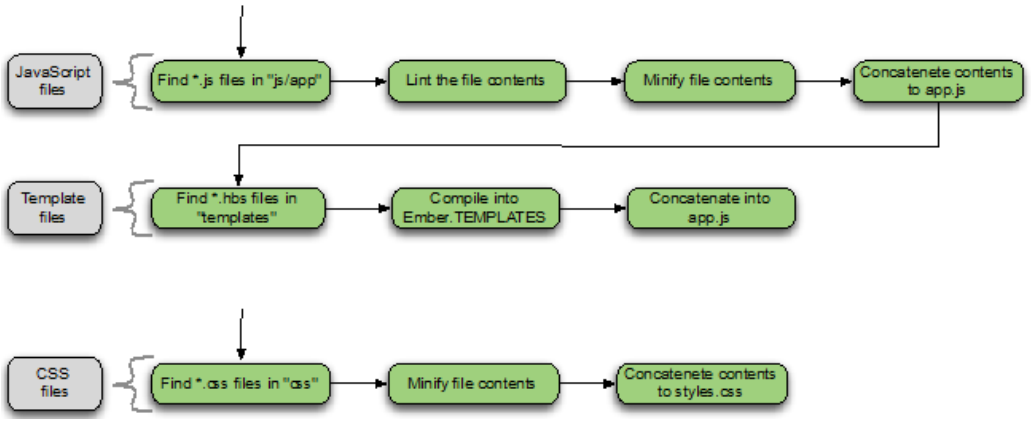


Figure 11.5 – The Ember.js JavaScript application assembly process

The figure above shows the complete Ember.js application assembly process, and includes assembly of your JavaScript source code, your Handlebars.js templates, as well as your CSS files. The goal is to end up with two single files, one file containing all of your applications custom JavaScript as well as a single file containing all of your applications CSS.

Now that we have had a look at how you should structure your Ember.js application, lets move on and explore how we can use grunt.js in order to build and assemble your application.

11.2 Using grunt.js as You Build Tool

Grunt.js describes itself as "the JavaScript Task Runner". It is a third party application, written in JavaScript, and running inside of Node.js. Its main task is to be able to automate the process of script minification, linting, running unit tests and concatenating your applications different assets into single files. Grunt.js is built around the notion of a pipeline. This pipeline defines how your application will get assembled and tested. Grunt.js expects to find two files inside your application's top-level directory, a Gruntfile.js as well as a package.json file.

Grunt.js is a node.js application, which mean that you need to have the Node Package Manager (npm) installed on your system in order to get started. Head over to http://nodejs.org in order to install Node.js, and to http://npmjs.org in order to install NPM.

We will implement a complete application assembly pipeline with Grunt, including

- Bootstrapping the Grunt.js build system
- Concatenating our JavaScript Files into a single file
- Applying Linting to the concatenated file
- Compiling templates into the concatenated file
- Minifying the concatenated file into a production, ready for deployment

Note, that Grunt.js is plugin-based, and any tasks that we want to perform will be executed by a Grunt.js plugin. We will be installing the plugin we need to assemble the Montric application as we go along throughout the chapter.

We have a lot to go though, so lets get started by bootstrapping the Grunt.js build system for Montric. After we have gone through how to assemble an Ember.js application with Grunt.js, we will discuss the advantages and disadvantages of using Grunt.js.

11.2.1 Bootstrapping Montrics Grunt.js build

The first thing we need to do is to add a basic package.json file, which will describe the Montric application, along with any dependencies that either the application or the build pipeline requires. Listing 11.1 shows our initial package.json file.

Listing 11.1 – The initial package.json file

```
{
  "name": "Montric",                                    #A
  "version": "0.9.0",                                   #B
  "devDependencies": {                                  #C
    "grunt": "~0.4.1"                                   #D
  }
}
```

#A: The name of your project. In this case, Montric
#B: The current version of your project
#C: A list of dependencies that grunt needs in order to assemble your application
#D: We only require grunt.js, version 0.4.1 or newer to get started

As you can see, we need to specify a few details in order to get started inside the package.json file. This file tells Grunt.js the name of your project, its current version, as well as any `devDependencies` that Grunt.js requires in order to assembly your project. We will be adding additional dependencies inside the `devDependencies` section as we build up the Montric assembly pipeline.

Next, we need to add a Gruntfile.js file. This file will describe how your application is going to be assembled. The initial Gruntfile.js is shown in listing 11.2.

Listing 11.2 – The initial Gruntfile.js

```
module.exports = function(grunt) {                      #A

  // Project configuration.
  grunt.initConfig({                                    #B
    pkg: grunt.file.readJSON('package.json'),           #C
  });

  // Default task(s).
  grunt.registerTask('default', []);                    #D

};
```

#A: Grunt.js will execute any code inside the module.exports function

#B: Any initial configurations go into grunt.initConfig
#C: The configuration starts out by reading the package.json configuration into the pkg namespace
#D: Register the tasks that Grunt.js will execute by default. At this point in time, we haven't added any task yet.

At this stage, our gruntfile won't be performing any task. However, we have added just enough information in order to be able to execute grunt. But before we can get going, we need to install the Grunt.js command line interface (CLI).

To install the Grunt.js CLI, open up Terminal (Mac/Linux) or Command Prompt (Windows) and type in `npm install –g grunt-cli`. This will cause NPM to install grunt-cli into your global environment, allowing the `grunt` command to be executed from any directory in your system.

We can now test our Gruntfile.js and package.json configuration. Navigate to you project directory, and type in the command grunt. When you hit enter Grunt.js will attempt to assemble your application using the default task that we created above. The result should be similar to Figure 11.6.

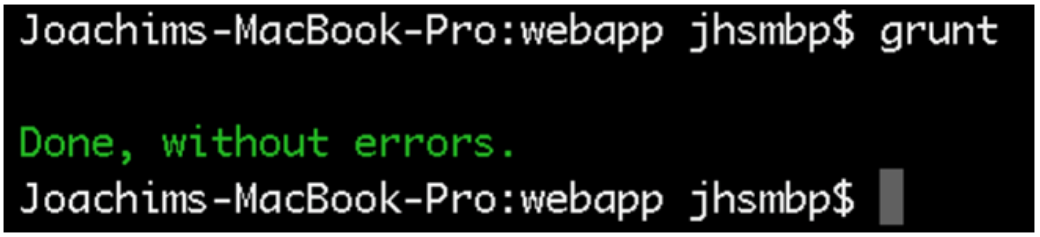


Figure 11.6 – Executing grunt

Now that we have a working Grunt.js setup, lets start adding a task to concatenate all of our JavaScript code into a single file.

NOTE All the screenshots in this chapters is taken after running the command on my Mac with MacOS X 10.8, and Grunt.js version 0.4.1. Your output might vary slightly depending on your operating system and version of Grunt.js and Grunt.js plugins.

11.2.2 Concatenating the JavaScript Code

Now that we have a working application build pipeline set up, lets review where we are in the assembly process, as shown below in figure 11.7.

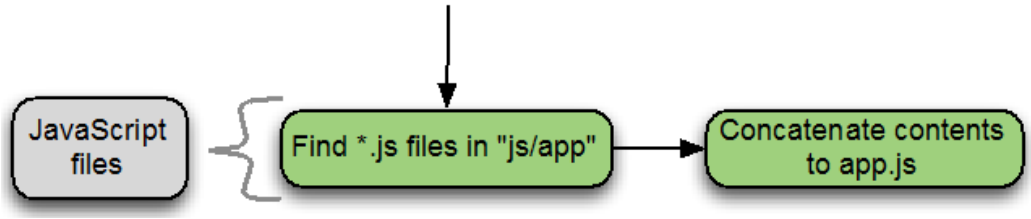


Figure 11.7 – The current stage in the assembly process

The very first thing we need to do, is to install Grunt.js' concat plugin. We can do this via the command line by executing `npm install grunt-contrib-concat --save-dev`. This will download and install the most recent version of the grunt-contrib-concat plugin into your project directory. The –-save-dev parameter will tell NPM to also include this version into your package.json file. Figure 11.8 below, shows the result of executing the above command.

```
Joachims-MacBook-Pro:webapp jhsmbp$ npm install grunt-contrib-concat --save-dev
npm WARN package.json Montric@0.9.0 No README.md file found!
npm http GET https://registry.npmjs.org/grunt-contrib-concat
npm http 200 https://registry.npmjs.org/grunt-contrib-concat
npm http GET https://registry.npmjs.org/grunt-contrib-concat/-/grunt-contrib-concat-0.3.0.tgz
npm http 200 https://registry.npmjs.org/grunt-contrib-concat/-/grunt-contrib-concat-0.3.0.tgz
grunt-contrib-concat@0.3.0 node_modules/grunt-contrib-concat
```

Figure 11.8 – Executing npm install grunt-contrib-concat --save-dev

As you can see, NPM have identified our existing package.json file. After the package.json file has been identified, NPM downloads the grunt-contrib-concat plugin. In addition, NPM will extend our package.json configuration by including a reference to the plugin we just installed. If you open up your package.json file, it should look like listing 11.3.

Listing 11.3 – The Resulting package.json File After Installing the concat Plugin

```
{
  "name": "Montric",
  "version": "0.9.0",
  "devDependencies": {
    "grunt": "~0.4.1",
    "grunt-contrib-concat": "~0.3.0"                                    #A
  }
}
```

#A: NPM have added a reference to version 0.3.0 (or newer) of grunt-contrib-concat

Next, we can extend Gruntfile.json in order to concatenate all of our JavaScript code into a single file. Listing 11.4 shows the updated Gruntfile.js file.

