



Community Experience Distilled

Mastering Ember.js

Build fast, scalable, dynamic, and ambitious single-page web applications by mastering Ember.js

Mitchel Kelonye

[PACKT] open source*
PUBLISHING community experience distilled

Mastering Ember.js

Table of Contents

[Mastering Ember.js](#)

[Credits](#)

[About the Author](#)

[About the Reviewers](#)

[www.PacktPub.com](#)

[Support files, eBooks, discount offers, and more](#)

[Why subscribe?](#)

[Free access for Packt account holders](#)

[Preface](#)

[What this book covers](#)

[What you need for this book](#)

[Who this book is for](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Introduction to Ember.js](#)

[The origin of Ember.js](#)

[Downloading Ember.js](#)

[Creating your first application](#)

[Router](#)

[Route](#)

[Controller](#)

[View](#)

[Template](#)

[Component](#)

[Initializing the application](#)

[Embedding Ember.js applications](#)

[Summary](#)

[2. Understanding Ember.js Objects and Mixins](#)

[Creating objects in Ember.js](#)

- [Accessing object properties](#)
- [Defining class instance methods](#)
- [Defining computed properties](#)
- [Defining property observers](#)
- [Creating property bindings](#)
- [Using mixins](#)
- [Reopening classes and instances](#)
- [Event subscription](#)
- [Summary](#)

[3. Routing and State Management](#)

- [Creating the application's router](#)
- [Mapping URLs to routes](#)
- [Nesting resources](#)
- [Understanding the state transition cycle](#)
- [Configuring the router](#)
- [Specifying a route's path](#)
- [Defining route and resource handlers](#)
- [Specifying a route's model](#)
- [Serializing resources](#)
- [Asynchronous routing](#)
- [Configuring a route's controller](#)
- [Rendering the route's template](#)
- [Redirecting state](#)
- [Catching routing errors](#)
- [Summary](#)

[4. Writing Application Templates](#)

- [Registering templates](#)
- [Inserting templates](#)
- [Writing out templates](#)
- [Expressing variables](#)
- [Writing bound and unbound expressions](#)
- [Adding comments in templates](#)
- [Writing conditionals](#)
- [Switching contexts](#)
- [Rendering enumerable data](#)
- [Writing template bindings](#)
- [Defining route links](#)

[Registering DOM element event listeners](#)

[Writing form inputs](#)

[Extending templates](#)

[Defining custom helpers](#)

[Creating subexpressions](#)

[Summary](#)

[5. Controllers](#)

[Defining controllers](#)

[Providing controllers with models](#)

[Rendering dynamic data from controllers](#)

[Properties](#)

[Computed properties](#)

[Observables](#)

[Object and array controllers](#)

[An object controller](#)

[An array controller](#)

[addObject\(object\)](#)

[pushObject\(object\)](#)

[removeObject\(object\)](#)

[addObjects\(objects\), pushObjects\(objects\), and removeObjects\(objects\)](#)

[contains\(object\)](#)

[compact\(\)](#)

[every\(callback\)](#)

[filter\(object\)](#)

[filterBy\(property\)](#)

[find\(callback\)](#)

[findBy\(key, value\)](#)

[insertAt\(index, object\), objectAt\(index\), and removeAt\(index, length\)](#)

[map\(callback\)](#)

[mapBy\(property\)](#)

[forEach\(function\)](#)

[uniq\(\)](#)

[sortProperties and sortAscending](#)

[Handling event actions](#)

[Specifying controller dependencies](#)

[State transitions in controllers](#)

[Summary](#)

6. Views and Event Management

Defining views

Accessing a view's controller

Specifying a view's template

Specifying a view's element tag

Updating a view's element class attribute

Updating other views' element attributes

Inserting views into DOM

Inserting views into templates

Specifying view layouts

Registering event handlers in views

Emitting actions from views

Using built-in views (components)

Textfields

Textareas

Select menus

Checkboxes

The container view

Integrating with third-party DOM manipulation libraries

Summary

7. Components

Understanding components

Defining a component

Differentiating components from views

Passing properties to components

Customizing a component's element tag

Customizing a component's element class

Customizing a component's element attributes

Managing events in components

Defining component actions

Interfacing a component with the rest of the application

Components as layouts

Summary

8. Data Persistence through REST

Making Ajax requests

Understanding Ember-data

Ember-data namespace

[Creating a data store](#)

[Defining models](#)

[Declaring relations](#)

[One to one](#)

[Finding records](#)

[Defining a store's adapter](#)

[Creating REST APIs](#)

[Customizing a store's serializer](#)

[Creating custom transformations](#)

[Summary](#)

[9. Logging, Debugging, and Error Management](#)

[Logging and debugging](#)

[Objects](#)

[Router and routes](#)

[Templates](#)

[Controllers](#)

[Views](#)

[Using the Ember.js inspector](#)

[Client-side tracing](#)

[Error management](#)

[Summary](#)

[10. Testing Your Application](#)

[Writing tests](#)

[Asynchronous test helpers](#)

[Synchronous test helpers](#)

[Wait helpers](#)

[Writing unit tests](#)

[Testing computed properties](#)

[Testing method calls](#)

[Testing observers](#)

[Writing integration tests](#)

[Summary](#)

[11. Building Real-time Applications](#)

[Setting up Socket.io](#)

[Connecting the user](#)

[Summary](#)

[12. Modularizing Your Project](#)

[Installing the Component build tool](#)

[Code organization](#)

[Installing components](#)

[Building components](#)

[Loading the built files](#)

[Game logic](#)

[Serving images and fonts](#)

[Summary](#)

[Index](#)

Mastering Ember.js

Mastering Ember.js

Copyright © 2014 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: October 2014

Production reference: 1131014

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78398-198-4

www.packtpub.com

Cover image by Rajnish Jha (<rajnishlalitjha@gmail.com>)

Credits

Author

Mitchel Kelonye

Reviewers

John Christopher

Chad Hietala

James A Rosen

Commissioning Editor

Amarabha Banerjee

Acquisition Editor

Owen Roberts

Content Development Editor

Arun Nadar

Technical Editor

Manan Badani

Copy Editors

Janbal Dharmaraj

Sayanee Mukherjee

Alfida Paiva

Project Coordinator

Priyanka Goel

Proofreaders

Simran Bhogal

Stephen Copestake

Faye Coulman

Indexers

Hemangini Bari

Mariammal Chettiyar

Graphics

Abhinash Sahu

Production Coordinator

Nitesh Thakur

Cover Work

Nitesh Thakur

About the Author

Mitchel Kelonye is a software developer from Nairobi who is deeply attached to open source web and mobile technologies. He helps clients all over the world build amazing mobile and web experiences using great tools such as Ember.js. As a self-taught engineer, he was lucky enough to be involved in the use of Ember.js since its inception. Mitchel is currently completing an undergraduate course in Information Technology and can be followed on Twitter @kelonye.

About the Reviewers

John Christopher has more than 15 years of technical experience ranging from military weather applications to online auction platforms. He is currently the VP of Engineering at C2FO. In previous years, John was a weather forecaster in the United States Navy, and he still enjoys talking weather.

I would like to thank my wife, Debbie, and my four wonderful kids (Ian, Ethan, Evelyn, and Aiden) for letting me take the time to work on this project.

Chad Hietala is a self-taught frontend engineer and has been developing software for over 8 years. During this time, he has architected and developed several large JavaScript applications using a variety of frontend technologies such as Backbone, Angular, and Ember.

Chad is currently employed at LinkedIn where he works on frontend infrastructure and other cross-functional projects.

James A Rosen is a senior user happiness engineer at Zendesk. He writes Ruby and JavaScript and is currently working on improving performance, scalability, and developer happiness on large-scale distributed web applications. He holds a BS degree in Computer Science and Music from Washington University in St. Louis and an MS degree in Information Security Policy and Management from Carnegie Mellon University. He has written for the Zendesk Developers blog and contributed to technical books, including editing *Understanding the Four Rules of Simple Design*, Corey Haines.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Preface

Mastering Ember.js is a must-read for any web developer who wishes to start writing ambitious web applications that rival native web apps. It's packed with practical samples that show how easy it is to architect these applications. This book was inspired by the need for an Ember.js resource that explained Ember.js better using real-world examples.

What this book covers

[Chapter 1](#), *Introduction to Ember.js*, introduces the key concepts of Ember.js.

[Chapter 2](#), *Understanding Ember.js Objects and Mixins*, discusses Ember.js primitive objects that are the foundation of the other higher-level concepts.

[Chapter 3](#), *Routing and State Management*, details how a browser's location-based state management is accomplished in Ember.js apps.

[Chapter 4](#), *Writing Application Templates*, discusses how templates in an Ember.js application are defined and written.

[Chapter 5](#), *Controllers*, explains how controllers function as proxies to route models.

[Chapter 6](#), *Views and Event Management*, discusses how tight DOM-specific logic can be implemented in the view layer.

[Chapter 7](#), *Components*, introduces the reader to Ember.js components that enable authors to create custom and modular HTML elements.

[Chapter 8](#), *Data Persistence through REST*, discusses the different ways in which Ember.js applications can connect to remote data sources through REST.

[Chapter 9](#), *Logging, Debugging, and Error Management*, discusses how bugs within Ember.js applications can be traced and monitored.

[Chapter 10](#), *Testing Your Application*, outlines the different unit- and integration-testing techniques that can be employed to guarantee the development of stable applications.

[Chapter 11](#), *Building Real-time Applications*, explains how real-time web technologies can be integrated into Ember.js applications.

[Chapter 12](#), *Modularizing Your Project*, provides an in-depth explanation of how to better organize large-scale Ember.js applications.

What you need for this book

For the major part, this book requires you to have a modern browser installed. In addition, some sections will require you to install Node.js in order to run the provided server programs.

Who this book is for

This book targets both beginners and intermediary-level Ember.js users. It's packed with many practical samples that are well suited for any mid-level JavaScript developer who would like to start creating ambitious web applications.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `app.js` file contains all our application code but later on we may separate application concerns into more files when the application grows."

A block of code is set as follows:

```
<script src="js/libs/jquery-1.10.2.js"></script>
<script src="js/libs/handlebars-1.1.2.js"></script>
<script src="js/libs/ember-1.7.0.js"></script>
<script src="js/app.js"></script>
```

Any command-line input or output is written as follows:

```
npm install
node server
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "If an application uses Ember.js data, the **Data** tab will display all the loaded models."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to [<feedback@packtpub.com>](mailto:feedback@packtpub.com), and mention the book title via the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Introduction to Ember.js

This chapter will introduce **Ember.js**, including its origin, release cycle, and its key elements. It will focus on describing the different functions that can be undertaken once an application is created. Therefore, a better understanding of the following will be gained at the end of the chapter:

- Ember.js's origin
- Downloading Ember.js and its dependencies
- Creating a basic Ember.js application
- Ember.js application concepts

The origin of Ember.js

Ember.js is a fun and productive open source JavaScript framework used for creating ambitious web applications. It powers complex client-side applications and guarantees development productivity through use of common web conventions over trivial configurations. Its official website is <http://emberjs.com>.

It was forked from SproutCore by Yehuda Katz and Tom Dale. SproutCore is an MVC framework that strives to provide a robust JavaScript widget toolkit similar to Apple's Cocoa API for Mac OS X. The additional user interface widget feature was found to be unnecessary to most developers, hence the fork. The result was a more lightweight, easy-to-use library that still lived up to the promise of:

- Reducing development time
- Creating robust applications through use of common client-side web application development best practices
- Friendly API that makes client-side programming fun

Ember.js has a wide range of applications. It is well suited for applications that display dynamic data and have increased user interaction. Such applications include task managers, dashboards, forums, chat and messaging applications, and so on. Think of applications such as Gmail, Facebook, and Twitter. That being said, Ember.js is not ideal for static websites.

Ember.js is used by many companies throughout the world including, but not limited to, Apple, Groupon, Square, Zendesk, and Tilde Inc.

Downloading Ember.js

One of the most asked questions is, where do I download Ember.js from? The most stable version of library can be downloaded from <http://emberjs.com/builds/#/release>. However, the home page (<http://emberjs.com/>) usually contains a link to a starter kit that also contains the required dependencies. At the time of writing this book, the current stable version of Ember.js is version 1.7.0, which we will be using throughout the book. In our case, we will be using the corresponding starter kit from <https://github.com/emberjs/starter-kit/archive/v1.7.0.zip>, which you should download and unarchive into your working directory.

Upgrading Ember.js has been made much easier. New releases are usually announced at <http://emberjs.com/blog/tags/releases.html> and go in detail to discuss what to expect in the release.

Now, after unarchiving the provided starter kit, under `js/libs`, we notice the two basic requirements for running Ember.js:

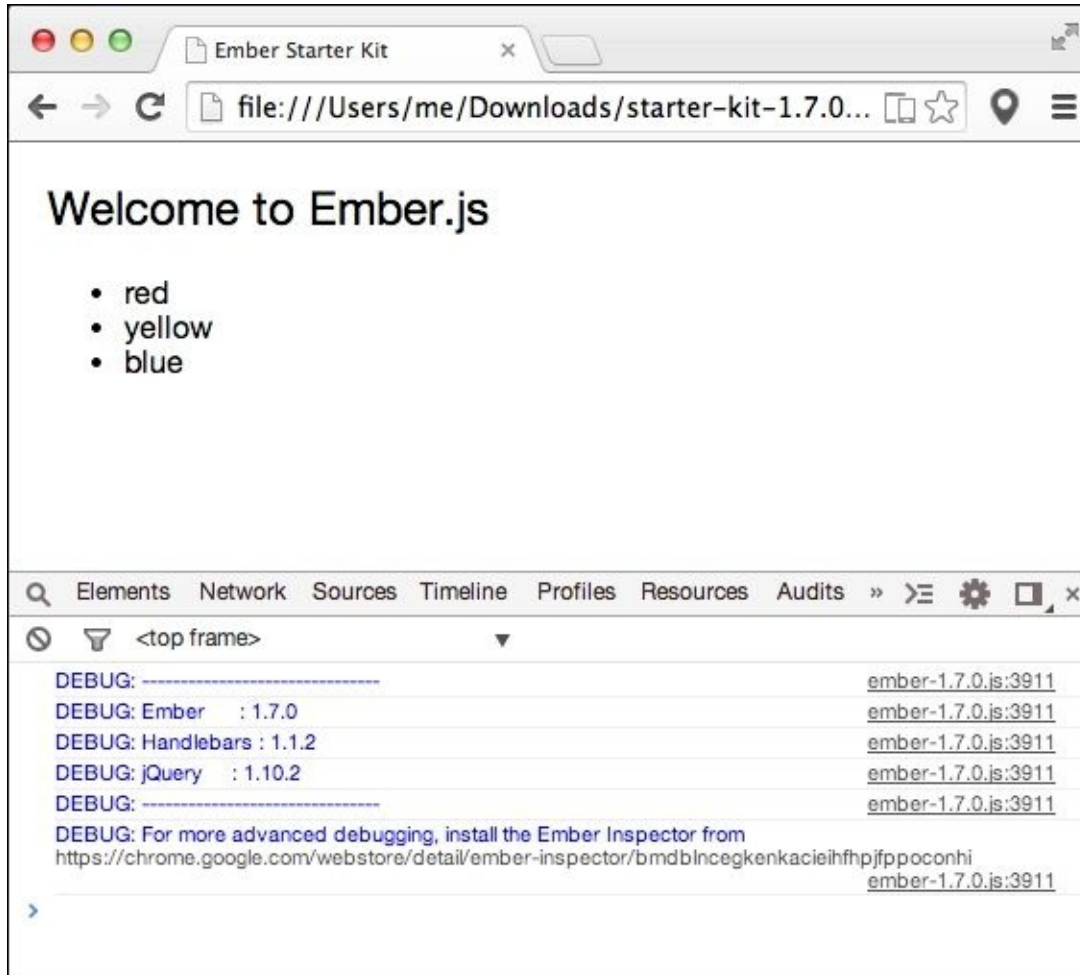
- **jQuery:** Ember.js uses jQuery for fundamental functions such as HTTP requests, DOM manipulation, and events management. jQuery is the most popular DOM manipulation library; hence, readers with past experience in it will feel at home. This also means that we will be able to easily integrate our favorite third-party jQuery libraries into our Ember.js applications.
- **Handlebars:** This is the template engine library that Ember.js uses to display reactive pages to the users through autoupdates and better user interactions. It's worth noting that we can still use other template engines such as Ender or Jade with a little bit of effort.

The index file loaded these files as:

```
<script src="js/libs/jquery-1.10.2.js"></script>
<script src="js/libs/handlebars-1.1.2.js"></script>
<script src="js/libs/ember-1.7.0.js"></script>
<script src="js/app.js"></script>
```

The `app.js` file contains all our application code but later on we may separate application concerns into more files when the application grows. It's worth noting that the order in which the scripts are loaded is important. Once the page

loads, Ember.js logs the dependencies used together with their versions, as shown in the following screenshot:



The two libraries and Ember.js can be accessed from the global scope as jQuery (or \$), Handlebars, and Ember (or Em) respectively, as shown in the following code:

```
console.log(jQuery);  
console.log(Handlebars);  
console.log(Ember);
```

Tip

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Creating your first application

The application script file (`js/app.js`) in the starter kit contains a basic Ember.js application. If you load the `index.html` file in your browser, you should see the three primary colors displayed:

```
App = Ember.Application.create();

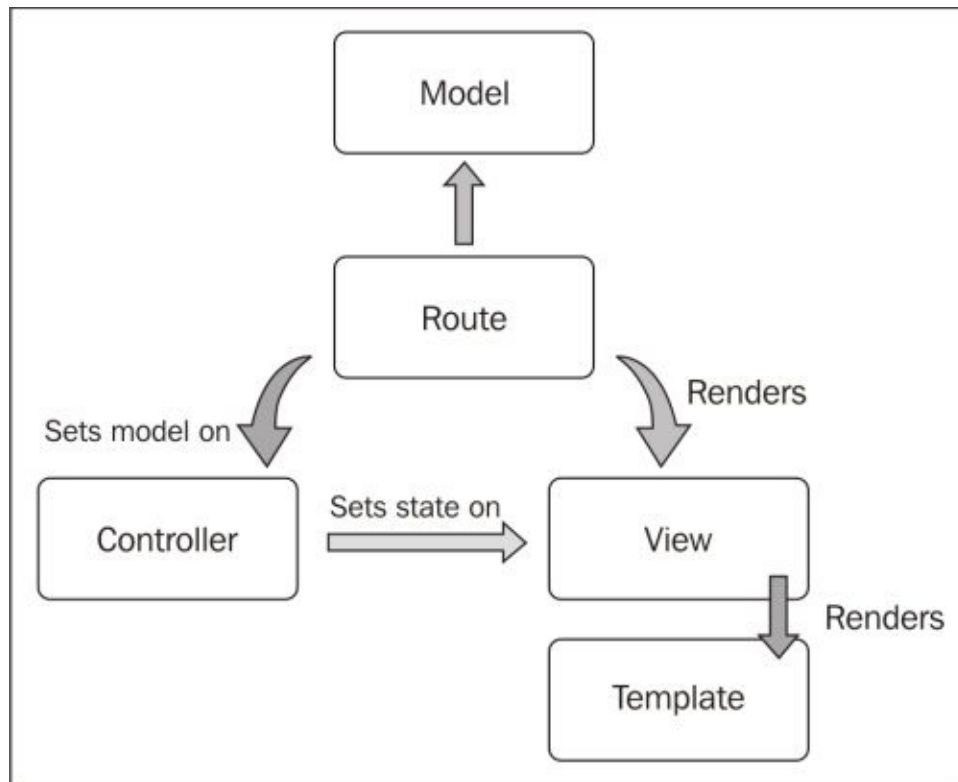
App.Router.map(function() {
  // put your routes here
});

App.IndexRoute = Ember.Route.extend({
  model: function() {
    return ['red', 'yellow', 'blue'];
  }
});
```

What steps led to the result?

- First, an Ember.js application was created which then created a router.
- The router that is responsible for state management transitioned the application into two states, the first of which was the application state. This state resulted in the application template being rendered into the DOM, hence the **Welcome to Ember.js** message.
- Superseding the application state was the index state whose route, `App.IndexRoute` rendered the index template inside the application template.
- The index route also provided the template, the lists, and the colors as the model context.

This can be summarized as in the following figure:



This example alone has introduced some of the following key Ember.js concepts.

Router

The router coordinates the application's state with the browser's location. It supports traditional web features such as navigation of the application's history using the browser's back and forward buttons, as well as linking back to the application using links.

Depending on the current URL, it calls matched routes that render several nested templates on a page. Each of these templates has a model context. The router is automatically created by an application on initialization. Therefore, we only need to call its `map` method to define the application routes, as shown in the following code:

```
App.Router.map(function() {  
  // put your routes here  
});
```

Route

The route is primarily responsible for providing a template's model context. It's defined from the `Ember.Route` class, as shown:

```
App.IndexRoute = Ember.Route.extend();
```

It will be covered extensively in [Chapter 3](#), *Routing and State Management*.

Controller

A controller proxies models provided by routes and further decorates them with display logic. They are also the channel of communication between the different states of an application through explicit dependency specification, as we shall learn in [Chapter 5](#), *Controllers*. To create controllers, we extend the `Ember.Controller` class, as shown in the following line of code:

```
App.IndexController = Ember.Controller.extend();
```

View

Views are used to manage events. They delegate user-generated events back to controllers and routes. Views are typically used to integrate other DOM manipulation libraries, for example, third-party jQuery packages. They are usually created from the `Ember.View` class:

```
App.IndexView = Ember.View.extend();
```

We will discuss them in detail in [Chapter 6](#), *Views and Event Management*.

Template

A template is set of expressions that are compiled down to HTML and rendered into the DOM. Templates are usually defined with the following signature:

```
<script type="text/x-handlebars" id="index">
<!-- our template goes in here -- >
</script>
```

Component

The component is a new concept in Ember.js that enables creation of reusable elements in accordance to the **W3C Web Components** specification. These elements are ideally not application-specific and can therefore be reused in other applications.

Initializing the application

An Ember.js application is created by instantiating the `Ember.Application` class:

```
App = Ember.Application.create();
```

A few things happen when an application is first created.

A new namespace is created for the app on which we define views, controllers, and routes. This prevents us from polluting the global scope. Therefore, defining a route, for example, should be attached to it as:

```
App.IndexRoute = Ember.Route.extend({
  model: function() {
    return ['red', 'yellow', 'blue'];
  }
});
```

Ember.js usually initializes the application by calling its `initialize` method. This initialization can be delayed by calling the application's `deferReadiness` method and then recommenced with `advanceReadiness`. For example, imagine our application needed to load the Google Client library beforehand, as shown:

```
<script src="https://apis.google.com/js/client.js?
onload=onLoadCallback">< /script>
<script src="app.js">
```

Here's how our application would finalize its readiness as soon as the library is loaded:

```
App = Em.Application.create();
App.deferReadiness();
window.onloadCallback = function(){
  App.advanceReadiness();
}
```

Now that our application is ready to use the SDK, we would load any currently logged-in user using an initializer via a promise. Initializers are called when the application is being initialized and are therefore a good opportunity to perform various functions, such as inject dependencies using the application's container. This container is used to organize the different components within the

application and can be referenced as:

```
App.__container__
```

For example, internally, an instance of a route could be accessed as:

```
App.__container__.lookup('route:index');
```

Since we have now loaded the third-party library, we can proceed to create a user initializer that would load any currently logged-in user:

```
Ember.Application.initializer({
  name: 'user',
  initialize: function(container, App) {

    var user = new Ember.RSVP.Promise(function(resolve, reject) {

      var opts = {
        'client_id': '-- --',
        'scope': 'email',
        'immediate': false
      };
      gapi
        .auth
        .authorize(opts, function(res){

          if (!res || res.error) {
            return reject();
          }
          resolve(res);
        });

    });

    container.register('user:main', user);
    container.lookup('user:main');
  }
});

Ember.Application.initializer({
  name: 'injectUser',
  initialize: function(container, App) {
    container.typeInjection('controller', 'user', 'user:main');
    container.typeInjection('route', 'user', 'user:main');
  }
});
```

The two blocks demonstrate the two uses of the application container as mentioned in the preceding code. The first one shows how to register an accessible application component; in this case, the user will now be accessed as:

```
App.__container__.lookup('user:main');
```

This user first starts off as a promise in the initializer. A **promise** is a stateful object whose value can be set at a later point in time. We will not cover much about promises in this chapter but one thing to note is that once the sign-in process completes, the sign-in callback `gapi.auth.authorize` either rejects or resolves the promise. Resolving the promise transitions the user object from the pending state to the fulfilled state.

The second block demonstrates dependency injection, which we also talked about earlier. In this case, we will now be able to access this user in routes and controllers as shown in the following code:

```
this.get('user');
```

Embedding Ember.js applications

An Ember.js application can be embedded into an existing page by specifying the application's `rootElement`. This attribute is a jQuery selector. For example, to embed the application into a `#chat-container` element, use the following lines of code:

```
App = Ember.Application.create({  
  rootElement: '#chat-container'  
});
```

This is useful when we are creating such applications as widgets. Specifying the root element ensures that only events invoked inside the element are managed by the Ember.js application.

Summary

This chapter has been an introductory guide to Ember.js. It focused on introducing the key elements that compose an Ember.js application. These elements inherit from the `Ember.Object` primitive, which will be discussed in the next chapter.

Chapter 2. Understanding Ember.js Objects and Mixins

Having learned how to create a basic Ember.js application in the previous chapter, this chapter will introduce us to Ember.js objects, which are the foundation of the rest of the base classes. Therefore, most of the concepts discussed will be applied throughout the book. By the end of this chapter, we'll be able to:

- Create Ember.js objects
- Define, get, and set object properties
- Define computed properties
- Register property change observers
- Use mixins

Creating objects in Ember.js

We all know how to define and create instances of function objects in JavaScript, as shown in the following code:

```
function Point(x, y, z){
  this.x = x;
  this.y = y;
  this.z = z;
}

Point.prototype.logX = function(){
  console.log(this.x);
}
Point.prototype.logY = function(){
  console.log(this.y);
}

Point.prototype.logZ = function(){
  console.log(this.z);
}

var point = new Point(3, 5, 7);
point.logX();
// 3
point.logY();
// 5
point.logZ();
// 7
```

The preceding example creates instances of a defined `Point` object that has three methods.

Ember.js uses JavaScript prototypes to simulate object-oriented features. More importantly, it introduces conveniences that enable easier inheritance and management of objects in the evented browser environment. A class, which is an object definition, is usually created by *extending* another user-defined or built-in class, typically `Ember.Object`. Classes have two methods, `create` and `extend`, which are used to create instances of objects and perform inheritance respectively. For example, the preceding code snippet would be implemented in Ember.js as:

```
var Point = Ember.Object.extend({
  x: null,
  y: null,
```

```

    z: null,
    logX: function(){
        console.log(this.get('x'));
    },
    logY: function(){
        console.log(this.get('y'));
    },
    logZ: function(){
        console.log(this.get('z'));
    }
});

```

We have just created an Ember.js class that has three properties x, y, and z, and their corresponding log methods. To create a new instance of this class, we will call the `create()` method on the class, as shown in the following example:

```

var point = Point.create({
    x: 3,
    y: 5,
    z: 7
}); point.logX();
// 3point.logY();
// 5point.logZ();
// 7

```

We can go further and extend our Point class to form a new class using the `extend()` method. For example, we can define a Vector class that defines an `add()` method, which adds to a provided vector, as shown in the following code:

```

var Vector = Point.extend({
    add: function(vector){

        var x = this.get('x') + vector.get('x');
        var y = this.get('y') + vector.get('y');
        var z = this.get('z') + vector.get('z');

        this.set('x', x);
        this.set('y', y);
        this.set('z', z);

    }
});

var vectorA = Vector.create({
    x: 3,

```

```
        y: 5,  
        z: 7  
    });  
  
    var vectorB = Vector.create({  
        x: 1,  
        y: 2,  
        z: 3  
    });  
  
    vectorA.add(vectorB);  
  
    vectorA.logX();  
    // 4  
    vectorA.logY();  
    // 7  
    vectorA.logZ();  
    // 10
```

After extending the Point class to a Vector class in the example, we created two vectors named vectorA and vectorB, and finally composed them.

Accessing object properties

We have just seen how Ember.js objects are created. Did you notice how Ember.js object properties are accessed? Ember.js provides the `get` and `set` property accessor methods. Why not access these values directly? Well, these methods are used to recalculate values as well as notify any changes made when necessary. For example:

```
var point = Point.create();
point.set('x', 3);
console.log(point.get('x')); // 3
```

Properties can also be read and set collectively using the `getProperties` and `setProperties` methods. This prevents Ember.js from unnecessarily making too many notifications about these changes, for example:

```
var point = Point.create();
point.setProperties({
  x: 1,
  y: 2,
  z: 3
});
console.log(point.getProperties('x', 'y', 'z'));
//{x: 1, y: 2, z: 3}
```

Defining class instance methods

Classes can also define instance methods. These methods have a similar signature to object properties, as shown in the following code:

```
var MyClass = ClassName.extend({
  methodName: function(){
    // method implementation
  }
});
```

Ember.js provides the ability to reuse implementations of parent methods in extended classes by the use of the `_super()` method. For example, the following example reimplements and reuses the `logX` method in the `Point` class:

```
var MyPoint = Point.extend({
  logX: function(){
    var x = this._super(); // call parent method
    console.log('x: %s', x);
  }
});
var myPoint = MyPoint.create({
  x: 3
});
myPoint.logX(); // x: 3
```

Ember.js objects usually define a constructor method called `init()`, which is called on an instance creation. Any initializations should be done inside this method. It's worth noting that the `_super()` method should always be called on any inherited methods such as `init()` to avoid losing parent implementations, for example:

```
var Book = Ember.Object.extend({
  init: function(){
    this._super();
    this.set('name', 'Mastering Ember.js'); // initialization
  }
});
```

Defining computed properties

What is a computed property? A computed property is the one whose value is returned from a function. For example:

```
var Movie = Ember.Object.extend({
  name: function(){
    return 'Transformers';
  }.property()
});

var movie = Movie.create();
console.log(movie.get('name')); // Transformers
```

The preceding example creates a computed property name that returns the name of a movie instance.

We simply transformed a method into a computed property by chaining the `property()` function to it. The true power of computed properties comes from them being able to produce different values based on prespecified dependent properties. These dependent properties are usually passed as arguments to the `property()` function. For example:

```
var Movie = Ember.Object.extend({
  year: '2007',
  seriesNumber: '1',
  name: function(){
    return this.get('seriesNumber') + '. Transformers - ' +
    this.get('year');
  }.property('year', 'seriesNumber')
});

var movie = Movie.create();
console.log(movie.get('name')); // 1. Transformers - 2007

movie.set('year', '2014');
movie.set('seriesNumber', '4');
console.log(movie.get('name')); // 4. Transformers - 2014
```

In the preceding example, the name property is always recomputed whenever a movie's `seriesNumber` and `year` change.

Computed properties can also have property dependencies of enumerable data.

The `@each` helper can be used to set up computed properties on such kinds of properties. For example:

```
var Country = Ember.Object.extend({
  stateNames: function(){
    return this.get('states').map(function(state){
      return state['name'];
    });
  }.property('states.@each.name')
});

var country = Country.create({
  states: [ {name: 'Texas'}, {name: 'Ohio'} ],
});
console.log(country.get('stateNames')); // ['Texas', 'Ohio']

country.set('states', [ {name: 'Alabama'}, {name: 'Arizona'} ]);
console.log(country.get('stateNames')); // ['Alabama', 'Arizona']
```

In the preceding example, we created a country object that has two states. We then defined a computed property, `stateNames` that returns an array of the state names. A change to any of the state names results in a recalculation of the property.

Defining property observers

In addition to computed properties, you can also set observers to properties. Observers are functions that get called when the properties they subscribe to change. They have the same signature as computed properties but use the `observes` function, as shown in the following code:

```
var MyClass = ClassName.extend({
  observerName: function(){
    // observer implementation
  }.observes([properties, ...])
});
```

The following example defines a session class that sets up an observer that makes the user relogin as soon as the session expires:

```
var Session = Em.Object.extend({
  expiredChanged: function(){
    if (this.get('expired')){
      window.location.assign('/login');
    }
  }.observes('expired')
});
```

Observers have no strict naming convention, but most developers name observers by appending `DidChange` to the property that is being observed, as shown:

```
var Song = Em.Object.extend({
  playedDidChange: function(){

  }.observes('played')
});
```

Just like computed properties, observers can also subscribe to an unlimited number of properties:

```
var Player = Em.Object.extend({
  inMotion: function(){

  }.observes('running', 'walking')
});
```

Here, the `inMotion` property will be recalculated when either the `running` or `walking` property changes.

Observers can also be set up and torn down using the `addObserver()` and `removeObserver()` methods respectively. These become handy when you want to manage observers yourself. For example, the preceding sample can be rewritten as:

```
var Session = Em.Object.extend({
  init: function(){

    var self = this;

    self._super();
    self.addObserver('expired', function(){
      if (self.get('expired')){
        self.removeObserver('expired');
        window.location.assign('/login');
      }
    });
  }
});
```

As shown in the preceding code, the `addObserver` method takes at least two arguments: the property to observe and the function to call whenever the property changes. In our example, we also tear down the observer listener by calling the `removeObserver` method. This method takes one argument, which is the property to unbind from.

Ember.js also provides a way to pass the context to use in the observer function. For example:

```
var Session = Em.Object.extend({
  init: function(){

    this._super();
    this.addObserver('expired', this, function(){
      if (this.get('expired')){
        this.removeObserver('expired');
        window.location.assign('/login');
      }
    });
  }
});
```

```
    }  
  });
```

It is important to note that observers only fire on property changes that occur after object initialization. An `on('init')` method can be applied to an observer to make it fire on changes that could occur during an object initialization. For example:

```
var Song = Em.Object.extend({  
  skipped: false,  
  played: false,  
  skippedDidChange: function(){  
    // does not fire on object initialization  
    console.log('song was skipped');  
  }.observes('skipped'),  
  playedDidChange: function(){  
    // fires on object initialization  
    console.log('song finished playing');  
  }.observes('played').on('init'),  
  init: function(){  
    this._super();  
    this.set('skipped', true);  
    this.set('played', true);  
  }  
});  
  
Song.create();
```

The example defines two observers: `skippedDidChange` and `playedDidChange`, of which, only the latter is called after an object initialization.

Creating property bindings

Ember.js provides support for both one- and two-way bindings. A binding is a link between two properties of the same or different objects, such that they are always in sync. This means that an update to one of the properties results in the other property being updated to the new value. Bindings are defined in the following signature:

```
property: Ember.computed.alias('otherProperty'),
```

In this case, the two properties `property` and `otherProperty` always stay in sync. Here's an example:

```
var author = Em.Object.create({
  name: 'J. K. Rowling'
});

var book = Em.Object.create({
  name: 'Harry Potter',
  authorName: Ember.computed.alias('author.name')
});

console.log(book.get('authorName')); // J. K. Rowling

author.set('name', 'Joanne Rowling');
console.log(book.get('authorName')); // Joanne Rowling
```

In the preceding example, the `book` instance has an `author` property that binds to the created global `author` instance. Any changes made to the name of the author will be reflected in the bound `book` author property. Likewise, any changes made to the book's author property will be propagated back to the global author as shown:

```
book.set('authorName', 'Joanne Jo Rowling');
console.log(author.get('name')); // Joanne Jo Rowling
```

This is an example of a two-way binding where an update to either property results in the other property being updated.

Ember.js also supports one-way bindings where updates are unidirectional. A property can subscribe to updates from a different property but will not update the latter if the former changes. For example:


```
var author = Em.Object.create({
  name: 'J. K. Rowling'
});

var book = Em.Object.create({
  name: 'Harry Potter',
  authorName: Ember.computed.oneWay('author')
});

console.log(book.get('authorName')); // J. K. Rowling

book.set('author.name', 'Joanne Rowling');
// author's name remains unchanged
console.log(author.get('name')); // J. K. Rowling
```

In the preceding snippet, the book property changes will not affect the bound author's name property.

Using mixins

Mixins are abstract definitions that define methods and properties that classes and objects can reuse. For example, consider these two objects:

```
var myView = Em.View.create({
  sum: function(a, b){
    return a+b;
  }
});

var myController = Em.Controller.create({
  sum: function(a, b){
    return a+b;
  }
});
```

These two objects share a common function that can be abstracted into a mixin:

```
var sumMixin = Em.Mixin.create({
  sum: function(a, b){
    return a+b;
  }
});

var myView = Em.View.createWithMixins(sumMixin);

var myController = Em.Controller.createWithMixins(sumMixin);
```

Any number of mixins can be passed to objects or classes on creation or definition, respectively:

```
App.Number = Em.Object.extend(sumMixin, diffMixin, productMixin);
```

It is important to note that mixins are always *created* and never *extended*. The example also showed that objects reusing mixins are always created using the `createWithMixins` method on instantiation and not the `create` method. However, classes still use the `extend` method when applying mixins.

Reopening classes and instances

Sometimes, it is necessary to update class implementations without redefining them. This is usually necessary when we do not wish to extend built-in classes, but only want to update their implementations. Ember.js refers to this as reopening of classes and objects. Class methods and properties can be reimplemented using the `reopenClass` method, while instance methods and properties can be updated using the `reopen` method. It's however discouraged to change built-ins as they may change in future versions.

For example:

```
var Book = Em.Object.extend();

Book.reopen({
  id: null,
  title: null,
  purchase: function(){
    console.log('sold');
  }
});

Book.reopenClass({
  getById: function(id){
    return Book.create({
      id: '456',
      title: 'Harry Potter'
    });
  }
});

Book.create({
  id: 456,
  title: 'Harry Potter'
});

var book = Book.getById(456);

book.purchase();
```

In the preceding example, we added an instance method, `purchase`, and two properties, `id` and `name` to the already defined `Book` class. We have also added a class method, `getById`, without extending the class.

Event subscription

Another way of notifying changes in an application is through event subscription. This paradigm is used heavily in Node.js to channel messages and events across different components of an application. Ember.js provides the `Ember.Evented` mixin that can be used to serve archive this easily. For example, blocks in a board game can subscribe to instructions from an actuator as in the following example:

```
var GRID_SIZE = 4;
var actuator = Em.Object.createWithMixins(Em.Evented);

var block = Em.Object.createWithMixins(Em.Evented);
actuator.on('moveRight', function(){
  var x = block.get('x') + 1;
  x = x % GRID_SIZE;
  block.set('x', x);
});

actuator.trigger('moveRight');
```

The mixin provides five essential methods, two of which have been illustrated in the preceding example:

- `on`: This is used to subscribe to an event
- `off`: This is used to disable a subscription
- `once`: This is used to subscribe once to an event
- `trigger`: This is used to emit an event
- `has`: This is used to check if an event has been subscribed to

Summary

This chapter has focused on introducing Ember.js objects. We will use these objects extensively in the next chapter, where we will learn how state management is accomplished in Ember.js using routers. We will discuss how to construct routes and routers. That being said, you should have learned the following Ember.js object concepts in this chapter:

- Creating objects and classes in Ember.js
- Getting and setting object properties
- Defining computed properties
- Defining property observers
- Creating property bindings
- Using mixins
- Reopening Ember.js classes

In the next chapter, we will be discussing routes, which are one of these classes that are extended from `Ember.Object`.

Chapter 3. Routing and State Management

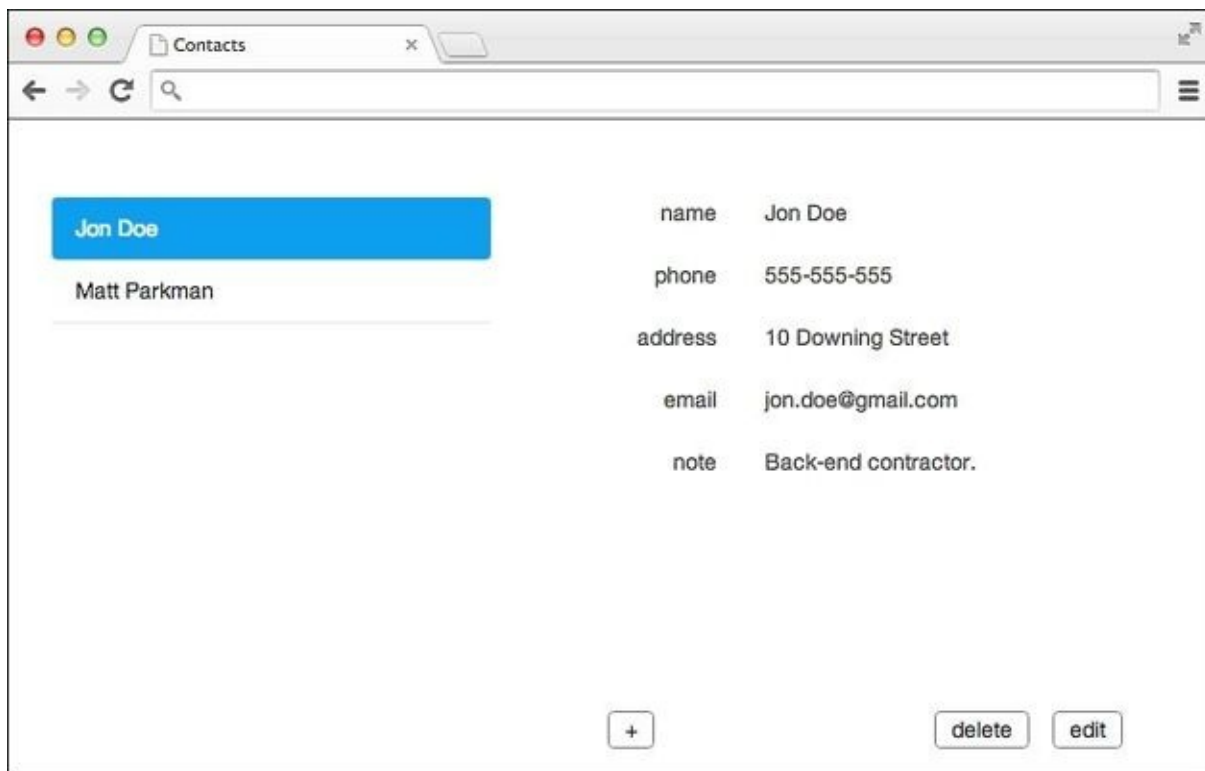
In this chapter, we will learn URL-based state management in Ember.js, which constitutes **routing**. Routing enables us to translate different states in our applications into URLs and vice-versa. It is a key concept in Ember.js that enables developers to easily separate application logic. It also enables users to link back to content in the application via the usual HTTP URLs. That being said, by the end of this chapter, we should be able to accomplish the following:

- Create a router
- Define resources and routes
- Define a route's model
- Perform a redirect
- Accomplish asynchronous routing

Creating the application's router

We all know that in traditional web development, every request is linked by a URL that enables the server make a decision on the incoming request. Typical actions include sending back a resource file or JSON payload, redirecting the request to a different resource, or sending back an error response such as in the case of unauthorized access.

Ember.js strives to preserve these ideas in the browser environment by enabling association between these URLs and state of the application. The main component that manages these states is the application router. As mentioned in the introductory section, it is responsible for restoring an application to a state matching the given URL. It also enables the user to navigate between the application's history as expected. The router is automatically created on application initialization and can be referenced as `MyApplicationNamespace.Router`. Before we proceed, we will be using the bundled chapter sample to better understand this extremely convenient component. The sample is a simple implementation of the Contacts OS X application as shown in the following screenshot:



It enables users to add new contacts as well as edit and delete existing ones. For simplicity, we won't support avatars, but that could be an implementation exercise for the reader at the end of the chapter.

We already mentioned some of the states in which this application can transition into. These states have to be registered in the same way server-side frameworks have URL dispatchers that backend programmers use to map URL patterns to views. The chapter sample already illustrates how these possible states are defined:

```
// app.js

var App = Ember.Application.create();

App.Router.map(function() {
  this.resource('contacts', function(){
    this.route('new');
    this.resource('contact', {path: '/:contact_id'}, function(){
      this.route('edit');
    });
  });
  this.route('about');
});
```

Notice that the already instantiated router was referenced as `App.Router`. Calling its `map` method gives the application an opportunity to register its possible states. In addition, two other methods are used to classify these states into *routes* and *resources*.

Mapping URLs to routes

When defining routes and resources, we are essentially mapping URLs to possible states in our application. As shown in the first code snippet, the router's `map` function takes a function as its only argument. Inside this function, we may define a resource using the corresponding method, which takes the following signature:

```
this.resource(resourceName, options, function);
```

The first argument specifies the name of the resource and coincidentally, the path to match the request URL. The next argument is optional and holds configurations that we may need to specify, as we shall see later. The last one is a function that is used to define the routes of that particular resource. For example, the first defined resource in the samples says, let the `contacts` resource handle any requests whose URL starts with `/contacts`. It also specifies one route, `new`, that is used to handle the creation of new contacts. Routes on the other hand, accept the same arguments for the function argument.

You must be asking yourself, "So how are routes different from resources?" The two are essentially the same, other than the former offers a way to categorize states (routes) that perform actions on a specific entity. We can think of an Ember.js application as a tree, composed of a trunk (the router), branches (resources), and leaves (routes). For example, the `contact` state (a resource) caters for a specific contact. This resource can be displayed in two modes: `read` and `write`; hence, the `index` and `edit` routes respectively, as shown:

```
this.resource('contact', {path: '/:contact_id'}, function(){
  this.route('index'); // auto defined
  this.route('edit');
});
```

Because Ember.js encourages convention, there are two components of routes and resources that are always autodefined:

- A default application resource: This is the master resource into which all other resources are defined. We therefore do not need to define it in the router. It's not mandatory to define resources on every state. For example, our `about` state is a route because it only needs to display static content to

the user. It can however be thought to be a route of the already autodefined application resource.

- A default index route on every resource: Again, every resource has a default index route. It's autodefined because an application cannot settle on a resource state. The application therefore uses this route if no other route within this same resource was intended to be used.

Nesting resources

Resources can be nested depending on the architecture of the application. In our case, we need to load contacts in the sidebar before displaying any of them to the user. Therefore, we need to define the contact resource inside the contacts. On the other hand, in an application such as Twitter, it won't make sense to define a tweet resource embedded inside a tweets resource because an extra overhead will be incurred when a user just wants to view a single tweet linked from an external application.

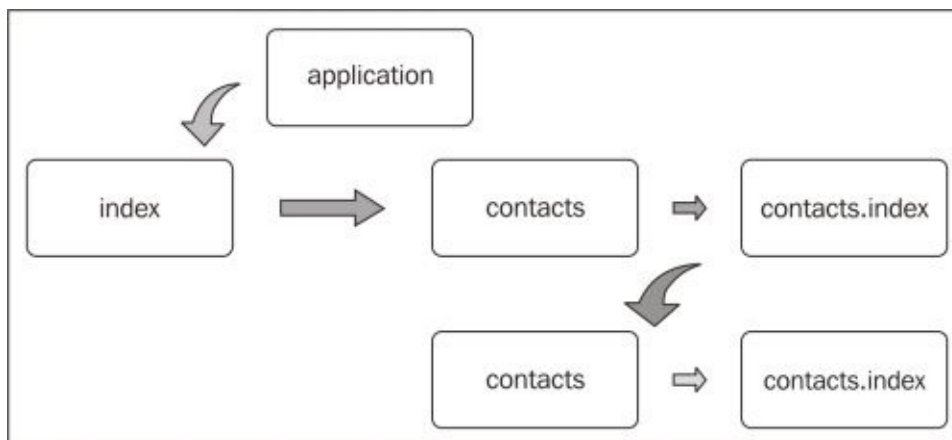
Understanding the state transition cycle

A request is handled in the same way water travels from the roots (the application), up the trunk, and is eventually lost off leaves. This request we are referring to is a change in the browser location that can be triggered in a number of ways, as we shall discover more in the next chapter.

Before we proceed into finer details about routes, let's discuss what happened when the application was first loaded. On boot, a few things happened as outlined:

- The application first transitioned into the application state, then the index state.
- Next, the application index route redirected the request to the contacts resource.
- Our application uses the browser's local storage to store the contacts and so for demoing purposes, the contacts resource populated this store with fixtures (located at `fixtures.js`).
- The application then transitioned into the corresponding contacts resource index route, `contacts.index`.
- Again, here we made a few decisions based on whether our store contained any data. Since we indeed have data, we redirected the application into the contact resource, passing the ID of the first contact along.
- Just as in the two preceding resources, the application transitioned from this last resource into the corresponding index route, `contact.index`.

The following figure gives a good view of the preceding state change:



Configuring the router

The router can be customized in the following ways:

- Logging state transitions
- Specifying the root app URL
- Changing browser location lookup method

During development, it may be necessary to track the states into which the application transitions. Enabling these logs is as simple as:

```
var App = Ember.Application.create({  
  LOG_TRANSITIONS: true  
});
```

As illustrated, we enable the `LOG_TRANSITIONS` flag when creating the application. If an application is not served at the root of the website domain, then it may be necessary to specify the path name used, as in the following example:

```
App.Router.reopen({  
  rootURL: '/contacts/'  
});
```

One other modification we may need to make revolves around the techniques Ember.js uses to subscribe to the browser's location changes. This makes it possible for the router to do its job of transitioning the app into the matched URL state. Two of these methods are as follows:

- Subscribing to the hashchange event
- Using the `history.pushState` API

The default technique used is provided by the `HashLocation` class documented at <http://emberjs.com/api/classes/Ember.HashLocation.html>. This means that URL paths are usually prefixed with the hash symbol, for example, `/#/contacts/1/edit`. The other one is provided by the `HistoryLocation` class located at <http://emberjs.com/api/classes/Ember.HistoryLocation.html>. This does not distinguish URLs from the traditional ones and can be enabled as:

```
App.Router.reopen({  
  location: 'history'  
});
```

We can also opt to let Ember.js pick which method is best suited for our app with the following code:

```
App.Router.reopen({  
  location: 'auto'  
});
```

If we don't need any of these techniques, we could opt to do so especially when performing tests:

```
App.Router.reopen({  
  location: none  
});
```

Specifying a route's path

We now know that when defining a route or resource, the resource name used also serves as the path the router uses to match request URLs. Sometimes, it may be necessary to specify a different path to use to match states. There are two common reasons that may lead us to do this, the first of which is good for delegating route handling to another route. Although, we have not yet covered route handlers, we already mentioned that our application transitions from the application index route into the `contacts.index` state. We may however specify that the contacts route handler should manage this path as:

```
this.resource('contacts', {path: '/'}, function(){
});
```

Therefore, to specify an alternative path for a route, simply pass the desired route in a hash as the second argument during resource definition. This also applies when defining routes.

The second reason would be when a resource contains dynamic segments. For example, our contact resource handles contacts who should obviously have different URLs linking back to them. Ember.js uses URL pattern matching techniques used by other open source projects such as Ruby on Rails, Sinatra, and Express.js. Therefore, our contact resource should be defined as:

```
this.resource('contact', {path: '/:contact_id'}, function(){
});
```

In the preceding snippet, `/:contact_id` is the dynamic segment that will be replaced by the actual contact's ID. One thing to note is that nested resources prefix their paths with those of parent resources. Therefore, the contact resource's full path would be `/contacts/:contact_id`. It's also worth noting that the name of the dynamic segment is not mandated and so we could have named the dynamic segment as `/:id`.

Defining route and resource handlers

Now that we have defined all the possible states that our application can transition into, we need to define handlers to these states. From this point onwards, we will use the terms *route* and *resource handlers* interchangeably. A route handler performs the following major functions:

- Providing data (model) to be used by the current state
- Specifying the view and/or template to use to render the provided data to the user
- Redirecting an application away into another state

Before we move into discussing these roles, we need to know that a route handler is defined from the `Ember.Route` class as:

```
App.RouteHandlerNameRoute = Ember.Route.extend();
```

This class is used to define handlers for both resources and routes and therefore, the naming should not be a concern. Just as routes and resources are associated with paths and handlers, they are also associated with controllers, views, and templates using the Ember.js naming conventions. For example, when the application initializes, it enters into the application state and therefore, the following objects are sought:

- The application route
- The application controller
- The application view
- The application template

In the spirit of *do more with reduced boilerplate* code, Ember.js autogenerates these objects unless explicitly defined in order to override the default implementations. As another example, if we examine our application, we notice that the `contact.edit` route has a corresponding `App.ContactEditController` controller and `contact/edit` template.

We do not need to define its route handler or view. Having seen this example, when referring to routes, we normally separate the resource name from the route name by a period as in the following:

```
resourceName.routeName
```


In the case of templates, we may use a period or a forward slash:

resourceName/routeName

The other objects are usually camelized and suffixed by the class name:

ResourcenameRoutenameClassname

For example, the following table shows all the objects used in our chapter sample. As mentioned earlier, some are autogenerated.

Route Name	Controller	Resource
applicationApplicationControllerApplicationRoute	ApplicationViewapplication	
indexIndexControllerIndexRoute	IndexViewindex	
about	AboutController	Ab
contactsContactsControllerContactsRoute	ContactsView	
contacts.indexContactsIndexControllerContactsIndexRoute	ContactsIndexViewcontacts/index	
contacts.newContactsNewController	ContactsNewRoute	
contact	ContactController	Co
contact.index	ContactIndexController	Co
contact.edit	ContactEditController	Co

One thing to note is that objects associated with the intermediary application state do not need to carry the suffix; hence, just index or about.

Specifying a route's model

In the first chapter, we mentioned that route handlers provide controllers, the data needed to be displayed by templates. These handlers have a `model` hook that can be used to provide this data in the following format:

```
AppNamespace.RouteHandlerName = Ember.Route.extend({
  model: function(){
  }
});
```

For instance, the route `contacts` handler in the chapter sample loads any saved contacts from local storage as:

```
model: function(){
  return App.Contact.find();
}
```

We have abstracted this logic into our `App.Contact` model. Notice how we reopen the class in order to define this static method. As a recap of this lesson in [Chapter 2, Understanding Ember.js Objects and Mixins](#), a static method can only be called by the class of that method and not its instances:

```
App.Contact.reopenClass({
  find: function(id){
    return (!!id)
      ? App.Contact.findOne(id)
      : App.Contact.findAll();
  },
  ...
});
```

If no arguments are passed to the method, it goes ahead and calls the `findAll` method, which uses the local storage helper to retrieve the contacts:

```
findAll: function(){
  var contacts = store('contacts') || [];
  return contacts.map(function(contact){
    return App.Contact.create(contact);
  });
}
```

Because we want to deal with contact objects, we iteratively convert the contents

of the loaded contact list. If we examine the corresponding template, contacts, we notice that we are able to populate the sidebar as shown in the following code:

```
<ul class="nav nav-pills nav-stacked">
  {{#each model}}
    <li>
      {{#link-to "contact.index" this}}{{name}}{{/link-to}}
    </li>
  {{/each}}
</ul>
```

Do not worry about the template syntax at this point if you're new to Ember.js. The important thing to note is that the model was accessed via the `model` variable. Of course, before that, we check to see if the model has any content in:

```
{{#if model.length}}
  ...
{{else}}
  <h1>Create contact</h1>
{{/if}}
```

As we shall see later, if the list was empty, the application would be forced to transition into the `contacts.new` state, in order for the user to add the first contact as shown in the following screenshot:

The screenshot shows a web browser window with a single tab titled 'Contacts'. The address bar is empty. The main content area displays a 'Create contact' form. On the left, the text 'Create contact' is in a large, bold font. To the right of this text is a form with several input fields, each with a label to its left: 'first name' (containing 'Mac'), 'last name', 'email', 'phone', 'address', and 'note'. At the bottom of the form, there are two buttons: a small square button with a '+' sign and a rounded rectangular button labeled 'done'.

The contact handler is a different case. Remember we mentioned that its path has a dynamic segment that would be passed to the handler. This information is passed to the model hook in an options hash as:

```
App.ContactRoute = Ember.Route.extend({
  model: function(params){
    return App.Contact.find(params.contact_id);
  },
  ...
});
```

Notice that we are able to access the contact's ID via the `contact_id` attribute of the hash. This time, the `find` method calls the `findOne` static method of the contact's class, which performs a search for the contact matching the provided ID, as shown in the following code:

```
findOne: function(id){
  var contacts = store('contacts') || [];
  var contact = contacts.find(function(contact){
    return contact.id == id;
```

```
});  
if (!contact) return;  
return App.Contact.create(contact);  
}
```

Serializing resources

We've mentioned that Ember.js supports content to be linked back externally. Internally, Ember.js simplifies creating these links in templates. In our sample application, when the user selects a contact, the application transitions into the `contact.index` state, passing his/her ID along. This is possible through the use of the `link-to` handlebars expression:

```
{{#link-to "contact.index" this}}{{name}}{{/link-to}}
```

Again, we will revisit this later, in detail, in [Chapter 4, Writing Application Templates](#), but for now, the important thing to note is that this expression enables us to construct a link that points to the said resource by passing the resource name and the affected model. The destination resource or route handler is responsible for yielding this path constituting **serialization**. To serialize a resource, we need to override the matching `serialize` hook as in the `contact` handler case shown in the following code:

```
App.ContactRoute = Ember.Route.extend({  
  ...  
  serialize: function(model, params){  
    var data = {}  
    data[params[0]] = Ember.get(model, 'id');  
    return data;  
  }  
});
```

Serialization means that the hook is supposed to return the values of all the specified segments. It receives two arguments, the first of which is the affected resource and the second is an array of all the specified segments during the resource definition. In our case, we only had one and so we returned the required hash that resembled the following code:

```
{contact_id: 1}
```

If we, for example, defined a resource with multiple segments like the following code:

```
this.resource(  
  'book',  
  {path: '/name/:name/:publish_year'},
```

```
function(){  
  }  
);
```

The serialization hook would need to return something close to:

```
{  
  name: 'jon+doe',  
  publish_year: '1990'  
}
```

Asynchronous routing

In actual apps, we would often need to load the model data in an asynchronous fashion. There are various approaches that can be used to deliver this kind of data. The most robust way to load asynchronous data is through use of **promises**. Promises are objects whose unknown value can be set at a later point in time. It is very easy to create promises in Ember.js. For example, if our contacts were located in a remote resource, we could use jQuery to load them as:

```
App.ContactsRoute = Ember.Route.extend({
  model: function(params){
    return Ember.$.getJSON('/contacts');
  }
});
```

jQuery's HTTP utilities also return promises that Ember.js can consume. As a by-the-way, jQuery can also be referenced as `Ember.$` in an Ember.js application. In the preceding snippet, once data is loaded, Ember.js would set it as the model of the resource. However, one thing is missing. We require that the loaded data be converted to the defined contact model, as shown in the following little modification:

```
App.ContactsRoute = Ember.Route.extend({
  model: function(params){
    var promise = Ember
      .Object
      .createWithMixins(Ember.DeferredMixin);

    Ember
      .$
      .getJSON('/contacts')
      .then(reject, resolve);

    function resolve(contacts){
      contacts = contacts.map(function(contact){
        return App.Contact.create(contact);
      });
      promise.resolve(contacts)
    }

    function reject(res){
      var err = new Error(res.responseText);
      promise.reject(err);
    }
  }
});
```