Listing 11.4 – Adding Concatenatino to the Gruntfile.js File

```
module.exports = function(grunt) {

  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    concat: {                                                    #A
       options: {
         separator: '\n'                                         #B
       },
       dist: {
         src: ['js/app/**/*.js'],                                #C
         dest: 'dist/<%= pkg.name %>.js'                         #D
       }
     }
  });

  grunt.loadNpmTasks('grunt-contrib-concat');                    #E
  grunt.registerTask('default', ['concat']);                     #F
};
```

#A: Configuration for the concat plugin
#B: Define a string that is injected between each file in the concatenated result
#C: Define where the source files are located
#D: Define where to put the resulting concatenated file
#E: Load the grunt-contrib-concat plugin
#F: Register the concat plugin to run in the default task

As you can see, we have added quite a bit of information to our Gruntfile.js file. The most notable addition is the `concat` object, into which we have added the configuration require for the `grunt-contrib-concat` plugin to work. First, we have defined an options object where we can put the options that the plugin will use. In our case, we are simply telling the `grunt-contrib-concat` plugin to append a new line in between each of the JavaScript files when it concatenates the files into a single file.

Next, the `dist` object, are used to define where the plugin will find the JavaScript files to concatenate, as well as which file it will output the results to when it finishes. The `src` property tell the plugin to locate any `.js` file inside any subdirectory of "`js/app`", while the `dest` property tell the plugin to write the concatenated output to a file inside the dist directory. The name of this file, is specified within our package.json file as the `name` property. I our case, the final build file will be named "Montric.js".

Finally, we need to load the task into NPM and add the `concat` plugin to the default task. We will now be able to assemble our applications JavaScript code into a single file by executing grunt at the command line. Figure 11.9 shows the result of executing grunt.

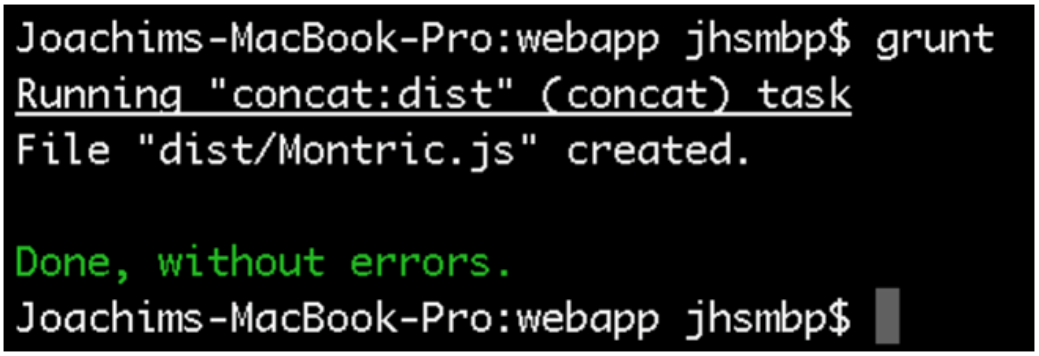


Figure 11.9 – Executing grunt to concatenate the JavaScript files

Before we move on, I'd like to clean up the code inside the Gruntfile.js file a little. As you can probably imagine, if we add all of our plugins directly into the grunt.initConfig function, this file would pretty soon become rather large and hard to navigate. So before we move on, lets look at how we can extract the configuration of each of the plugins out to separate files.

11.2.3 Extracting Plugin Configuration to Separate Files

In order to keep our main Gruntfile.js file as small and readable as possible lets take a look at how we can extract our plugin configurations into separate files. The first thing we are going to do is to create a new directory called "tasks" into which we will place our plugin configuration files. Inside this directory, lets create a file named concat.js, the content of which is shown in listing 11.5.

Listing 11.5 – Extracting the concat Plugin configurations into tasks/concat.js

```
module.exports = {                                                    #A
    options: {                                                        #B
        separator: '\n'
    },
    dist: {                                                           #C
        src: ['js/app/**/*.js'],
        dest: 'dist/<%= pkg.name %>.js'
    }
};
```

#A: Wrap the contents inside module.exports
#B: Include the options object from Gruntfile.js
#C: Include the dist object from Gruntfile.js

As you can see, we have extracted to contents of the old `concat` object from the Gruntfile.js file and added it into a `module.exports` object in the new concat.js file. Other that that, the contents of the grunt-contrib-concat plugin configuration remains the same as before.

Next, we need to tell Grunt.js to use this file instead of the `concat` object. Listing 11.6 shows the updated Gruntfile.js.

Listing 11.6 – The Updated Gruntfile.js file

```
function config(name) {                                          #A
  return require('./tasks/' + name);                             #C
}

module.exports = function(grunt) {
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
     concat: config('concat')                                    #D
  });

  grunt.loadNpmTasks('grunt-contrib-concat');
  grunt.registerTask('default', ['concat']);
};
```

#A: Adding a config function will bring in the new concat.js file
#B: Using require in order to bring in the file from the tasks directory
#C: Calling the new config function instead of defining the options directly inside Gruntfile.js

As you can see this step have greatly reduced the footprint while at the same time increased the readability of the Gruntfile.js file. In addition we have also created a mechanism where its simple to load new plugins into the Grunt.js configuration.

Now that we have created a simple way of adding new plugins to out Grunt.js assembly process, lets look at how we can perform linting in order to reduce the likelihood of our source code to include obvious bugs or API calls considered bad.

11.2.4 Linting Out Common Bugs

Linting is a process that will analyze your code for common bugs and potential errors. In the world of JavaScript applications, due to JavaScript being an interpreted language, linting also involves syntax verification. Figure 11.10 shows where we are in the assembly process.

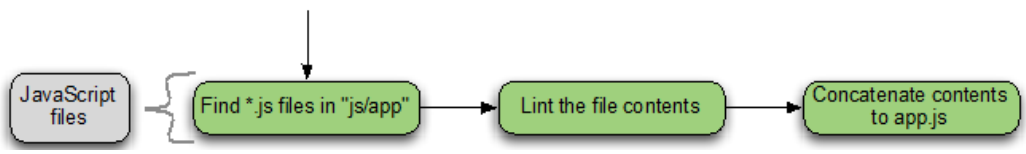


Figure 11.10 – Adding Linting to the assembly process

The first thing we need in order to get started, is to install and add the grunt-contrib-jshint plugin. This is done by executing `npm install grunt-contrib-jshint --save-dev` via the command line, as shown in figure 11.11.

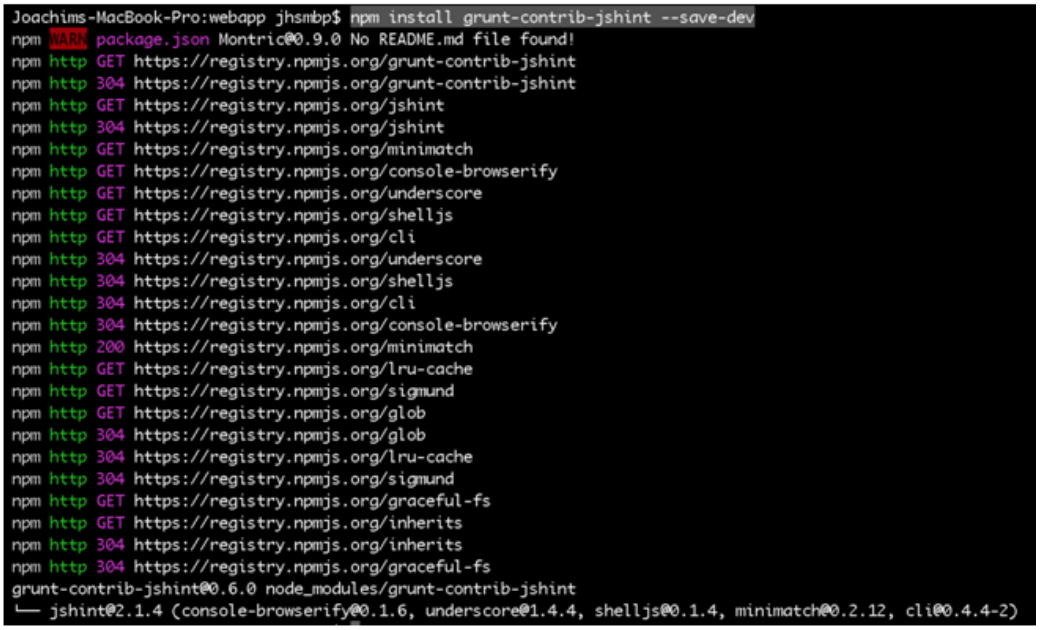


Figure 11.11 – Installing the grunt-contrib-jshint plugin

Now that we have installed the grunt-contrib-jshint plugin, we are ready to start adding linting functionality to our build process. Create a new file inside the tasks directory named jshint.js, with the contents shown below in Listing 11.7.

Listing 11.7 – Creating the jshint.js File

```
module.exports = {
    files: ['Gruntfile.js', 'js/app/**/*.js', 'js/test/**/*.js'],        #A
    options: {                                                           #B
        globals: {
            jQuery: true,
            console: true,
            module: true
        }
    }
};
```

#A: The files array tell the grunt-contrib-jshint plugin which files it will perform linting on
#B: In the options object we can define options to jshint

The code above tells JSHint that it will perform linting on any `.js` files inside both the `js/app` and `js/test` directories, as well as on the main Gruntfile.js file. Additionally, we are providing JSHint with a couple of global parameters. The possible options are defined for the JSHint project, and can be found at http://www.jshint.com/docs/.

The final step we need to perform is to include our new jshint.js file into our Gruntfile.js file and add it to Grunt.js' default task. The updated Gruntfile.js is shown in listing 11.8.

Listing 11.8 – The Updated Gruntfile.js

```
function config(name) {
     return require('./tasks/' + name);
}

module.exports = function(grunt) {
     grunt.initConfig({
            pkg: grunt.file.readJSON('package.json'),
            concat: config('concat'),
            jshint: config('jshint')                                    #A
     });

     grunt.loadNpmTasks('grunt-contrib-concat');
     grunt.loadNpmTasks('grunt-contrib-jshint');                        #B
     grunt.registerTask('default', ['jshint', 'concat']);               #C
};
```

#A: Adding jshint to the Grunt.js configuration
#B: Loading the grunt-contrib-jshint pllugin
#C: Adding jshint to the default task

If you run the grunt command from the command line, linting will be performed for all of your JavaScript source files. Hopefully you wont receive any errors, but most likely JSHint will report a couple of areas where your code could be improved. The result of running the grunt command is shown in figure 11.12.

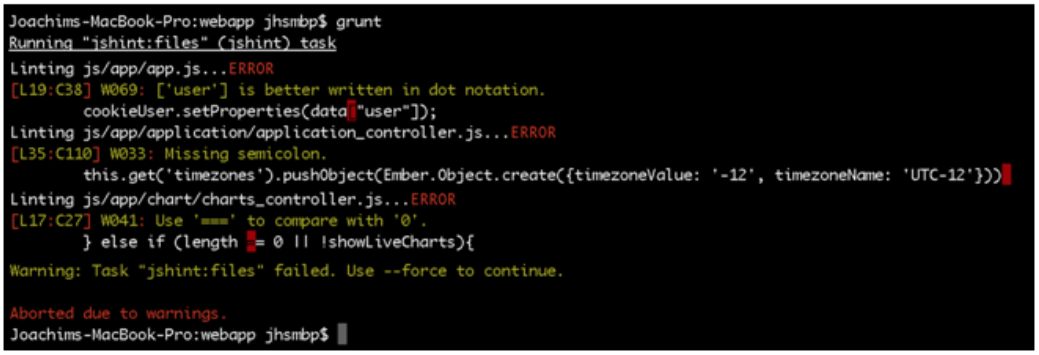


Figure 11.12 – The result of running JSHint

As you can see, JSHint found a few errors in our source code. As you can see, JSHint start out by listing the file that it performed the linting on, as well as a status code for the linting process. Next it tells you what the issue with the code is, before it prints out the code line it

has an issue with. This makes it easy for you to go back into your source code and fix the issues JSHint reports.

Once you have successfully implemented linting into your build pipeline, you should ensure that your application source code passes the linting process before moving on.

Next up, we will look at how we can compile the applications Handlebars templates into a single JavaScript file.

11.2.5 Precompiling Handlebars Templates

The templates for the Montric application are all located inside the templates directory within separate .hbs files, and follows the general structure that de discussed in chapter 11.1. Separating your templates in this manner is great for the maintainability of your application, but in order for Ember.js to understand your templates, you need to precompile them into your application. Figure 11.13 shows the current stage in the assembly process.

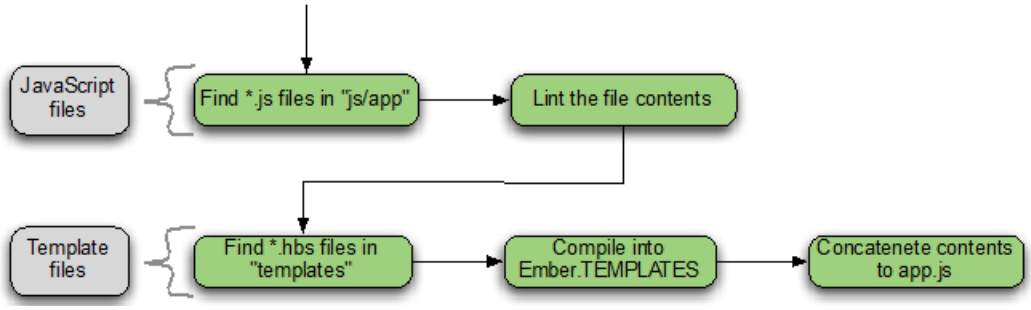


Figure 11.13 – Adding file minifying to the assembly process

Dan Gebhart has created an Ember.js template plugin for Grunt, which allows us to do just that. As with any other Grunt.js plugin, we need to install this plugin via the Node Package Manager, which can be done by executing `npm install grunt-ember-templates --save-dev`.Figure 11.14 shows the result of installing the grunt-ember-templates plugin.

```
Joachims-MacBook-Pro:webapp jhsmbp$ npm install grunt-ember-templates --save-dev
npm WARN package.json Montric@0.9.0 No README.md file found!
npm http GET https://registry.npmjs.org/grunt-ember-templates
npm http 200 https://registry.npmjs.org/grunt-ember-templates
npm http GET https://registry.npmjs.org/grunt-ember-templates/-/grunt-ember-templates-0.4.10.tgz
npm http 200 https://registry.npmjs.org/grunt-ember-templates/-/grunt-ember-templates-0.4.10.tgz
grunt-ember-templates@0.4.10 node_modules/grunt-ember-templates
Joachims-MacBook-Pro:webapp jhsmbp$
```

Figure 11.14 – Installing the grunt-ember-templates plugin

Now that we have installed the grunt-ember-templates plugin, we are ready to start adding precompiling our Handlebars templates. Create a new file inside the tasks directory named emberTemplates.js. Listing 11.9 shows the resulting package.json file.

Listing 11.9 – The package.json File After Installing the grunt-ember-templates plugin

```
{
  "name": "Montric",
  "version": "0.9.0",
  "devDependencies": {
    "grunt": "~0.4.1",
    "grunt-contrib-concat": "~0.3.0",
    "grunt-contrib-jshint": "~0.6.0",
    "grunt-ember-templates": "~0.4.10"                                    #A
  }
}
```

#A: Adding version 0.4.10 or greater of the grunt-ember-templates plugin

As you can see, NPM have added the grunt-ember-templates plugin to the package.json file for us, as we expected. Next we will create a file inside the tasks directory, named emberTemplates.js, with the contents of listing 11.10.

Listing 11.10 – The emberTemplates.js File

```
module.exports = {
     compile: {                                                           #A
         options: {
           templateName: function(sourceFile) {                           #B
             return sourceFile.replace(/templates\//, '');
           }
         },
         files: {
           "dist/templates.js": "templates/**/*.hbs"                      #C
         }
     }
};
```

#A: The grunt-ember-templates plugin is configured via the compule object
#B: The templateName function will return the template name for each of your templates
#C: The files directory tells the plugin where the templates are located, and where to store the final result

The emberTemplates.js file will look for any `.hbs` files inside any of the subdirectories of the `templates` directory. In addition to the contents of the `.hbs` files, we also need to tell Ember.js the name of the template we are compiling. We can deduce the template name from the path and filename of the `.hbs` file, but we need to strip off the "templates/" string from the path. We use the `templateName` function for this where we are simply replacing any occurrence of `"templates/"` with an empty string.

The final step in order to be able to precompile the applications template is to add the `grunt-ember-templates` plugin to Gruntfile.js. Listing 11.11 shows the updated file.

Listing 11.11 – The updated Gruntfile.js

```
function config(name) {
     return require('./tasks/' + name);
}

module.exports = function(grunt) {
     grunt.initConfig({
            pkg: grunt.file.readJSON('package.json'),
            concat: config('concat'),
            jshint: config('jshint'),
            emberTemplates: config('emberTemplates')                    #A
     });

     grunt.loadNpmTasks('grunt-contrib-concat');
     grunt.loadNpmTasks('grunt-contrib-jshint');
     grunt.loadNpmTasks('grunt-ember-templates');                       #B
     grunt.registerTask('default',
        ['jshint', 'emberTemplates', 'concat']);                        #C

};
```

#A: Adding the emberTemplates.js file to our build
#B: Loading the grunt-ember-templates plugin via NPM
#C: Adding the emberTemplates step to the default task

Above, we have simply added the emberTemplates.js file to Grunt.js' configuration, told Grunt.js to load the `grunt-ember-templates` plugin and registered `emberTemplates` as a step for the default build task.

Figure 11.15 shows the result of running grunt.

Figure 11.15 – Running grunt to compile our templates.

When grunt finishes, all of our templates will be compiled into a single file named "templates.js", located within the "dist" directory. But we do not really want to have to minify and assemble two files for our final deployable application. We can solve this easily by extending our configuration of the `concat` plugin. Listing 11.12 shows the updated concat.js file.

Listing 11.12 – Concatenating the Precompiled Templates Into Montric.js

```
module.exports = {
    options: {
        separator: '\n'
    },
    dist: {
        src: ['js/app/**/*.js', 'dist/templates.js'],                    #A
        dest: 'dist/<%= pkg.name %>.js'
    }
};
```

#A: Adding the dist/templates.js for concatenation

This will concatenate our templates in together with the rest of the JavaScript application, and our Montric.js file is almost ready for final deployment.

Next up, we will look at how we can minify the concatenated source code in order to create a single file that is ready for final deployment to production.