```

    }
    return promise;
  }
});

```

We first create the promise, kick off the XHR request, and then return the promise while the request is still being processed. Ember.js will resume routing once this promise is rejected or resolved. The XHR call also creates a promise, so we need to attach to it the then method which essentially says, *invoke the passed resolve or reject function on successful or failed load respectively*. The resolve function converts the loaded data and resolves the promise; passing the data along thereby resumes routing. If the promise was rejected, the transition fails with an error. We will see how to handle this error in a moment.

Note that there are two other flavors we can use to create promises in Ember.js, as shown in the following examples:

```

var promise = Ember.Deferred.create();

Ember
  .$.getJSON('/contacts')
  .then(success, fail);

function success(){
  contacts = contacts.map(function(contact){
    return App.Contact.create(contact);
  });
  promise.resolve(contacts)
}

function fail(res){
  var err = new Error(res.responseText);
  promise.reject(err);
}

return promise;

```

The second example is as follows:

```

return new Ember.RSVP.Promise(function(resolve, reject){

  Ember

```

```
.$.  
.getJSON('/contacts')  
.then(success, fail);  
  
function success(){  
  contacts = contacts.map(function(contact){  
    return App.Contact.create(contact);  
  });  
  resolve(contacts)  
}  
  
function fail(res){  
  var err = new Error(res.responseText);  
  reject(err);  
}  
  
});
```

Configuring a route's controller

We just learned that routes provide their corresponding controllers, data that they proxy to templates and views. This usually happens in the `setupController` hook of the route. For example:

```
App.ContactsRoute = Ember.Route.extend({
  setupController: function(controller, model){
    controller.set('model', model);
  }
});
```

Although we'll rarely need to use it, this hook provides a good opportunity to modify other controllers. For example, we could set a property on the application controller as:

```
App.ContactsIndexRoute = Ember.Route.extend({
  setupController: function(controller, model){
    this._super(controller, model);
    this
      .controllerFor('application')
      .set('contacts', this.modelFor('contacts'));
  }
});
```

We'd never actually write such code in an actual app but this snippet introduces two convenience methods, `modelFor` and `controllerFor`, that can be used to access the models and controllers of other handlers respectively. Note that the argument passed is the route's or resource's name. Here are more examples:

```
this.modelFor('contacts.index');
this.controllerFor('contact.edit');
```

Sometimes, we may want to specify a different controller that a handler should use. For example, the `contact.edit` route is used to edit the contact resource model. In this case, we needed to specify that the former depended on the latter via the `needs` property. That way, as we'll learn in [Chapter 5, Controllers](#), the `contact.edit` route's template was able to access the model that was set on the controller as:

```
{{#with controller.controllers.contact}}
...
```

```
{{/with}}
```

An alternative approach would be to use this controller directly by specifying it in the handler as:

```
App.ContactEditRoute = Ember.Route.extend({  
  controllerName: 'contact'  
});
```

The result would be that the `contact.edit` template would be similar to that of the `contacts.new` template and, therefore both can be removed as explained in the next section.

Rendering the route's template

Before we discuss more on route handler template rendering, it's worth discussing what happens when the application transitions between states in the context of templates. This section will be revisited in detail in the next chapter. In our chapter example, the application eventually knows the transition, as outlined in order:

- application state
- contacts state
- contact state
- contact.index state

Therefore, the application template is first rendered on the screen. The next template, contacts, is then rendered into the application template to constitute the sidebar. Next, the contact template is inserted into the contacts template. Lastly, the contact.index template was inserted into the contact template to complete the transition. Each template specifies an outlet portion into which child route handlers can render their templates. For example, note the outlet expression in the following application template:

```
<script type="text/x-handlebars">
  <div class="container">
    {{outlet}}
  </div>
</script>
```

A route handler can specify the template to use in the same way as controllers. Again, revisiting the contact.edit route template, it uses a partial, which we shall discuss in the next chapter, and includes the shared contacts.form template into the host template.

The renderTemplate hook is the last chance for the handler to specify a custom template to use by invoking the render method and passing in the template to use along, as shown in the following code:

```
App.ContactEditRoute = Ember.Route.extend({
  renderTemplate: function() {
    this.render('contacts/form');
  }
});
```

```
});
```

In this case, we can therefore get rid of the defined controllers and routes of the `contacts.new` and `contact.edit` routes. Lastly, templates are not limited to a single outlet. This means that you can render different templates with different controller contexts in the current state template. For example, in a game application, we could define two outlets to host two different templates serving different purposes as:

```
<script type="text/x-handlebars" data-template-name="game">
  <div id="leaderboard">{{outlet leaderboard}}</div>
  <div id="mainboard">{{outlet mainboard}}</div>
</script>
```

And then, render them via the handler as:

```
App.GameRoute = Ember.Route.extend({
  renderTemplate: function() {
    this.render('mainboard', {
      into: 'game',
      outlet: 'mainboard',
      controller: 'mainboard'
    });
    this.render('leaderboard', {
      into: 'game',
      outlet: 'leaderboard',
      controller: 'leaderboard'
    });
  }
});
```

Redirecting state

A common use of a handler is to redirect the application into another state in the same way we may be redirected to a 404 page if the requested resource was not found by the underlying server. In our sample app, the index controller overrides the route handler's `redirect` hook in order to redirect the application into the `contacts` state using the `transitionTo` method, as shown in the following code:

```
App.IndexRoute = Ember.Route.extend({
  redirect: function(){
    this.transitionTo('contacts');
  });
```

There are two cases where we may need to perform this redirection. The first is when we don't need to know the model of the route handler. We used one of these hooks, `beforeModel`, to populate the contact list local store with fixtures just before the same handler went ahead to load them, as shown in the following code:

```
App.ContactsRoute = Ember.Route.extend({
  beforeModel: function(){
    var contacts = store('contacts') || CONTACTS;
    store('contacts', contacts);
  }
});
```

On the other hand, we can either use the `redirect` and `afterModel` hooks if we needed to wait for the handler's model to load. In fact, the latter actually just calls the former. For example, we used the `afterModel` hook in the `contacts` route handler in the chapter sample to determine whether we needed to force the user to add a new contact or redirect them to view the first contact, as shown in the following code:

```
App.ContactsIndexRoute = Ember.Route.extend({
  afterModel: function(){
    var model = this.modelFor('contacts') || [];
    var contact = model.get('firstObject');
    if (!contact) return this.transitionTo('contacts.new');
    return this.transitionTo('contact.index', contact);
  },
});
```

Catching routing errors

If a transition into a route fails, for example, failure to load the model, Ember.js emits the error action in that handler. Although we have not covered actions, think of them as events that can be delegated to a handler from templates or other route handlers and controllers. The following example catches such errors by redirecting the application to an appropriate error handling route:

```
App.ContactsRoute = Ember.Route.extend({
  action: {
    error: function(error){
      this.controllerFor('error').set('error', error);
      this.transitionTo('error');
    }
  }
});
```


Summary

This chapter has detailed how states are managed in Ember.js. We have particularly discussed how an Ember.js application boots from an application state into other nested states. A few concepts were introduced in the chapter sample and will be discussed in detail in the next chapter. That being said, in the next chapter, we will discuss templates, especially how they render the data proxied by controllers and how they delegate user generated actions back to routes. Therefore, you should have a solid understanding of the following topics covered in this chapter since they'll be frequently revisited:

- Defining application routes
- Defining an application route
- Implementing a route's model
- Setting up a route's controllers
- Specifying a route's template
- Performing asynchronous routing

The next chapter will describe how to include templates using script tags or compiling and bundling/shipping templates from the server.

Chapter 4. Writing Application Templates

Now that we know how to manage the states in Ember.js applications using routes, this chapter will help us master how to present application logic to users using templates. You will soon realize that the bulk of your application resides in templates. This being said, the chapter will frequently revisit what we already learned so far. Therefore, by the end of this chapter, the following concepts will be learned:

- Creating templates
- Writing binding template expressions and conditionals
- Changing contexts in templates
- Creating event listeners in templates
- Extending templates
- Writing custom template helpers

Registering templates

As promised, we will continue to explore the chapter sample introduced in the previous chapter in the context of templates. When an application transitions into a state, each of the route handlers in that state path renders a template into the page. These templates are defined in the following signature:

```
<script type="text/x-handlebars" id="index">
  <h1>My Index Template</h1></script>
```

As shown, templates are registered using **script** tags of the `text/x-handlebars` type. The `id` or `data-template-name` attribute is used to identify the template. For instance, the chapter sample `contacts` template was defined as:

```
<script type="text/x-handlebars" data-template-name="contacts">
  ...
</script>
```

One thing to note is that it's wiser to identify templates using the `data-template-name` attribute instead of the `id` attribute, as the former is more likely to collide with other existing elements. Also, notice that the first template was not *identified*. This is because any unidentified template is considered as the application template:

```
<script type="text/x-handlebars">
  <div class="container">
    {{outlet}}
  </div>
</script>
```

Inserting templates

In the previous chapter, we discussed how a state in an application is composed of various routes whose handlers are sequentially called in order to perform various functions that make up this state. As a recap, when the user loaded the application in the chapter sample, the application transitioned into the application state. The application route handler then rendered its corresponding application template into the DOM. The next route handler to be called was the `contacts` route handler that also loaded and rendered its template into the application template. We already discussed that the `{{outlet}}` Handlebars expression was the portion of the application template that got swapped out. This process repeated itself until the application settled on the destination state.

As we will discuss later, a *parent* template can specify named outlets into which multiple *child* templates can be rendered. Route (not resource) handlers need not include this expression because they usually render the final template.

Writing out templates

As discussed earlier, Ember.js templates are written in the Handlebars (www.handlebarsjs.com) syntax whose library was created by the same authors to simplify the creation of client-side templates. Handlebars is a powerful templating library that offers many features, which will be discussed in the upcoming sections.

Expressing variables

We just mentioned that templates are backed by data that is proxied by the corresponding controller. Handlebars walks down a template, replacing defined expressions with matching values obtained from this data. These expressions are usually variable names enclosed in curly braces. The `{{outlet}}` expression we just discussed is one such expression. In the chapter sample, the `contact.index` state is responsible for displaying a contact's detail on the right-hand side of the page. In its corresponding template, we notice that the contact's attributes are expressed using these expressions, but later get replaced, as shown in the following code:

```
<script type="text/x-handlebars" data-template-
name="contact/index">
  {{#with controller.controllers.contact}}
    <div class="row">
      <div class="col-sm-4 text-right">name</div>
      <div class="col-sm-8">{{name}}</div>
    </div>
    <br>
    ...
  </script>
```

In the preceding example, Handlebars finds the `name` expression, retrieves this variable from the provided model, and performs the swap. Handlebars always works in the provided controller context, which in turn proxies requests to its model. Therefore, the value used to swap the preceding expression is evaluated as:

```
model.name;
```

This value can also be:

```
{{controller.model.name}}
```

Whenever the `name` variable reference request is made, Ember.js first checks if the controller defines the variable. Since this is not true, the controller *proxies* this request to its model.

Writing bound and unbound expressions

We just learned that an expression is resolved by referencing the specified variable from the binding context. Ember.js goes further and makes these expressions *reactive*. This means that if the underlying variable changes, the replaced expression portion will also be updated. Sometimes, we might not want to suppress this behavior, especially when the variable is too large and constitutes unbound expressions.

These expressions only resolve once on render, and no further changes of the corresponding variable are subscribed. These expressions are written using three braces instead of two, as shown in the following example, where the main content of an Ember.js-powered blog post can be rendered:

```
{{{post}}}
```

Adding comments in templates

Comments in Handlebars have the `{{! ... }}` signature. For example, we can add a documentation that signifies the end of a footer:

```
</footer> {{! end of footer}}
```

These expressions serve the same purpose as normal HTML comments, other than the fact that they are not actually converted to the latter. Therefore, a good reason to use them is when we don't want comments to appear in the rendered output.

Writing conditionals

Handlebars supports the `if`, `if...else`, `unless`, and `unless...else` conditionals. This means that we can render different portions of our templates based on specified conditions. They are block expressions that wrap template portions and usually begin and end with the `{{#` and `{{/` template tags, respectively. For example, if the user has no stored contacts in the `contacts` template of the chapter sample, the application will transition into the `contacts.new` state to force the user to add one. Therefore, we need to display a placeholder string in the now blank left-hand side of the page. We do this by checking whether the passed contact list is indeed empty, as shown in the following code:

```
{{#if model.length}}
...
{{else}}
    <h1>Create contact</h1>
{{/if}}
```

The placeholder element is placed inside the `else` block. As illustrated, the block expression is only met when the value passed evaluates to `True`. Hence, the following values will result in the `else` block being rendered instead:

- `false`
- `undefined`
- `null`
- `[]` (empty array)
- `''`
- `0`
- `NaN`

The `unless` expression, on the other hand, is only met when the variable evaluated is `False`.

Note that Handlebars is logic-less, hence we cannot express conditions using the bitwise operators, as in the following cases:

```
{{#if user.score > 1000 }}
    <span>Level passed.</span>
{{else}}
```

```
    <span>Level failed.</span>
  {{/if}}

  {{#if (temp.high + temp.low)/2 > 100 }}
    <span>It's hot today</span>
  {{/if}}
```

We can, however, define these conditions in the controller layer using computed properties or bindings. For example, the preceding samples can be implemented correctly as:

```
App.ApplicationController = Em.Controller.extend({
  levelPassed: function(){
    return this.get('user.score') > 1000;
  }.property('user.score')
});
```

```
{{#if controller.levelPassed }}
  <span>Level passed.</span>
{{else}}
  <span>Level failed.</span>
{{/if}}
```

```
App.ApplicationController = Em.Controller.extend({
  isHot: function(){
    var temp = this.get('temp');
    return (temp.high + temp.low)/2 > 100;
  }.property('temp.high', 'temp.low')
});
```

```
{{#if controller.isHot }}
  <span>It's hot today</span>
{{/if}}
```

Switching contexts

As previously discussed, Ember.js resolves expressions against the model context. The `{{#with}}...{{/with}}` helper allows us to specify the context to prioritize during the check. A good case is illustrated in the chapter sample, where we need to reuse the form used to create or update contacts. This form is contained in the `contacts/form` template. The only problem is that while the context of the `contacts/new` template is a newly created contact object, the `contact/edit` template has to reference the contact proxied by the contact controller. Thanks to the `with` helper and controller dependencies, we are able to change the context of the latter template as:

```
<script type="text/x-handlebars" data-template-name="contact/edit">
  {{#with controller.controllers.contact}}
  {{partial "contacts/form"}}
  {{/with}}
</script>
```

We will revisit this case when discussing the `partial` helper, but the important thing to note is that the main context is not the corresponding route handler model now, it is rather the contact controller.

Just like the `each` helper, we can create a new context without losing the existing one, as shown in the following examples:

```
<script
  type="text/x-handlebars"
  data-template-name="contact/edit">
  {{#with controller.controllers.contact as contact}}
  {{partial "contacts/form"}}
  {{/with}}
</script>