11.2.6 Minifying Your Source Code

Even though the above file could have been included into a production application, it is fairly verbose to send JavaScript files like this between the server and the user's browser. This is an artifact that we have written the JavaScript code in a way that makes it easy to develop and easy to maintain. The code itself is sprinkled with whitespace, inline comments and other artifacts that make the code more readable to humans. For a computer, though, this information is useless and redundant. Minifying is a procedure where we attempt to remove as much of this redundancy as possible without breaking any of the functionality of the application we are building. Figure 11.16 shows the current state of the assembly process.

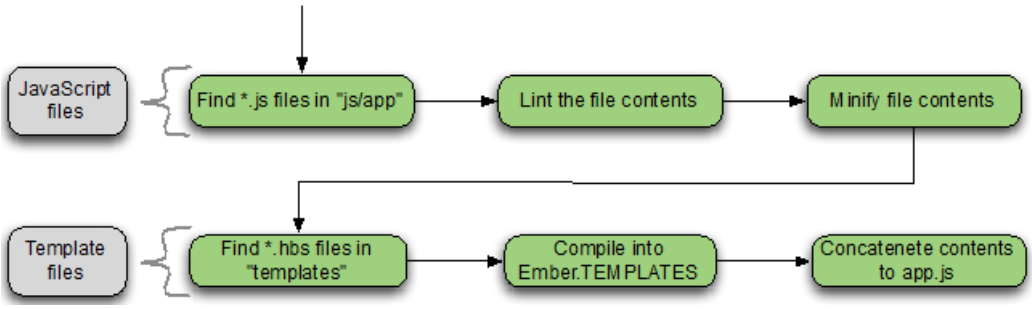


Figure 11.16 – Adding File minifying to the assembly process

Grunt uses an application called uglify in order to minify JavaScript code. Like with the other Grunt.js plugins, we start out by installing the grunt-contrib-uglify plugin with the command npm install grunt-contrib-uglify --save-dev, as shown in figure 11.17.

```
Joachims-MacBook-Pro:webapp jhsmbp$ npm install grunt-contrib-uglify --save-dev
npm WARN package.json Montric@0.9.0 No README.md file found!
npm http GET https://registry.npmjs.org/grunt-contrib-uglify
npm http 304 https://registry.npmjs.org/grunt-contrib-uglify
npm http GET https://registry.npmjs.org/grunt-lib-contrib
npm http GET https://registry.npmjs.org/uglify-js
npm http 304 https://registry.npmjs.org/uglify-js
npm http 304 https://registry.npmjs.org/grunt-lib-contrib
npm http GET https://registry.npmjs.org/zlib-browserify/0.0.1
npm http GET https://registry.npmjs.org/source-map
npm http GET https://registry.npmjs.org/async
npm http GET https://registry.npmjs.org/optimist
npm http 304 https://registry.npmjs.org/zlib-browserify/0.0.1
npm http 304 https://registry.npmjs.org/source-map
npm http 304 https://registry.npmjs.org/optimist
npm http 200 https://registry.npmjs.org/async
npm http GET https://registry.npmjs.org/wordwrap
npm http GET https://registry.npmjs.org/amdefine
npm http 304 https://registry.npmjs.org/amdefine
npm http 304 https://registry.npmjs.org/wordwrap
grunt-contrib-uglify@0.2.2 node_modules/grunt-contrib-uglify
├── grunt-lib-contrib@0.6.1 (zlib-browserify@0.0.1)
└── uglify-js@2.3.6 (async@0.2.9, source-map@0.1.25, optimist@0.3.7)
Joachims-MacBook-Pro:webapp jhsmbp$
```

Figure 11.17 – Installing the grunt-contrib-uglify plugin

As we might have come to expect by now, this will add a dependency to the plugin inside the package.json file, shown below in listing 11.13.

Listing 11.13 – The Updated package.json file with grunt-contrib-uglify Added

```
{
  "name": "Montric",
  "version": "0.9.0",
  "devDependencies": {
    "grunt": "~0.4.1",
    "grunt-contrib-concat": "~0.3.0",
    "grunt-contrib-jshint": "~0.6.0",
    "grunt-ember-templates": "~0.4.10",
    "grunt-contrib-uglify": "~0.2.2"                                    #A
  }
}
```

#A: grunt-contrib-uglify version 0.2.2 or newer added as a dependency

Next, we need to configure the uglify plugin. Create a file named uglify.js inside your "tasks" directory with the contents from listing 11.14.

Listing 11.14 – Configuring the grunt-contrib-uglify Plugin Inside tasks/uglify.js

```
module.exports = {
    options: {
        banner: '/*! <%= pkg.name %> <%= grunt.template.today("dd-mm-yyyy")
%> */\n'                                                                  #A
    },
    dist: {
        files: {
            'dist/<%= pkg.name %>.min.js': ['<%= concat.dist.dest %>']  #B
        }
    }
};
```

#A: Define the banner that will be added to the very top of the generated output
#B: Configure the input and output file that the plugin will work on

As you can see, we are defining two important pieces of information to the uglify plugin. The `banner` property includes a piece of text that will be added to the top of the final output file. The `files` property defines both the output file, as well as the input file that the plugin is working on. In our case we want to work on the destination file that the `concat` plugin created, and we want to create a new file inside the "dist" directory named Montric.min.js.

This is the entire configuration we need in order to minify our concatenated source file. But before we can build the minified file, we need to update the Gruntfile.js file with the contents of listing 11.15.

Listing 11.15 – The Updated Gruntfile.js

```
function config(name) {
    return require('./tasks/' + name);
}

module.exports = function(grunt) {
    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json'),
        concat: config('concat'),
        jshint: config('jshint'),
        emberTemplates: config('emberTemplates'),
        uglify: config('uglify')                                        #A
    });

    grunt.loadNpmTasks('grunt-contrib-concat');
    grunt.loadNpmTasks('grunt-contrib-jshint');
    grunt.loadNpmTasks('grunt-ember-templates');
    grunt.loadNpmTasks('grunt-contrib-uglify');                         #B
    grunt.registerTask('default',
         emberTemplates', 'concat', 'uglify']);          #C

};
```

#A: Adding the uglify.js file to our build
#B: Loading the grunt-contrib-uglify plugin via NPM
#C: Adding the uglify step to the default task

Above, we have simply added the uglify.js file to Grunt.js' configuration, told Grunt.js to load the `grunt-contrib-uglify` plugin and registered `uglify` as a step for the default build task.

Figure 11.18 shows the result of running grunt.

```
Joachims-MacBook-Pro:webapp jhsmbp$ grunt
Running "emberTemplates:compile" (emberTemplates) task
File "dist/templates.js" created.

Running "concat:dist" (concat) task
File "dist/Montric.js" created.

Running "uglify:dist" (uglify) task
File "dist/Montric.min.js" created.

Done, without errors.
Joachims-MacBook-Pro:webapp jhsmbp$
```

Figure 11.18 – Running grunt to build out final deployable application

At this point, if you look at inside the dist directory, you will find three files, shown below in Figure 11.19.

Name	Date Modified	Size
Montric.js	Today 6:25 PM	115 KB
Montric.min.js	Today 6:25 PM	76 KB
templates.js	Today 6:25 PM	4 KB

Figure 11.19 – The final contents of the dist directory.

There are more tasks that we could use Grunt.js for, that you will probably want to investigate further. The task of adding new steps to your build process follows the same outlines as described here, and it should be fairly straightforward for you to add new steps to your build process. Steps that you will probably want to investigate further are:

- Running your QUnit tests before the concat plugin
- Concatenating and minifying your CSS

The Grunt.js community is large and active which means that you should be able to find a plugin that fits your requirements. The Grunt.js project maintains a list of plugins, published at http://gruntjs.com/plugins/.

We have seen what Grunt.js have to offer, but lets quickly run through some of its advantages and drawbacks.

11.2.7 Advantages and Drawbacks of Grunt.js

Grunt.js is a rather new tool, which is both an advantage and a disadvantage. The advantage is that the tool is tailored to be able to build modern JavaScript applications, while it also have a modern plugin-centric structure. Because Grunt.js is built on Node.js, it is a native JavaScript build tool, meaning that it does have direct access to a powerful JavaScript interpreter. This means that it is easy to integrate Grunt.js with JavaScript specific tools. We take advantage of this fact when we are using the grunt-ember-template function, which uses the Node.js JavaScript interpreter to compile out .hbs templates into JavaScript functions.

However, there are a few disadvantages to Grunt.js as well. Most notably, because of Grunt.js' lenient approach to how its plugins can be configured, there are no standard ways to configure the operation of the myriad of plugins. If you browse back over the scripts we have created inside the "tasks" directory, you will quickly see that the different scripts have taken a different approach toward how they are configured. This means that you have to peruse the documentation for each of the plugins whenever you want to change or add configuration properties.

Another drawback, which I the main drawback for me, is the fact that it pollutes your project's directory with the file that the build system requires to operate. If you take a look at your projects directory, you will find that NPM have created a new directory called "node_modules". The contents of this folder is shown in Figure 11.20.

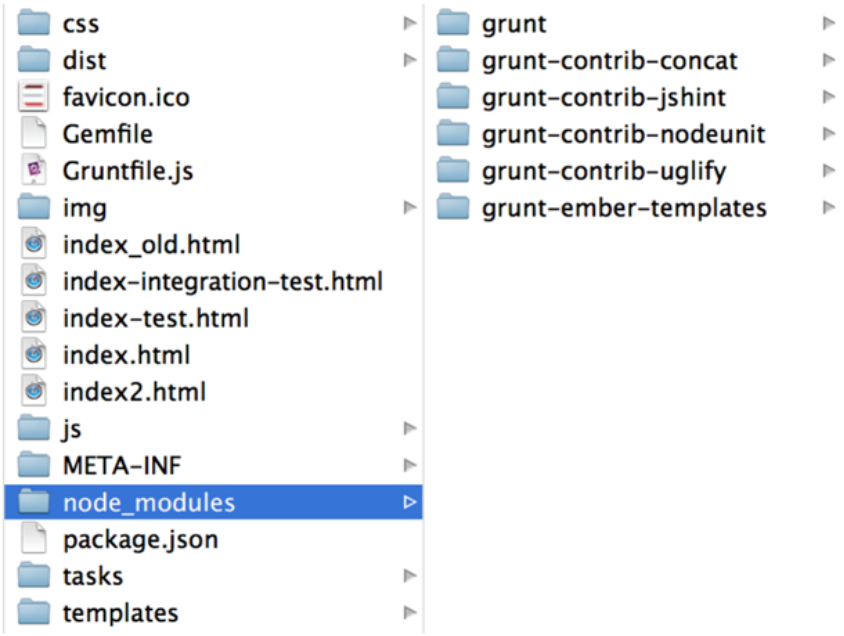


Figure 11.20 – The contents of the node_modules directory

As you can see, the "node_modules" directory contains any of the dependencies that we have added to our Grunt.js build, including grunt itself. In fact, this directory takes up 33MB and consists of over 2500 files. While I understand that NPM needs to keep track of which projects depend on which dependency and which version of that dependency it requires, I don't see why NPM needs to store this information inside of my project directory. Here, I believe that NPM and Grunt.js could have implemented a similar structure as Maven has with its dependency resolution, where Maven will keep a copy of any dependency that have ever been requested, along with every version of every dependency in a separate directory. By default Maven places these dependencies into a ~/.m2/repository directory, which it refers to when its building your application.

11.3 Summary

Throughout this chapter, we have looked at the JavaScript assembly and packaging pipeline. Even though JavaScript is an interpreted language, the fact that we need to deliver our application to our users via a browser over the HTTP protocol make these builds tools necessary. We utilize the build tools both in order to assist us while developing and maintaining our source code and during the assembly and packaging stages to minimize the amount of files as well as the amount of bits that we actually send across the wire between out servers and the users browser.

We have gone through the steps that are involved in packaging our application for production deployment, as well as seen an example of using Grunt.js examples of build tools. Grunt.js is a Node.js based build tool, which means that it is written in the same language as your Ember.js application, and has direct access to a JavaScript interpreter. We utilize this interpreter when compiling our template files into JavaScript functions.

Now that you have come to the end of the book, I wish to congratulate you on reaching the end of your journey into learning the basics, the pros and cons as well as some of the more advanced features of Ember.js. Additionally I would also like to wish you the best of luck with your continued exploration of Ember.js and your journey into writing truly ambitious web based applications that push the envelope on what is possible to achieve on the web. Looking back at how Ember.js have evolved since it started out as “SproutCore 2.0”, I am truly optimistic that this will be one of the most important frameworks for the web of the future!

9

Authentication through a third-party authentication system - Mozilla Persona

This chapter covers:

- Using third-party Authentication platforms for single-sign on with Ember.js
- Integrating and authentication with Mozilla Persona
- Re-authenticating via HTTP Cookies

Once you get to this point, you are familiar with the Ember.js architecture and you know how you can structure and build rich web applications using Ember.js. A concept that might not be as clear to you is how you will be able to provide a seamless sign in experience for your application users.

This chapter will look at implementing a single-sign-on solution by integrating Mozilla Persona. If you haven't heard about Mozilla Persona yet, you can think of it as a full featured authentication system that enables your application authenticate millions of users that are ready to use Mozilla Persona, via their own personal email account.

Even though this chapter uses Mozilla Persona as an example of a third party user authentication and authorization platform, the lessons learned throughout this chapter can be easily adapted to other authentication systems. Persona being based on JavaScript, makes integration with Ember.js applications a lot easier.

The core idea behind Mozilla Persona is the notion that your users online identity should solely belong to your users. Persona allows your users to associate a one or more email accounts to their identity, which allows them to sign in to a range of websites and applications using a single username (email) and password. In addition, because Mozilla Persona is created

by a non-profit organization that thrives on open source, you can gain your users trust by showing that you are relying on technology from a company that is trusted throughout the open source community.

There are many advantages of using a third-party authentication provider. The most significant of which is the fact that your applications won't have the responsibilities that are related to keeping usernames and passwords, while supplying secure off-site storage, password hashing and keeping your users information safe from hackers.

This chapter will go through

- Introducing Mozilla Persona
- Understanding how Montric implements user authentication and authorization using Mozilla Persona
- Integrating Mozilla Persona into the Montric Ember.js frontend application

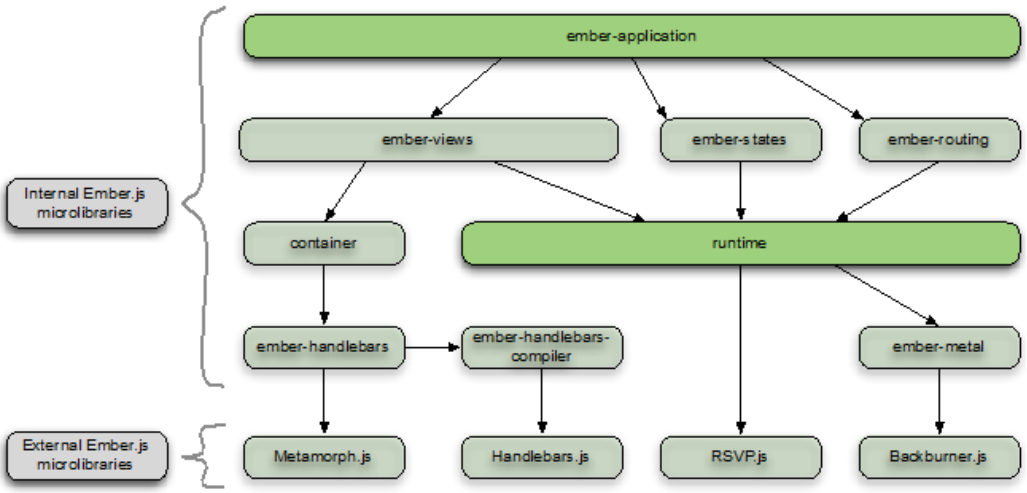


Figure 9.1 - The parts of Ember.js we will be working on in this chapter

So without much further ado, let's dive into what Mozilla Persona gives us in terms of features and functionality.

9.1 Integrating a Third-party Authentication system with Ember.js

As previously stated, Mozilla Persona is a full-featured third party authentication provider that will take care of the following aspects of user authentication and authorization:

- Email registration
- Email validation

- Storing passwords in a secure manner
- Re-issuing lost or forgotten passwords
- Any other issues that the user might have in relation to either their email address, account or password

With any third party web-based authentication mechanism there will be multiple ways for a user to authenticate towards your system and to log in. For our purposes, we will look the three different models which all needs to be implemented in order to provide the user with a smooth login and authentication mechanism:

- First time login and user registration
- Login via the third party authentication provider (Mozilla Persona)
- Signing users in via HTTP Cookies

9.1.1 First time login and user Registration

How you choose to implement the first login and user registration will be largely based on your systems requirements. Montric will accept any user, authenticated via Mozilla Persona, to register a new account with the system. This account will then be marked as “new”, which means that the user is authenticated with Mozilla Persona, but still unknown to Montric. At this point, the user will be notified that their account is awaiting activation. Due to the fact that Mozilla Persona will take care of authenticating that the current user does, in fact, belong to the given email address, Montric is able to simply redirect the user to the user-registration page whenever a new user authenticates towards the system. This, in turn, means that there aren’t any differences between a first login and any subsequent logins to Montric. This is a huge deal for the usability of your application as it means that there is a single point of entry into the application from the users point of view.

Montric, however, needs to check the account type of the logged in user and make sure that any account that has type “new” will be redirected to the awaiting activation page. Conceptually, this process is shown below in figure 9.2.

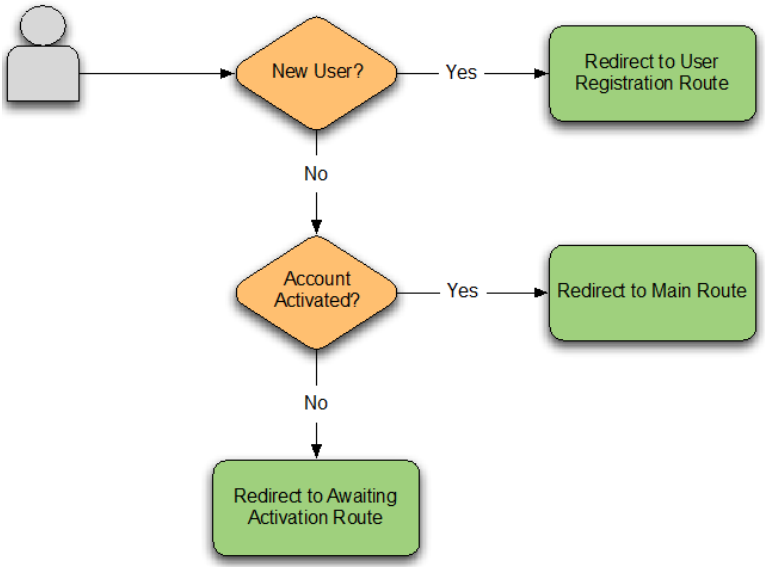


Figure 9.2 – Handling new Users in Montric

As you can see from the diagram above, the application needs to perform two checks. The first check will check if the user is entering the application for the first time. If so, the user will be redirected to the User Registration Route. If the user is already registered, the application will move on to the next check to see if the users account is activated. If the user's account is activated, the application will redirect the user to the applications Main route. If the user's account is not activated, Montric will redirect the user to the Awaiting Activation Route.