<script
  type="text/x-handlebars"
  data-template-name="contacts/new">
  {{#with model as contact}}
  {{partial "contacts/form"}}
  {{/with}}
</script>
```

With the two cases, the email field in the form, for example, will now need to bind to the contact context as:

```
{{input type="text" id="form-email" value=contact.email}}
```

Rendering enumerable data

Often, applications will need to display enumerable data that can be accomplished using the `{{#each}} ... {{/each}}` block expression. For example, our `contacts` template used this expression to display the list of contacts on the left as:

```
<ul class="nav nav-pills nav-stacked">
  {{#each model}}
    <li>
      ...{{name}}...
    </li>
  {{/each}}
</ul>
```

We left out the `link-to` expression, which we'll discuss shortly. The `each` block expression switches the working context on each iteration, as discussed in the previous section. If we don't wish to do so, we can specify the name of the current iteration object, as shown in the following reimplementation:

```
<ul class="nav nav-pills nav-stacked">
  {{#each contact in model}}
    <li>
      ...{{contact.name}}...
    </li>
  {{/each}}
</ul>
```

One good thing about this block expression is that we can check if the iterator is empty using the `else` expression. For example, the use of the `else` and `if...else` expressions in our `contacts` template can be reduced to the following:

```
<ul class="nav nav-pills nav-stacked">
  {{#each model}}
    <li>
      ...{{name}}...
    </li>
  {{else}}
    <h1>Create contact</h1>
  {{/each}}
</ul>
```

Writing template bindings

In the *Writing bound and unbound expressions* section, we mentioned that the Ember.js Handlers library enables a variable defined in an expression to subscribe and, hence, updates to the changes of the bound context. The library also enables us to bind these variables to HTML element attributes, including classes using the `{{bind-attr ... }}` helper. In the following example, we define a link whose href property is bound to the provided user profile as:

```
<a {{bind-attr href="profile.link"}}>User Profile</a>
```

By now we all know how the profile context will be provided by the route handler's model hook. For example, if this is the application's template, the corresponding route handler will provide the context as:

```
App.ApplicationRoute = Em.Route.extend({
  model: function(){
    return { profile: {
      link: '@jondoe'
    }}
  }
});
```

The resulting rendered template will then resemble:

```
<a HYPERLINK "mailto:href%3D'@jondoe"href='@jondoe'>User
Profile</a>
```

Every time the profile link changes, the link element's href property will automatically be updated.

We might also wish to toggle states in attributes, for example, the commonly used required and disabled attributes. A common use case is when we want to allow single clicks in e-commerce applications, as shown in the following code:

```
<button {{bind-attr disabled='isCheckedOut'}}>
  Checkout
</button>
```

In the preceding example, when the user clicks on the checkout button, the checkout action should toggle the `isCheckedOut` property, which will result in

the button being disabled. Therefore, attributes can be added or removed from DOM elements if the passed conditions become `True` or `False`, respectively.

Element class names can also be dynamically updated in the same way, with a little difference in the binding behavior. For example, we might wish to add an active property to a clicked link in an application, as shown:

```
<a href="/" {{bind-attr class='selected'}}>Click me</a>
```

When the context's `selected` property evaluates to `active`, the link will be updated to:

```
<a href="/" class='active'>Click me</a>
```

On the other hand, if the property becomes `undefined`, the link changes to:

```
<a href="/">Click me</a>
```

Just like an attribute's presence can be updated dynamically, class names can also be inserted and removed from elements, depending on specified bound conditions. Hence, the preceding example can be reimplemented as:

```
<a href="/" {{bind-attr class='selected:active:inactive'}}> Click me</a>
```

Here, the element's class will be `active` and `inactive` when the context's `selected` property becomes `True` and `False`, respectively.

If only one argument is passed after the semicolon, the passed argument will be used as the class name instead. For example, the following code demonstrates this:

```
<a href="/" {{bind-attr class='isSelected:selected'}}> Click me</a>
```

This yields the following if the context's `isSelected` property becomes `True`:

```
<a href="/" class='selected'>Click me</a>
```

It's also worth noting that camelCase class names get *dasherized*, as shown in the following example:

```
<a href="/" {{bind-attr class='isSelected'}}>Click me</a>
```

This becomes the following:

```
<a href="/" class='is-selected'>Click me</a>
```

Unlike other attributes, we can bind to multiple classes with the same signature, as shown in the following example:

```
<a href="/" {{bind-attr class='isSelected isActive'}}> Click me</a>
```

This becomes the following:

```
<a href="/" class='is-selected is-active'>Click me</a>
```

Sometimes, we might want to use both bound and unbound classes in an element. The following example demonstrates this:

```
<a href="/" {{bind-attr class='isSelected :active'}}> Click me</a>
```

This yields the following:

```
<a href=' ' class='is-selected active'>Click me</a>
```

As shown, the unbound class names begin with a semicolon. Note that the following example won't work since all the class names should be defined inside the bind-attr expression if one of them is bound:

```
<a href="/" class='active' {{bind-attr class='isSelected'}}> Click me</a>
```


Defining route links

A typical Ember.js application has several routes that we might need to link to in templates. The `{{#link-to}}...{{/link-to}}` helper serves this purpose and lets the application create anchors to these routes easily. For example, the list on the left-hand side of our sample application is composed of links that the user can use to view the details of the various contacts. We use this helper to generate these links as:

```
<ul class="nav nav-pills nav-stacked">
  {{#each model}}
  <li>
    {{#link-to "contact.index" this}}{{name}}{{/link-to}}
  </li>
  {{/each}}
</ul>
```

If we inspect one of the generated links, we notice that it resembles the following line of code:

```
<a href="#/contacts/1">Jon Doe</a>
```

The helper takes the route name as the first handler, followed by the resources needed by the corresponding route. As discussed in the previous chapter, since the path of the affected route has dynamic segments, its handler is responsible for resolving the required parameters to replace these segments. In this case, the contact route path has one dynamic segment, `contact_id`, which is used as discussed in the previous chapter.

If we want to link to the blog route, we only need to specify the route name as:

```
{{#link-to "about"}}about{{/link-to}}
```

Just like the `bind-to` expression, the `link-to` expression also accepts other element attributes such as `rel`, `target`, or `class`. The following example opens the link in a new tab or window:

```
{{#link-to "about" target="_blank"}}about{{/link-to}}
```

Registering DOM element event listeners

In vanilla JavaScript, an application script traverses the DOM, setting up event listeners along the way. A typical form might look like the following:

```
<form name='tweet'>
  <textarea name='content' required></textarea>
  <input type='submit' value='tweet'></form>

<script>
  var form = document.forms.tweet;
  form.onsubmit = function(event){
    event.preventDefault();
    alert(form.content.value);
  };
</script>
```

Ember.js provides an abstraction over this, which allows developers to easily subscribe to element-specific events using the `{{action ...}}` helper. Our chapter contains such a form, as shown in the following code:

```
<form
  class="form form-horizontal"
  role="form"
  {{action "saveContact" this on="submit"}}>

  ...

  <button class="btn" type="submit">
    done
  </button>

  ...
</form>
```

The preceding snippet shows an action that gets triggered when the form gets submitted. As shown, the action helper takes the following arguments:

- A function to trigger
- An optional context object
- An optional type of event to listen to which defaults to click on

The function to call is usually defined in the `actions` property of the route and

takes an unlimited number of arguments. By default, the default event type to bind to is usually the `click` event. However, you can specify this type using the `on` attribute. The `actions` property can be defined on either the corresponding route or controller. For example, the action handler for the preceding action was defined in the `contacts` route as:

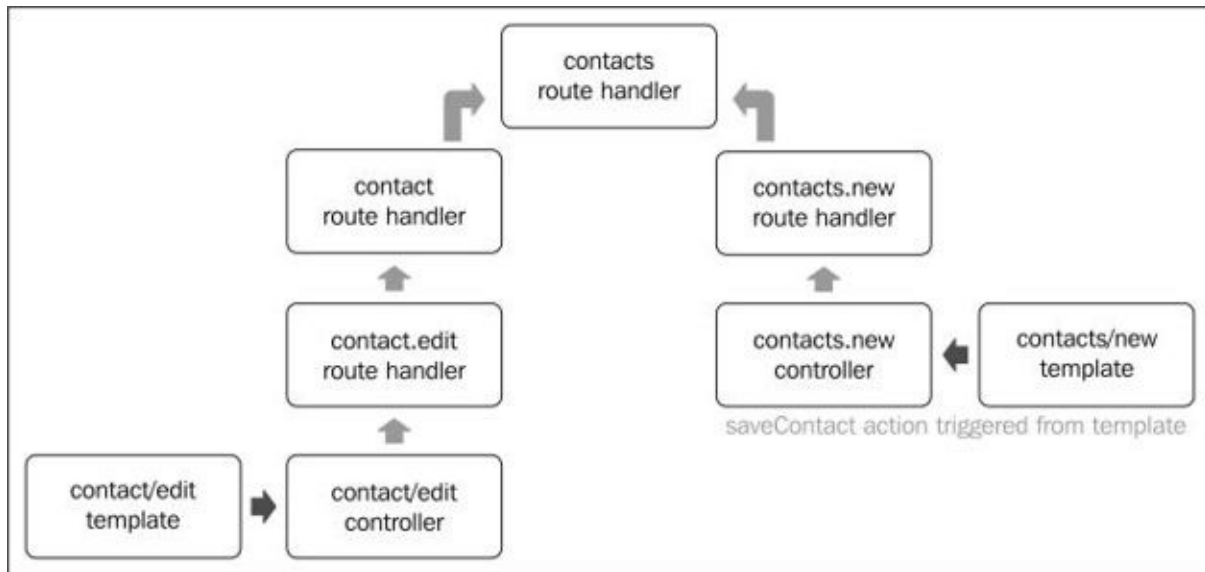
```
App.ContactsRoute = Ember.Route.extend({
  actions: {
    saveContact: function(contact){
      var id = contact.get('id');
      contact.save();
      if (!id){
        this.controllerFor('contacts').pushObject(contact);
      }
      this.transitionTo('contact.index', contact);
    }
  }
});
```

As mentioned, this action can still be defined in the controller layer as:

```
App.ContactsController = Ember.ArrayController.extend({
  actions: {
    saveContact: function(contact){
      var id = contact.get('id');
      contact.save();
      if (!id){
        this.pushObject(contact);
      }
      this.transitionToRoute('contact.index', contact);
    }
  }
});
```

The main point to consider when deciding where to put an action is when we need to take advantage of the **bubbling** action. When an action is triggered, the function specified is looked up in the corresponding controller. If this action is defined, it gets executed. If the action is not defined, Ember.js performs the check in the corresponding route handler. If this action is defined in either controller or route handler and returns a value equal to `True`, Ember.js goes ahead and checks for a similar function in the parent route handlers, until one of them doesn't contain the function or does not define it.

One important thing to note is that the bubbling action occurs in the route handler layer only. This is one of the reasons we will opt to define the function in the route handler. For example, since we should be able to call the `saveContact` action from both the `contacts.new` and `contact.edit` templates, we define it in the `contacts` route. Here's an illustration of how the action is propagated in both cases:



The preceding action function can still be located in any controller if we specify the target object of the action. The target object contains the actions hash that Ember.js checks for the action function. By default, it is usually the corresponding controller of the template, as shown in the preceding figure. Therefore, we can define the action function in the `contact` controller, and then set the target as:

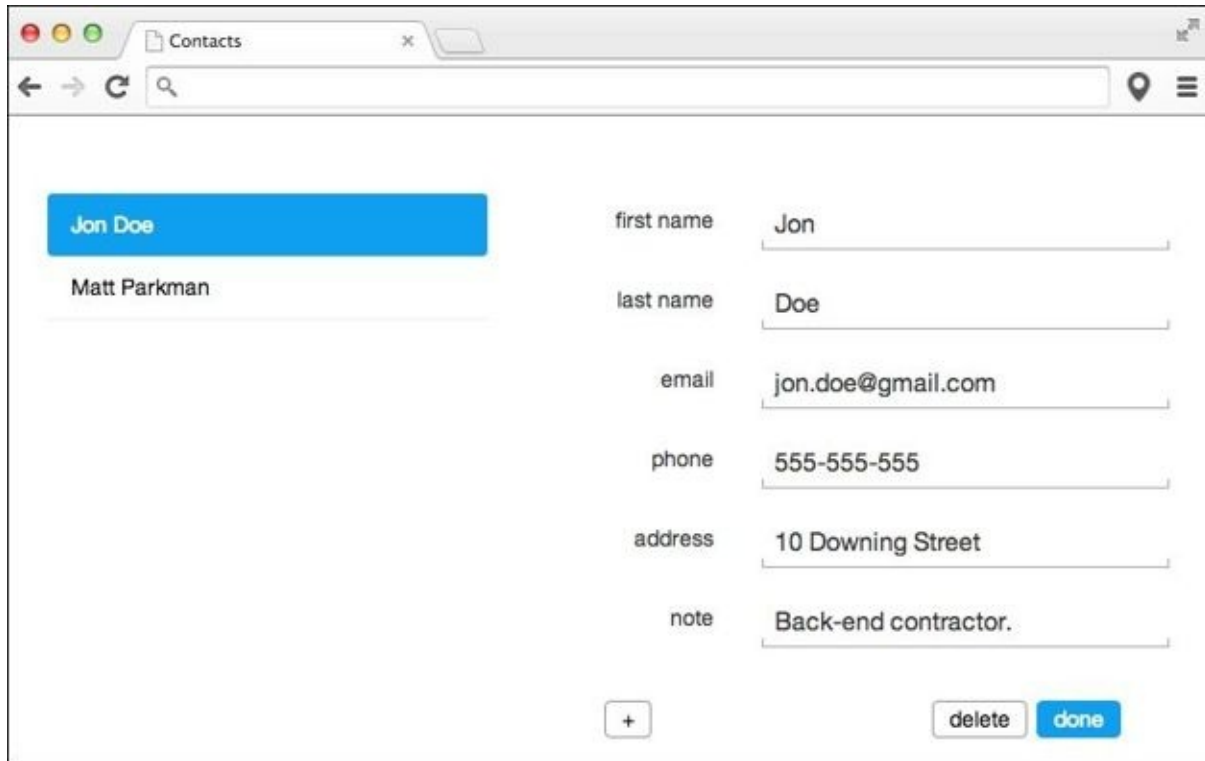
```
<form
  class="form form-horizontal"
  role="form"
  {{action "saveContact" this target="controllers.contact"
on="submit"}}>
  ...
</form>
```

Writing form inputs

Writing forms is a common practice that Ember.js simplifies by providing template helpers of the many HTML5 form controls. The following table shows these common controls and the attributes they can accept:

Control	Attributes
input	value size name pattern placeholder disabled maxlength tabindex
textarea	checked disabled tabindex indeterminate name
checkbox	rows cols placeholder disabled maxlength tabindex

In the sample application, the user is able to edit a contact by clicking on the **edit** button in the footer, as shown in the following screenshot:



If the user edits the first name of the contact, we notice that their names are updated in the sidebar. The input helper is used in the `contacts/form` template to *bind* the input element's value with the first name of the contact as:

```
{{input
  type="text"
  id="form-first-name"
  value=first_name
  required='required'}}
```

Here, we defined four attributes that were either bound or unbound. A bound attribute gets updated each time the specified variable changes and vice versa. For example, in the preceding case, the `value` attribute was bound while the `required` attribute was not. Unbound attributes are quoted while those bound are not.

Here are more examples that show how to use the other two form helpers:

```
{{textarea value=model.content}}
{{checkbox checked=model.isPaid}}
```

Extending templates

In the course of application development, you might find the need to abstract templates for reuse. There are several helpers that can help us implement this easily:

- `partial`
- `view`
- `render`
- `named outlets`

The `partial` helper is used to include templates inside others. It simply inserts the desired template where the `partial` expression has been specified. As discussed earlier, the chapter sample used this helper in two instances:

```
<script
  type="text/x-handlebars"
  data-template-name="contact/edit">
  {{#with controller.controllers.contact}}
    {{partial "contacts/form"}}
  {{/with}}
</script>
```

```
<script
  type="text/x-handlebars"
  data-template-name="contacts/new">
  {{partial "contacts/form"}}

</script>
```

This helper takes the template that should be inserted into the current template as the only argument. One thing to note is that using the helper doesn't lead to the loss of context, as seen in the `contact/edit` case.

We might also wish to insert views inside other templates. In this case, the view's template will be inserted into the specified portion of the current template and the defined event listeners will be set up. For example, the first name input we saw earlier can also be written as:

```
{{view
  Em.TextField
  id="form-first-name"}}
```

```

    value=first_name
    required='required'
  }}

```

We will discuss this in more detail in [Chapter 6, Views and Event Management](#), where we will deal with views. The important thing to note here is that the input helpers named are actually Handlebars helpers defined from these views. We will discuss how these helpers are created in a moment. The render helper works in the same way as the partial helper, except that it takes an optional context as the second argument. For example, we will define the contact/edit and contact/edit templates as:

```

<script
  type="text/x-handlebars"
  data-template-name="contact/edit">
  {{render "contacts/form" controller.controllers.contact}}
</script>

<script
  type="text/x-handlebars"
  data-template-name="contacts/new">
  {{render "contacts/form"}}
</script>

```

Instead of switching the context in the first template, we simply passed the controller to be used as the context. Note that, by default, the context passed is the corresponding controller instance, and so we did not need to specify this context in the case of the contacts/new template.

The last way to extend templates is by the use of named outlets we already discussed in the preceding chapter. Here is the example we used:

```

<script type="text/x-handlebars" data-template-name="game">
  <div id="leaderboard">{{outlet leaderboard}}</div>
  <div id="mainboard">{{outlet mainboard}}</div>
</script>

```

We then render the outlets via the handler as:

```

App.GameRoute = Ember.Route.extend({
  renderTemplate: function() {
    this.render('mainboard', {
      into: 'game',

```



```
        outlet: 'mainboard',
        controller: 'mainboard'
    });
    this.render('leaderboard', {
        into: 'game',
        outlet: 'leaderboard',
        controller: 'leaderboard'
    });
    }
});
```

It's very similar to how the `partial` helper is used, but in this case, we also specify the template in the route handler's `renderTemplate` hook.

Defining custom helpers

Handlebars provides ways to create your own helpers. The following is the format used to register new helpers:

```
Ember.Handlebars.register(helper_name, helper_function_or_class);
```

For example, let's create a heading helper that creates h1 tags:

```
Ember.Handlebars.register('heading', function(text, options){
  var escapedText = Handlebars.Utils.escapeExpression(text);
  var heading = '<h1>'+escapedText+'</h1>';
  return new Handlebars.SafeString(heading);
});
```

This can then be used in our application templates as:

```
{{heading 'Title'}}
```

Now, yield the following:

```
<h1>Title</h1>
```

The following example also demonstrates how to create helpers from existing views:

```
Ember.Handlebars.register('loader', App.LoaderView)}
```

This can now be used simply as:

```
{{loader}}
```

This is equivalent to the following:

```
{{view App.LoaderView}}
```

Creating subexpressions

A subexpression, as the name suggests, is an expression contained in another expression, which takes the following signature:

```
{{outer-helper (inner-helper 'arg1') 'arg2'}}
```

Here's an example that implements a `Number.toFixed` helper:

```
Ember.Handlebars.registerHelper('to-fixed', function(num, decimals)
{
  return new Ember.Handlebars.SafeString(
    num.toFixed(decimals)
  );
});
```

This can be used inside a `link-to` helper as:

```
{{#link-to 'checkout' (to-fixed cart.total)}}
checkout
{{/link-to}}
```

This will result in something like the following:

```
<a href='#/checkout/10.10'>checkout</a>
```

Summary

This has been an exciting chapter that walked us through the template layer. This chapter will be revisited when we will discuss controllers in the next chapter. Here are the key concepts we learned and those that will be revisited in the next chapters:

- Writing binding template expressions
- Writing conditionals in templates
- Changing contexts in templates
- Creating event listeners in templates
- Extending templates
- Writing custom template helpers

Chapter 5. Controllers

In the previous chapter, we discussed how templates in Ember.js are used to present data to users. We also covered how user interactions, in our applications, are easily made possible through these templates. We noted that templates serve their purpose by communicating with controllers. This chapter will expound on this, and will cover the following topics:

- Defining controllers
- Storing models and objects in controllers
- Using object and array controllers
- Specifying controller dependencies
- Registering action handlers in controllers
- State transitions in controllers

Defining controllers

Just like route handlers, a controller can be defined by extending the `Ember.Controller` class, as shown in the following line of code:

```
AppNamespace.ControllernameController = Ember.Controller.extend();
```

A defined controller can further be extended to create yet another new controller class:

```
App.TweetsController = Ember.Controller.extend();  
App.RetweetsController = App.TweetsController.extend();
```

These controller classes can then be instantiated with the `create` method, as shown in the following examples:

```
var controller = Ember.Controller.create();  
var tweetsController = App.TweetsController.create();
```

Just like objects, if we need to use mixins when instantiating controllers, we need to use the `createWithMixins` method instead:

```
var mixin = Ember.Mixin.create({  
  model: [1, 2, 3]  
});  
var controller = Ember.Controller.createWithMixins(mixin);
```

This is equivalent to the following:

```
var Controller = Ember.Controller.extend({  
  model: [1, 2, 3]  
});  
var controller = Controller.create();
```

We rarely instantiate application controllers ourselves because Ember.js does it for us when needed.

Providing controllers with models

Before we proceed, let's recap and see how data is loaded and stored in controllers. Most of the applications we'll build will communicate with REST endpoints, and therefore, Ember.js comes with features that make the creation of such applications trivial. In [Chapter 3, Routing and State Management](#), we learned that data can be loaded from the server in an asynchronous fashion via a route handler's `model` hook. For example, let's define a blog post route that loads a particular blog post from our server. First, we'll define our application's router as:

```
App.Router.map(function(){
  this.resource('posts', function(){
  });
  this.resource('post', {path: '/post/:post_id'});
});
```

We just defined a post resource that will handle requests to a post's detail page, as shown in the following line of code:

```
this.resource('post', {path: '/post/:post_id'});
```

If a user visits a post's path, say `/post/100`, Ember.js expects that the post route handler will define a `model` hook that will load the matching post from the server. Here's an example that illustrates this using jQuery:

```
App.PostRoute = Ember.Route.extend({
  model: function(params){
    // load matching post from server
    return Ember.$.getJSON('/posts/'+params.post_id);
  }
});
```

In the preceding example, the handler's `model` hook took an options object that contained the post's ID. This ID was then used to load the matching post from the server using jQuery's `getJSON()` method that returned a promise with which our application resolved on load. Once resolved, Ember.js expected this route to define a `setupController` hook that stored the resolved post into the corresponding controller. This is the default behavior that is implemented as the following code:

```
App.PostRoute = Ember.Route.extend({
  model: function(params){
    return Ember.$.getJSON('/posts/'+params.post_id);
  },
  setupController: function(controller, model){
    controller.set('model', model);
  }
});
```

The `setupController` hook receives two arguments: an instance of the corresponding controller and the resolved model. This model is stored inside the controller's `model` property. Note that this is just the default implementation; we can store the model in any other desired property or controller.

Rendering dynamic data from controllers

After loading data from the server, the controller's purpose is to make this model available to the corresponding template for display. These templates will then register bindings to the properties of the provided model and send updates of changes made to these properties using form controls. Since controllers are an extension of `Ember.Object`, they realize better management of the evented nature of browser environments using the following:

- Properties
- Computed properties
- Observables

Properties

Templates can display properties of bound controllers using expressions. For example, the post template in the previous example will display the loaded post as:

```
{{! post.hbs}}  
  
<h1>{{model.title}}</h1>  
<p>{{model.body}}</p>
```

When the post loads, the rendered post template will resemble the following:

```
<h1>Introduction to Ember.js.</h1>  
<p>A gentle introduction to Ember.js.</p>
```

If the post's title changes at a later point in time, the title portion of the template will be rerendered to display the new title, as shown:

```
controller.set(model.title, 'A guide to using Ember.js.');
```

Templates can also push updates back to controllers. This is typically done using HTML-form elements. Ember provides Handlebars expressions that abstract the use of these controls to create two-way bindings, as we discussed in the previous chapter. To illustrate this, let's add a new route in our blog application, which will enable the blog's admin to add a new post entry, as shown in the following code:

```
this.resource('posts', function(){  
  this.route('/new'); // route to add a new post  
});
```

The admin will of course create the new post on the `/posts/new` page. Ember.js will expect a `posts/new` template for this route, which will resemble the following code:

```
{{! posts/new template }}  
  
<form>  
  {{input name='title' value=model.title}}  
  {{textarea name='body' value=model.body}}  
  <button type='submit'>Save post</button>
```

</form>

The `PostsNewRoute` handler will also need to provide the model for the template that will serve as its context. As you might have guessed, its `model` hook will return a new post object to be updated, as shown in the following example:

```
App.PostsNewRoute = Ember.Route.extend({  
  model: function(params){  
    return Ember.Object.create();  
  }  
});
```

The `model` hook returns an Ember.js object that will serve as the post to be created by the admin. Since this is common practice, we'll learn how to use `ember-data`, a higher level library that helps in the definition and creation of such models in a later chapter. Any update to both text controls will update the new post, thanks to the two-way bindings.

Computed properties

Computed properties are functions that evaluate to properties and depend on other properties. We can use computed properties to create states and properties that depend on other properties. This is especially helpful when we want to obtain states through aggregation or any form of map reduce. Here's a use case example:

```
{%! application template }}
{{input name="firstName" value=model.firstName}}
}
{{input name="lastName" value=model.lastName}}
Full name: {{fullName}}

// application controller

App.ApplicationController = Em.Controller.extend({
  fullName: function(){
    return this.get('model.firstName') + ' ' +
    this.get('model.lastName');
  }.property('model.firstName', 'model.lastName')
});

// application route
App.ApplicationRoute = Em.Route.extend({
  setupController: function(controller){
    controller.set('model', Em.Object.create({
      firstName: 'Jon',
      lastName: 'Doe',
    }));
  });
});
```

In this example, the template is able to display the user's `fullName` based on their first and last names. We already mentioned that such an implementation is impossible to accomplish in the template layer alone, as shown in the following example:

```
Full name: {{firstName+lastName}} // wrong
```

One thing to note is that instances of objects are never set on a class definition unless they're meant to be static. For example, if the template is a form that updates the values of the model, we might be tempted to provide the default

model of the controller as:

```
App.ApplicationController = Em.Controller.extend({
  model: Em.Object.create({
    firstName: 'Jon',
    lastName: 'Doe',
  }),

  fullName: function(){
    return this.get('firstName') + ' ' + this.get('lastName');
  }.property('firstName', 'lastName')
});
```

The preceding implementation can lead to updates not being isolated as intended. Here's another example of a search feature that can be added in our blog application:

```
{{! search template}}

{{input name="search" value=queryTerm}}
<ul>
  {{#each results}}
    <li>{{user}}: {{body}}</li>
  {{/each}}
</ul>

// search controller
App.SearchController = Em.Controller.extend({

  // require posts controller
  needs: ['posts'],

  // compute matching posts
  results: function(){

    var queryTerm = this.get('queryTerm');
    if (!queryTerm && queryTerm.trim() === '') return;

    return this
      .get('controllers.posts.model')
      .filter(function(){
        return post.match(queryTerm);
      });

  }.property('queryTerm')
```

```
});
```

This example contains some features that we already discussed. If a user visits the search page at say/search, they will be presented with a search input that automatically updates the search controller's `queryTerm` property on input. As you might notice, this controller's `results` property will be recalculated because it depends on the controller's `queryTerm` property.

One feature that we introduced is the ability of the controller to reference other controllers. We will discuss this in a later section, but the important thing to note is that we are able to generate results by filtering the model of the `posts` controller. The search template automatically redisplay results as the user types along. This example demonstrates how trivial it is to add such seemingly difficult features in single-page applications. Here are a few other features that you can try adding into the application:

- A spinner that shows up new posts every time are being loaded from the server. Now is a good time to revisit the loading-and-error action hooks we discussed in [Chapter 3, Routing and State Management](#).
- Define a `humanizedDate` computed property for each of the loaded posts. Let this property return the post's date in a readable format such as Mon, 15th. Moment.js (<http://momentjs.com>) can come in handy.

Observables

In addition to computed properties, we also learned how to use observables. These are functions that react to changes made on other properties. For example, let's make the search functionality previously mentioned more user friendly. Most users expect the search request to kick off after a second or two, after they've stopped typing the query term. We, therefore, need a way to *debounce* this search. Ember.js provides a function that serves this purpose and can be referenced as `Ember.run.debounce`. Here's a possible implementation:

```
App.SearchController = Em.Controller.extend({
  needs: ['posts'],

  queryTermDidChange: function(){
    Em.run.debounce(this, this.searchResults, 1000);
  }.observes('queryTerm'),
  searchResults: function(){
    var queryTerm = this.get('queryTerm');
    if (!queryTerm && queryTerm.trim() === '') return;
    var results = this
      .get('controllers.posts.model')
      .filter(function(){
        return post.match(queryTerm);
      });
    this.set('results', results);
  }
});
```

The controller defines an observer, `queryTermDidChange`, which invokes the search function after only a second of typing. As illustrated, the `debounce` function (http://emberjs.com/api/classes/Ember.run.html#method_debounce) takes three arguments—a context, a function to invoke within the specified context, and the time to wait for invocation if no other calls are made.

Object and array controllers

Ember.js ships with the following controllers that are meant to easily represent objects and enumerable data, respectively:

- Object controller
- Array controller

These controllers are a bit different from the other controllers in the sense that the data being represented is usually set as the `model` property of the controller, for example:

```
Ember.ObjectController.create({  
  model: {  
    name: 'Jon Doe'  
    age: 23  
  }  
});
```