Before we move on to explain how this can be achieved, lets take a closer look at Montric Router definition, as shown in Listing 9.1 below.

Listing 9.1 – The Montric Router Definition

```
Montric.Router.map(function() {
    this.resource("main", {path: "/"}, function() {                    #A
        this.resource("login", {path: "/login"}, function() {          #B
            this.route("register", {path: "/register"});               #C
            this.route("selectAccount", {path: "/select_account"});
        });
        this.route("activation");                                      #D

        //The rest of the Montric router omitted
    });
});
```

#A: The "main" route is the main route encompassing all other routes for the Montric application
#B: The user is directed to the login page via the URL /login

#C: If the user logs in via Mozilla Persona, but an account is not set up, the user is directed to the register route, where (s)he will be able to register a new account
#D: If the user is authenticated, but the user's account is not activated, the user is directed to the activation route.

As you can see the Montric application has a top-level route named "main" that encloses all of the other routes within that application. If Montric is unable to log the user in either via a HTTP cookie, or via Mozilla Persona automatically, the user is redirected to the `login.index` route. The `login.index` route is responsible for showing the login-screen to the user. This login screen is slightly different than the login screens you are accustomed to, simply because the user will never actually enter neither their username nor their password directly into the Montric User Interface. Figure 9.3, below, shows the Montric login screen.

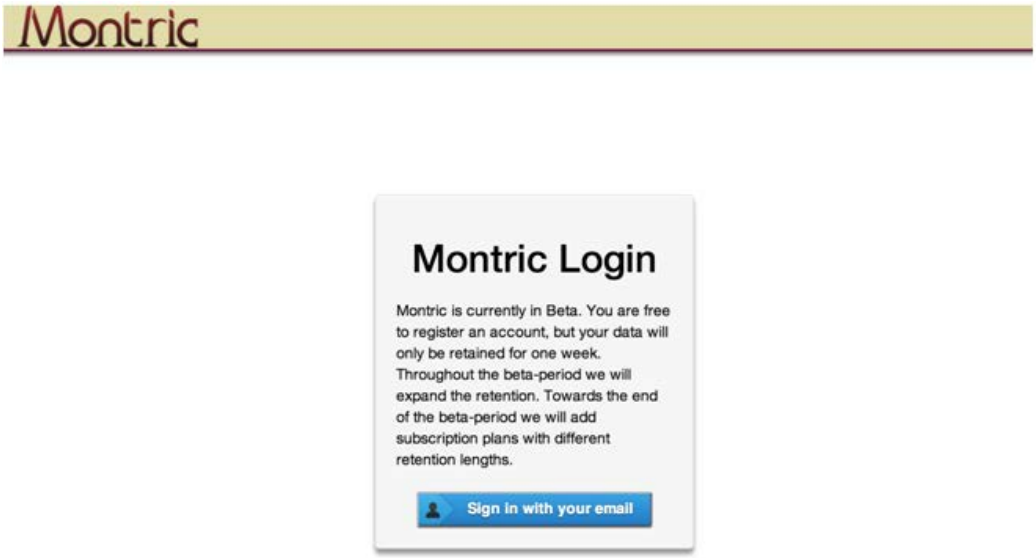


Figure 9.3 – The Montric Login Screen

Once the user is logged in Montric will verify the credentials that it received from Mozilla Persona. These credentials include the users email address as well as a timestamp, echoed back from the Mozilla Persona verifier, that shows when the Mozilla Persona session expires. If the user is not already registered as a Montric user, the user will be redirected to the `login.register` route.

Note that there are two possible paths that a user can take in order to end up on the `login.register` route. The most obvious way is to go through the `login.index` route, click on the "`Sign in with Mozilla Persona`" button, which will redirect them to the `login.register` route if the user is not already registered as a Montric user. If a user is already logged in to Mozilla Persona from a previous session, and navigates to the Montric

application (via any of Montrics valid routes), Montric will receive the user credentials from Mozilla Persona much in the same way as if the user had actually logged in via Mozilla Persona via the `login.index` route.

> **NOTE** Mozilla Persona will only send the user's credentials if the user have previously logged into Montric. This is a security measurement that ensures that Persona won't automatically reveal the users email address to websites that the user haven't actively logged in to before.

This is an important distinction between traditional authentication solutions and third party single-sign-on solutions like Mozilla Persona.

Figure 9.4, below, shows the login.register screen.

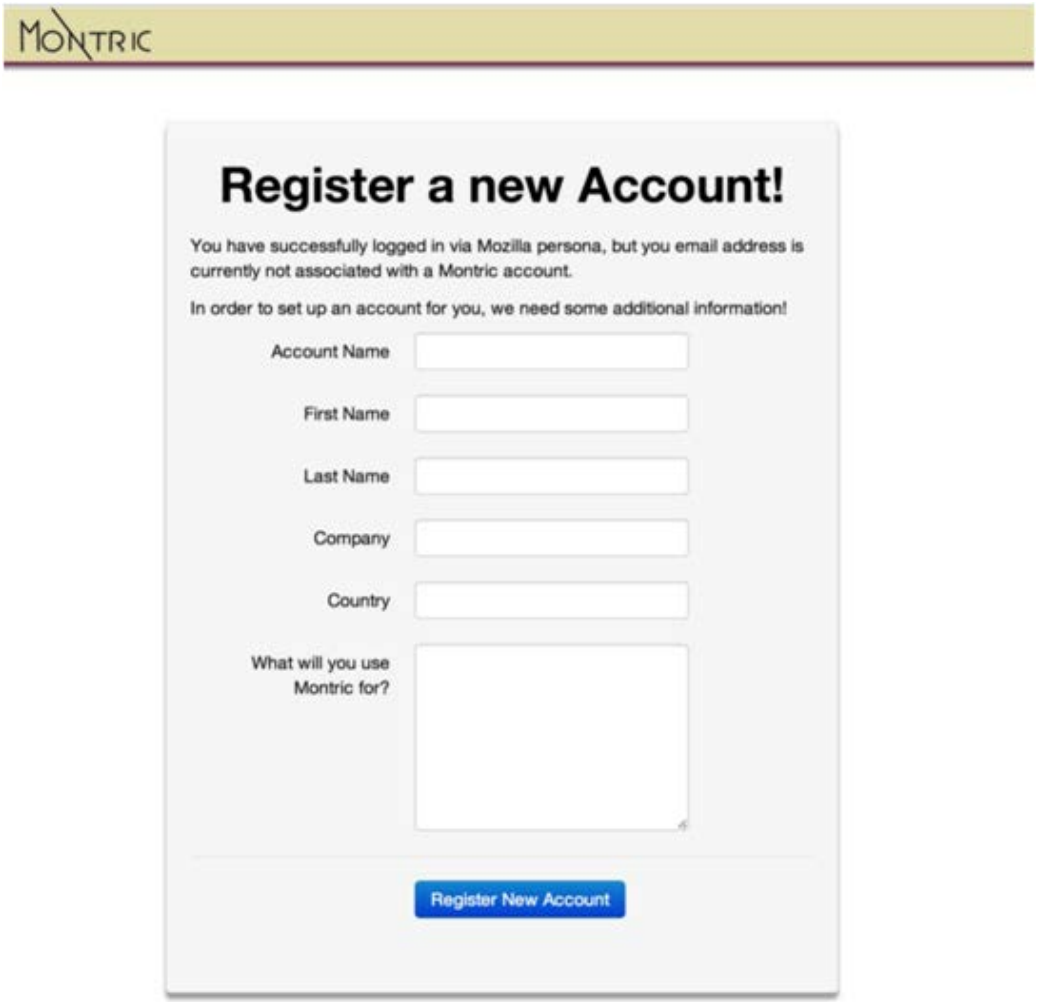


Figure 9.4 – The Montric Register User Screen

The registration form is quite simple. Mozilla Persona have already authenticated that the users email address is indeed correct, and that the Mozilla Persona session have not expired. Through the `login.register` route, the user is presented with simple input fields that lets the user set up an account name for the new account, as well as to fill in their full name, company and country information. Once the `"Register New Account"` button is clicked, Montric will create a new account, and associate the current user as an administrator of that account.

However, because Montric is currently in Beta, account creation is limited. This means that the newly created account will be locked until a Montric-administrator activated the account by changing the account type from "new" to "beta". The `main.activation` route is shown below, in figure 9.5.

Figure 9.5 – The Awaiting Activation Screen

As you can see, there isn't much going on in the `main.activation` route. Montric is simply explaining the user that their account requires validation and why this is the case. Because the user will not be able to perform any actions inside the Montric application, the application does not provide any links or buttons that the user can click in order to navigate out of the `main.activation` route.

Now that we have seen what the Montric authentication flow looks like, this would be a good time to delve into the code and see how authentication and authorization is implemented in the Montric application.

9.1.2 Logging Into Montric via the Third-party authentication provider

As we mentioned above, there are multiple ways in which a user can authenticate itself against Montric. We will start by looking at how the user is authenticated via Mozilla Persona, before looking at how the user can be automatically logged in via a HTTP cookie.

Logging in via Mozilla Persona is a three-step process, and involved the user, the Montric Ember.js Frontend application, the Montric Backend application as well as Mozilla Persona.

Figure 9.6 shows the process of authenticating a user via Mozilla Persona

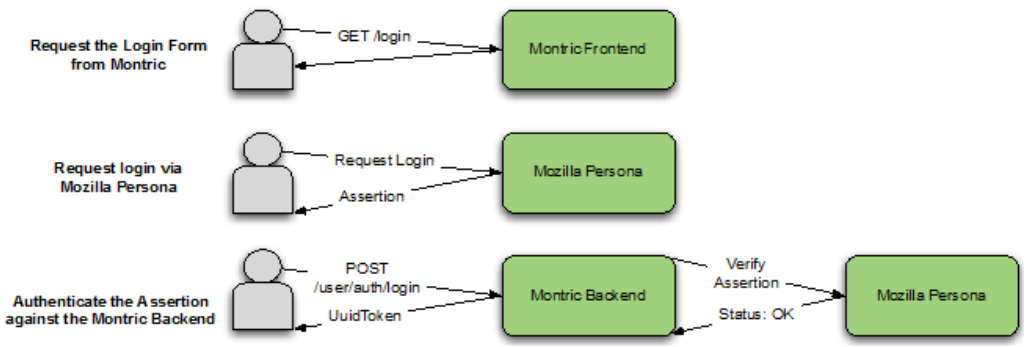


Figure 9.6 – Authenticating a User via Mozilla Persona

Before we can get started, the very first step we need to complete is to include the Mozilla Persona JavaScript file to our index.html file, as shown below in listing 9.2.

Listing 9.2 – Including the include.js Mozilla Persona JavaScript library

```
<script src="https://login.persona.org/include.js"></script>          #A
```

#A: Including the include.js script from login.persona.org

This script will most likely be the only script (in addition to you analytics script) that you will include via HTTP/HTTPS. Including this script will bootstrap your Mozilla Persona integration and make its methods available for use within your own application. The Mozilla Persona team plans to build Persona functionality right into browsers in the future. The include.js file can therefore be considered a polyfill that will ensure that Mozilla Persona works with browsers that lack built-in Persona functionality.

IMPLEMENTING THE MOZILLA PERSONA LOGIN

The next step involves watching for Mozilla Persona logins and logouts. This is simply done by calling `navigator.id.watch()`, into which you will provide Mozilla Persona with three critical pieces of information:

- The email address of the currently logged in user in your application
- An `onlogin`-callback that will be invoked when a user is authenticated through Mozilla Persona.
- An `onlogout`-callback that will be invoked when a user signs out through Mozilla Persona

The Montric has a single controller that takes care of the bookkeeping required in order to sign in and sign out a single user, called `Montric.UserController`. In order to call Mozilla Persona early enough in the Montric application start-up sequence, we will issue a call to `navigator.id.watch()` inside the `UserController`'s init function.

I have split the contents of the `Montric.UserController.init()` function into three separate listings in order for use to go through them in more detail separately. The first listing shows the structure of the init-function, while the next two goes through the `onlogin` and `onlogout` callbacks.

Listing 9.3 – The Montric.UserController.init() function

```
Montric.UserController = Ember.ObjectController.extend({
    init: function() {
        this._super();                                                    #A
        this.set('content', Ember.Object.create());                       #B
        var controller = this;                                            #C
        navigator.id.watch({                                              #D
            loggedInUser: null,                                           #E
            onlogin: function(assertion) {},                              #F
            onlogout: function() { }                                      #G
        });
    }
});
```

#A: Remember to call this._super() whenever you override Ember.js specific functions
#B: Initializing the controllers content property with a new Ember.js Object
#C: Creating a local variable to hold on to this instance of the UserController, which we will use inside the navigator.id.watch() function
#D: Calling navigator.id.watch() in order to watch for Mozilla Persona logins and logouts.
#E: Providing a callback that Mozilla Persona will invoke upon successful sign in
#F: Providing a callback that Mozilla Persona will invoke upon sign out

The listing above shows the structure of the `init`-function. It starts out by initiating a call to `this._super()` in order to ensure that Ember.js is able to set up everything it needs for the controller to operate as intended. Next it initializes the controller's content property to an empty Ember.js `Object`. We will utilize this object as a temporary storage object before we are able to fetch a real `Montric.User` object from the Montric Backend.

The final piece of code the `init` function needs to call is the `navigator.id.watch()` function. Inside this function we state that the current `loggedInUser` is `null` (no logged in user). In addition we provide it with the `onlogin` and `onlogout` callbacks.

Listing 9.4, below, shows the contents of the `onlogin` callback.

Listing 9.4 – The onlogin callback

```
onlogin: function(assertion) {                                            #A
    Montric.set('isLoggingIn', true);                                     #B
    $.ajax({                                                              #C
        type: 'POST',
        url: '/user/auth/login',
        data: {assertion: assertion},
        success: function(res, status, xhr) {
            if (res.uuidToken) {                                          #D
               controller.createCookie("uuidToken", res.uuidToken, 1);
            }
```

```
            if (res.registered === true) {                              #E
                //login user
                controller.set('content',
                    Montric.User.find(res.uuidToken));
            } else {                                                    #F
                controller.set('newUuidToken', res.uuidToken);
                controller.transitionToRoute('login.register');
            }
        },
        error: function(xhr, status, err) {                             #G
            console.log("error: " + status + ": " + err);
            navigator.id.logout();                                      #H
        }
    });
}
```

#A: The onlogin callback is provided with an assertion that we need to pass down to the Montric sBackend
#B: While we are logging in, set the isLoggingIn property to true
#C: Issue an HTTP POST request to the Montric Backend with the assertion
#D: If the response contains a UUID Token, create a cookie uuidToken with it
#E If the response indicated that the user is registered, fetch the user belonging to the UUID Token
#F: If the user is not registered, transition the user to the login.register route
#G: Provide error handling if HTTP POST to Montric Backend fails
#H: If authentication fails for any reason, ensure that the user is actually logged out

Mozilla Persona will pass in an assertion to the `onlogin` callback. You can think of the assertion as an encoded single-use, single-site password. This assertion contains the information that the Montric backend will use in order to verify that the response it received from Mozilla Persona is in-fact real. The assertion is also a privacy implementation that Mozilla Persona uses in order to keep the users login credentials out of the browser-session, as Mozilla Persona will only supply the users email address once it receives this assertion in return from the server-side implementation of your web application.

Because we want the Montric UI to tell the user that we are attempting to log the user in, the first thing we set inside the `onlogin` callback is the `isLoggingIn`, which we set to true. Next, we need to issue a HTTP POST request to the backend with the assertion that we got from Mozilla Persona.

One the Montric Backend responds to the HTTP POST, it will respond with both a `uuidToken` and a registered property. We will use the `uuidToken` to set a cookie that we will use for cookie-based sign in later in this chapter. This step is strictly not necessary, as Mozilla Persona will keep the user session active as long as the user is logged into Mozilla Pesona. However, re-authenticating the user via a session-based cookie will be faster that going through Mozilla Persona. If the backend indicates that the user is already registered with an account, we will fetch the current logged in user via a call to `Montric.User.find(uuid)`. If the user is not associated with an account, we will simply redirect the user to the `login.register` route.

VERIFYING THE ASSERTION WITH MOZILLA PERSONA

The backend will verify the contents of the assertion by passing it along to https://verifier.login.persona.org/verify, along with the URL that Montric is available from. Listing 9.5 below shows the contents of the message sent to the verify-endpoint.

Listing 9.5 – The message sent to the Mozllla Persona verify endpoint

```
assertion=<ASSERTION>&audience=https://live.montric.no:443
```

There are two important things to note here. Never authenticate the assertion on the client-side of your application because that might expose your users credentials to malicious third parties. Also the audience should be defined by the server-side application, and not by the client side or by the HTTP header information that you receive from Mozilla Persona.

If the Mozilla Persona is able to verify both the assertion and the audience, it will respond with a JSON hash containing the users credentials. Listing 9.6 shows the contents of a successful login attempt.

Listing 9.6 – The JSON Respnse from Mozilla Persona from a successful login attempt

```
{
    "status": "okay",                                          #A
    "email": "joachim@haagen-software.no",                     #B
    "audience": "http://live.montric.no:443",                  #C
    "expires": 1369060978610,                                  #D
    "issuer": "login.persona.org"                              #E
}
```

#A: If the assertion is verified, the retunred status is “okay”
#B: Once the assertion is verified, we are supplied with the email address of the logged in user
#C: The audience that we passed in is returned
#D: Every Mozilla Persona credential is valid for a limited amount of time. The expires property tells Montric how long the login is valid for
#E: The original issuer of the assertion, the domain that originally vouched for the validity of the users email address.

Once we have verified that the user is, in fact, authenticated by Mozilla Persona, and we have the email address of the logged in user, the Montric backend will associate the users email address with a unique `UUID Token`. We will use this `UUID Token` in order to be able to re-log the user in via a HTTP Cookie. The Response from the Montric Backend is shown below in listing 9.7.

Listin 9.7 – The JSON Response from the Montric Backend

```
{
    "uuidToken": "99d21a30-1564-4863-a368-0a890f59532e",       #A
    "registered": false                                        #B
}
```

#A: The UUID Token associated with the logged in user
#B: A Boolean registered property indicating if the user is already registered or not.

The above JSON is straightforward. The `UUID Token` is generated on the server-side once the user is authenticated and authorized as a new user. Montric will store this `UUID Token` as the unique identifier for this users email address. The registered property simply tells the Montric Frontend application whether or not the user is already registered. Montric will use this information in order to redirect the user to the `login.register` route if the user needs to register a new account.

Now that we have seen how we can authenticate users via Mozilla Persona, lets go ahead and see how we can leverage the HTTP Cookies in order to provide re-login directly via Cookies. The advantage of adding this approach in addition to Mozilla Persona is to reduce the amount of HTTP request that will go between the Ember.js application and your backend server, while also removing unnecessary HTTP requests to Mozilla Persona if you can already identify the user.

9.2 Signing Users in via HTTP Cookies

As we have seen above, we are setting a cookie named `uuidToken` with a text-string that we can use later on to authenticate a user without having to call upon Mozilla Persona. We do this in the `Montric.UserController.init` function by storing the `uuidToken` that we receive in the response from a successful login to the Montric Backend.

Logging in via HTTP `Cookies`, while being the fastest option, is also a lot simpler than it is via Mozilla Persona. Therefore, once the user have been authenticated via Mozilla Persona, we want to ensure that as many subsequent logins as possible will be made via the HTTP Cookie. Figure 9.7 shows the HTTP `Cookie` log in process.

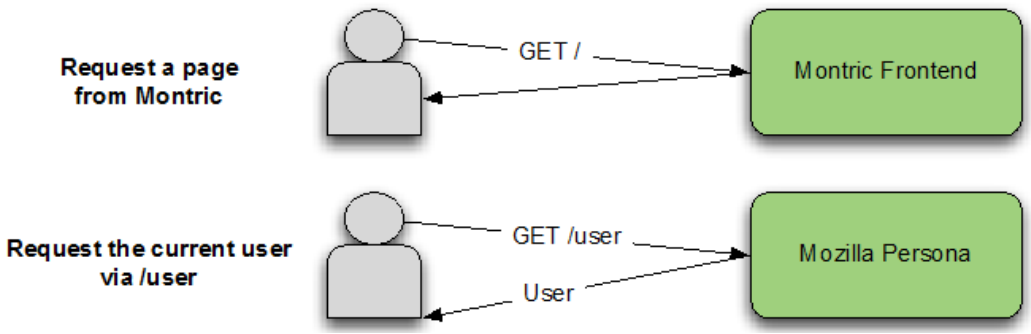


Figure 9.7 – The HTTP Cookie Login Process

In order to create both sign in and sign out functionality through the use of cookies, we need to provide our Montric Frontend application with the ability to

- Create Cookies
- Read Cookies

- Delete Cookies

In the Montric application we have chosen to implement these features as three separate functions on the `Montric.UserController`, as this is the controller that we have elected to be responsible for the book keeping related to the logged in user in Montric. This is an appropriate place to put these functions, as we want to keep all of our user-related functionality inside of the single `Montric.UserController` class.

Listing 9.8, below, shows the contents of these three functions.

Figure 9.8 – Working with Cookies

```
createCookie:function (name, value, days) {                              #A
    if (days) {
        var date = new Date();
        date.setTime(date.getTime()+(days*24*60*60*1000));
        var expires = "; expires="+date.toGMTString();
    }
    else var expires = "";
    document.cookie = name+"="+value+expires+"; path=/";
},

readCookie:function (name) {                                              #B
    var nameEQ = name + "=";
    var ca = document.cookie.split(';');
    for (var i = 0; i < ca.length; i++) {
        var c = ca[i];
        while (c.charAt(0) == ' ') c = c.substring(1, c.length);
        if (c.indexOf(nameEQ) == 0)
            return c.substring(nameEQ.length, c.length);
    }
    return null;
},

eraseCookie:function (name) {                                             #C
    this.createCookie(name, "", -1);
}
```

#A: The createCookie function will create a new cookie that exires in x-amount of days
#B: Read the cookie value from the cookie with the given name
#C: Delete the cookie with the given name by setting its contents to an empty st

We need to call the `createCookie` from within the `navigator.id.watch()` function when the user is successfully authenticated from the Montric Backend (see listing 9.4). When the user signs out of the system, we can simply sign the user out by invoking the `eraseCookie`-function with the parameter "`uuidToken`".

Because we are both providing the frontend application with the ability to sign in users based on both a cookie value and via Mozilla Persona, we need to ensure that the user we only initialize the Mozilla Persona functionality if the Montric backend is unable to authenticate the user via the supplied HTTP Cookie. Listing 9.9 shows the updated `Montric.UserController.init` function.

Listing 9.9 – Updating Montric.UserController.init with Cookie support

```
Montric.UserController = Ember.ObjectController.extend({
    needs: ['application', 'account'],

    init: function() {
        this._super();
        this.set('content', Ember.Object.create());
        var controller = this;
        var cookieUser = Montric.get('cookieUser');
        if (cookieUser == null) {                                    #B
            navigator.id.watch({
                loggedInUser: null,
                onlogin: function(assertion) { },
                onlogout: function() { }
            });
        } else {                                                     #B
            this.set('content', Montric.get('cookieUser'));
        }
    }
});
```

#A: If Montric.get('cookieUser') is falsy, initialize Mozilla Persona as before
#B: Update the content-property of the UserController with Montric.get('cookieUser')

Compared with the previous implementation of the `UserController's init` function, we have added a single check to see if `Montric.get('cookieUser')` has a value. If it does, we can simply update the controllers content-property with the contents of `Montric.get('cookieUser')`.

You might be wondering where the `cookieUser` comes from. Because cookies are available from the browsers, and will be part of each and every HTTP Request made from the client application to the server, we are able to authenticate the user before the Montric application is initialized.

We are able to perform this authentication early, due to the fact that Ember.js lets us halt application initialization early by calling `App.deferReadiness()`. Inside Montric's `app.js` file, we have included code that fetches the current user based on the `uuidToken` cookie value. This code is shown below in listing 9.10.

Listing 9.10 – Halting and resuming Application Initialization

```
Montric.deferReadiness();                                            #A

$.getJSON("/user", function(data) {                                  #B
    if (data["user"] && data["user"].userRole != null) {             #C
        var cookieUser = Montric.User.create();
        cookieUser.setProperties(data["user"]);
        Montric.set('cookieUser', cookieUser);
    } else {
        Montric.set('cookieUser', null);                             #D
    }

    Montric.advanceReadiness();                                      #E
```

```
});
```

#A: Halt the application initialization by calling Montric.deferReadiness().
#B: Fetch the current user by issuing a HTTP GET to /user
#C: If the server responds with a user object, create a new Montric.User object as set it as Montric.cookieUser
#D: If the server does not respond with a user, set cookieUser to null

We halt the initialization of Montric by calling `Montric.deferReadiness()`. This will tell Ember.js to wait with initializing controllers and the router until a call to `Montric.advanceReadiness()` is made. This provides us with a window inside which we are able to fetch any data that our application either absolutely need (like the authenticated user), or data that we can use to optimize our application (data that is large or frequently used that we want to pre-fetch).

In our case, we are simply fetching the current user by issuing a XHR request to the /user URL. If the Montric backend responds with a user object that has an associated `userRole`, we assume that the backend have authenticated the `uuidToken` cookie. We can the simply create a new `Montric.User` object and set the properties that we received from the server. Once the new `Montric.User` object is initialized, we can call `Montric.set('cookieUser', cookieUser)`. This will tell the `Montric.UserController` that the user was authenticated via HTTP Cookies and that we do not need to involve Mozilla Persona at all for the current session.

A note about security

Even though Mozilla Persona is a secure authentication provider, all authentication providers are subject to vulnerabilities that you should be aware of. Most of the vulnerabilities can be avoided by simply following simple guidelines, while others are harder to avoid.

The Mozilla Identity Team (the people behind Mozilla persona) has created a set of guidelines that you should follow when implementing a Mozilla Persona based authentication application.

You should at least read through the guidelines given in both the Best Practices document (https://developer.mozilla.org/en-US/docs/Mozilla/Persona/Security_Considerations) and in The Implementor's Guide (https://developer.mozilla.org/en-US/docs/Mozilla/Persona/The_implementor_s_guide). Together these outline the vulnerabilities that you need to consider when implementing your authentication solution, while also giving you a walkthrough on how to avoid exposing your users.

9.3 Summary

This chapter has gone through two of the possible ways in which you can implement authentication and authorization within your own Ember.js application. Through the use of

Mozilla Persona, we have shown how a third-party JavaScript based authentication solution can be integrated into an Ember.js application. User authentication is always a tougher implementation that you might initially estimate it as, because there are many edge cases to consider when implementing a proper authentication mechanism in your application.

The advantages of using a third-party authentication provider are many, but the most significant advantages are the fact that your applications wont have the responsibilities that are related to keeping usernames and passwords. In addition to secure storage, password hashing and keeping your users information safe from hackers, you get all the features related to account creation, editing, and lost and forgotten passwords for free. Mozilla Persona manages to pack all of these features into a single neat package that is fairly easy to integrate into your own applications.

Because Mozilla Persona is not based on any social network or any other application platform that you might use, Person users does not have to worry about their personal information leaking outside of their authentication mechanism. The only piece of personal information that Mozilla Persona will ever reveal to your sites is your email addresses.

This chapter concludes part two of the book, before we move on to packaging, deployment and building Cloud based applications with Ember.js.