```
Ember.ArrayController.create({  
  model: [1, 2]  
});
```


An object controller

An object controller is used to proxy properties of the object being represented. This means that if a controller property is accessed, Ember.js will look for the property first in the controller, and then the model. For example, let's create a post model class:

```
App.Post = Ember.Object.extend({
  title: null,
  body: null
});
```

Then, create a new post from this model as:

```
var post = App.Post.create({
  title: 'JavaScript prototypes.',
  body: 'This post will discuss JavaScript prototypes.'
});
```

This post's properties can obviously be accessed using the object's getter and setter methods as:

```
post.set('title', 'Design patterns.');
```

```
post.get('title'); // Design patterns.
```

When this post is set as the model of an `ObjectController` instance, as shown, access to the controller properties translates to access to the post, for example:

```
var postController = Ember.ObjectController.create();
postController.set('model', post);
postController.get('title'); // Design patterns.
```

Note that using a normal controller will not yield the same results:

```
var postController = Ember.Controller.create();
postController.set('model', post);
postController.get('title'); // undefined
postController.get('model.title'); // Design patterns.
```

Object properties are useful when we wish to create computed properties on the controller whose dependent properties are those of the model. For example, let's pass the preceding defined post as a route's model:

```
App.PostRoute = Ember.Route.extend({
  model: function(){
    return App.Post.create({
      title: 'JavaScript prototypes.',
      body: 'This post will discuss JavaScript prototypes.',
      tagIds: [1, 2, 3, 4]
    });
  }
});
```

Now, suppose we want to compute a tags property based on the given array of IDs, we will implement this in the corresponding controller as:

```
App.PostController = Ember.ObjectController.extend({

  TAGS: {
    1: 'ember.js',
    2: 'javascript',
    3: 'web',
    4: 'mvc'
  },

  tags: function(){
    var tags = this.get('TAGS');
    return this.get('tagIds').map(function(id){
      return tags[id];
    });
  }.property('TAGS', 'tags.length')

});
```

With the computed property defined, we can go ahead and use it in the post template as:

```
<p>{{title}}</p>
<p>{{body}}</p>
<ul>
  {{#each tags}}
    <li>{{this}}</li>
  {{/each}}
</ul>
```

This will yield something like the following:

```
<p>JavaScript prototypes.</p>
<p>This post will discuss JavaScript prototypes.</p>
```

```
<ul>
  <li>ember.js</li>
  <li>javascript</li>
  <li>web</li>
  <li>mvc</li>
</ul>
```

Note that we do not have to prefix the variables with `model.`, as we did in the preceding sections, because the template's context, the controller, forwarded these requests to the model. This type of controller uses the `Ember.ObjectProxy` (<http://emberjs.com/api/classes/Ember.ObjectProxy.html>) mixin, which enables a proxy (in this case, the controller) to forward all requests to the properties that it has not defined, and to its model, as we already discussed.

An array controller

Likewise, array controllers are used to represent enumerable data. An example of enumerable data is the JavaScript Array primitive:

```
var controller = Ember.ArrayController.create({  
  model: [1, 2, 3]  
});  
controller.get('length'); // 3
```

In this case, the corresponding template will list the items as:

```
{{#each}}  
  {{this}}  
{{/each}}
```

Note that we also did not need to reference the model, as shown in the following cases:

```
{{#each model}}  
  {{this}}  
{{/each}}
```

```
{{#each controller.model}}  
  {{this}}  
{{/each}}
```

Since array controllers represent enumerable data, they provide the following useful methods that can be used to manipulate their models.

addObject(object)

The `addObject(object)` method adds the given object to the end of the controller model if the latter does not contain the former, as shown in the following example:

```
var controller = Ember.ArrayController.create({
  model: []
});
controller.addObject('a'); // ['a']
controller.addObject('b'); // ['a', 'b']
controller.addObject('b'); // ['a', 'b'] // already added
```

If the model already contains the object, this method call fails silently.

pushObject(object)

The `pushObject(object)` method always adds the object, regardless of whether the model contains it or not, for example:

```
var controller = Ember.ArrayController.create({
  model: []
});
controller.pushObject('a'); // ['a']
controller.pushObject('b'); // ['a', 'b']
controller.pushObject('b'); // ['a', 'b', 'b'] // still added
```

removeObject(object)

The `removeObject(object)` method is used to remove the given object from the controller's model, as shown in the following example:

```
var controller = Ember.ArrayController.create({
  model: ['a', 'b', 'c']
});
controller.removeObject('a');    // ['b', 'c']
controller.removeObject('b');    // ['c']
controller.removeObject('b');    // ['c'] // fails silently
```

This method also does nothing if the model doesn't contain the object.

addObjects(objects), pushObjects(objects), and removeObjects(objects)

The three methods mentioned previously are used to perform the three methods we just discussed using multiple objects, for example:

```
var controller = Ember.ArrayController.create({
  model: []
});
controller.pushObjects(['a', 'b']);           // ['a', 'b']
controller.addObjects(['b', 'c']);           // ['a', 'b', 'c']
controller.removeObjects(['b', 'c']); // ['a']
```


contains(object)

To check if a model contains an object, we can use the `contains(object)` method that returns a Boolean:

```
var controller = Ember.ArrayController.create({  
  model: ['a', 'c']  
});  
controller.contains('a');    // true  
controller.contains('b');    // false
```

compact()

The `compact()` method returns a copy of the underlying model, with undefined and null items removed:

```
var controller = Ember.ArrayController.create({  
  model: ['a', 'b', 'c', undefined, null]  
});  
controller.compact();    // ['a', 'b', 'c']
```

every(callback)

The `every(callback)` method is used to check if each of the items contained in the model satisfies a given condition:

```
var areEven = [2, 2, 4, 24, 80].every(condition); // true
var areEven = [1, 2, 3, 4, 5].every(condition);    // false

function condition(integer){
  return integer %2 === 0;
}
```

filter(object)

Filter works in the same as way as the native JavaScript array object, `Array.filter`:

```
[2, 2, 4, 24, 80].filter(condition); // [2, 2, 4, 24, 80]  
[1, 2, 3, 4, 5].filter(condition);   // [2, 4]
```

```
function condition(integer){  
    return integer %2 === 0;  
}
```

filterBy(property)

Sometimes, we want to compact a model, but only if the items contained define the given property. We can use the preceding `filter` method, as shown:

```
var colors = [
  { name: 'red', isPrimary: true },
  { name: 'green', isPrimary: false },
  { name: 'black', isPrimary: undefined },
  { name: 'white', isPrimary: null },
];
colors.filter(condition);      // [{ name: 'red', isPrimary: true }]
function condition(color){
  return !!color.isPrimary;
}
```

We can also use a shorter version, as shown:

```
var colors = [
  { name: 'red', isPrimary: true },
  { name: 'green', isPrimary: false },
  { name: 'black', isPrimary: undefined },
  { name: 'white', isPrimary: null },
];
colors.filterBy('isPrimary');  // [{ name: 'red', isPrimary:
true }]
```

find(callback)

In the preceding example, we can use the `filter` method to return the first occurrence of a primary color, as shown:

```
var colors = [
  { name: 'red', isPrimary: true },
  { name: 'green', isPrimary: false },
  { name: 'black', isPrimary: undefined }
];
colors.filter(condition)[0];    // { name: 'red', isPrimary: true }
function condition(color){
  return !!color.isPrimary;
}
```

This is inefficient because we always loop through all the model items. The `find` method can be used to achieve this need as:

```
var colors = [
  { name: 'red', isPrimary: true },
  { name: 'green', isPrimary: false },
  { name: 'black', isPrimary: undefined }
];
colors.find(condition);        // { name: 'red', isPrimary: true }
function condition(color){
  return !!color.isPrimary;
}
```

As soon as a match is found, the check iteration is aborted.

findBy(key, value)

Just as in the `filter` versus `filterBy` case, we can reimplement the preceding example using the `findBy` method instead of `find`, as shown:

```
var colors = [  
  { name: 'red', isPrimary: true },  
  { name: 'green', isPrimary: false },  
  { name: 'black', isPrimary: undefined }  
];  
colors.findBy('isPrimary', true);    // { name: 'red', isPrimary:  
true }
```

insertAt(index, object), objectAt(index), and removeAt(index, length)

The `insertAt(index, object)`, `objectAt(index)`, and `removeAt(index, length)` methods are used to perform operations using item indices. The first method is used to add an object at the given index. An error is thrown if the index is out of bounds. The second method is used to retrieve an object at the specified index. Again, if the index is out of bounds, an undefined value is returned.

Note that we cannot use negative indices for lookups, as shown in the following example:

```
var colors = ['red', 'blue'];
colors.insertAt(1, 'yellow'); // ['red', 'yellow', 'blue'];
colors.insertAt(10, 'green'); // Error: Index out of range
colors.objectAt(0); // 'red'
colors.objectAt(10); // undefined
colors.objectAt(-1); // undefined - negative index
```

The last method removes objects matching the given index by an optional range:

```
var fruits = ['mango', 'apple', 'banana', 'orange', 'papaya',
'lemon'];
fruits.removeAt(0); // [apple, 'banana', 'orange', 'papaya',
'lemon'];
fruits.removeAt(1, 2); [apple, 'papaya', 'lemon'];
fruits.removeAt (10, 3); // Error: Index out of range
```


map(callback)

A map works in the same way as `Array.map`:

```
var fruits = ['mango', 'apple', 'banana', 'orange', 'papaya',  
  'lemon'];  
fruits.map(function(fruit){  
  return {name: fruit};  
});  
// [  
//   { name: 'mango'},  
//   { name: 'apple'},  
//   { name: 'banana'},  
//   { name: 'orange'},  
//   { name: 'papaya'},  
//   { name: 'lemon'}  
// ];
```

mapBy(property)

With the result generated in the preceding example, we can use the `mapBy` method to get back the original array as:

```
var fruits = [  
  { name: 'mango'},  
  { name: 'apple'},  
  { name: 'banana'},  
  { name: 'orange'},  
  { name: 'papaya'},  
  { name: 'lemon'}  
];  
fruits.mapBy('name');  
// ['mango', 'apple', 'banana', 'orange', 'papaya', 'lemon'];
```

As illustrated, this method returns a new array containing the values evaluated on the model items.

forEach(function)

This is a commonly used method that invokes the given function on each of the items contained in the model:

```
var Dog = Em.Object.extend({
  bark: function(){
    console.log('woof: %s', this.get('name'));
  }
});
// model
var dogs = ['bo', 'sunny'].map(function(name){
  return Dog.create({
    name: name
  });
});
dogs.forEach(function(dog){
  dog.bark();
});
// woof: bo
// woof: sunny
```

uniq()

As the name suggests, the `uniq()` method returns a new array devoid of duplicates:

```
['papaya', 'apple', 'banana', 'orange', 'papaya', 'apple'].uniq();  
// ["papaya", "apple", "banana", "orange"]
```

sortProperties and sortAscending

The `sortProperties` and `sortAscending` methods are used to sort the represented data. For example, we might have a music catalog that we want to sort by album name, and later by song name, as shown in the following table:

Album name	Song name
Folie a deux	Tiffany Blews
Folie a deux	W.A.M.S
Infinity on high	Thriller

To accomplish this, we need to define the following properties in the controller:

- `sortProperties`
- `sortAscending`

The first property specifies the properties to use when ordering the items, while the second property specifies the sort direction. In our cases, we will sort the music catalog as:

```
var controller = Ember.ArrayController.create({
  model: [
    {name: 'W.A.M.S', album: 'Folie a deux'}, {name: 'Thriller',
album: 'Infinity on high'}, {name: 'Tiffany blues', album: 'Folie
a deux'},
  ],
  sortProperties: ['name', 'album'],
  sortAscending: true
});
```

To sort the songs in the reverse order, we need to set the `sortAscending` property as `False`.

These are just a few of the common methods that are provided by the `Ember.ArrayProxy` (<http://emberjs.com/api/classes/Ember.ArrayProxy.html>) mixin, which `Ember.ArrayController` uses.

Handling event actions

In the previous chapter, we learned how user actions can easily be delegated to controllers and routes from templates. Let's have a recap with an example:

```
{{! posts/new template }}  
  
<form {{action 'save' model on='submit'}}>  
  {{input name='title' value=title}}  
  {{textarea name='body' value=body}}  
  <button type='submit'>Create</button>  
  <button type='cancel' {{action 'cancel' this}}>Cancel</button>  
</form>
```

In this example, we defined two actions that will be handled by the corresponding controller as:

```
App.PostsNewController = Ember.ObjectController.extend({  
  actions: {  
    save: function(post){  
      post.save();  
    },  
    cancel: function(post){  
      post.rollback();  
    }  
  }  
});
```

We already learned that all action handlers are defined in the `actions` property of the target controller or route. In this case, when the user submits the form either by clicking on the **submit** button or by hitting the *Enter* key, the `save` hook in the controller is called with the `post` context as the only argument. Likewise, clicking on the **cancel** button calls the corresponding `cancel` hook.

A typical widget consisting of tabs can be implemented in the same way, as shown in the following screenshot:



In this case, the widget template might look like the following code:

```
<ul class="tabs">
  {{#each}}
    <li {{bind-attr selected="selected"}}
      {{action "selected" this}}>
        {{name}}
      </li>
    {{/each}}
</ul>
```

This template contains the tabs element group generated from the tabs context property. If any of the tabs are clicked on, they will need to acquire a selected class. Here's suitable styling that will achieve this effect:

```
.tabs .selected{
  color: deepskyblue
}
```

The selected action handler for the controls will then be implemented as:

```
App.TabsRoute = Ember.Route.extend({
  model: function(){
    return [{
      name: 'tab 1',
      body: 'tab 1 content'
    }, {
      name: 'tab 2',
      body: 'tab 2 content'
    }
  ];
});

App.TabsController = Ember.ArrayController.extend({
  actions: {
```

```
selected: function(selectedTab){  
  this.forEach(function(tab){  
    var selected = tab.get('name') === selectedTab.get('name');  
    tab.set('selected', selected);  
  });  
}  
});
```

Note that these actions don't have to be caught in the context controller as we discussed in the previous chapter. When an action is triggered, typically from a template element, Ember.js checks to see if the appropriate action handler is defined inside the actions property of the immediate controller. If this is not the case, Ember.js proceeds to search for the action handler in the corresponding route. If this route still doesn't implement the handler, Ember.js will continue searching for the action handler in higher routes. One thing to note is that if an action handler returns `True`, Ember.js will still continue to search for this handler, constituting **action bubbling**.

Specifying controller dependencies

Controller dependencies enable controllers to associate. Therefore, whenever a controller needs to access the properties of another controller, it should first declare the controller as a dependency in order for it to be able to do so. These dependencies are defined in the `needs` property of the affected controller. For example, let's say we decided to add a commenting system to our blog application:

```
this.resource('post', {path: '/post/:post_id'}, function(){
  this.resource('comments', function(){
  });
});
```

In a typical blog, comments are usually displayed on a separate page, and in our case, at a page with a path such as `/post/100/comments`. We will need to define a comments template that lists the loaded comments as:

```
{{! comments template}}
<h1> Comments for <h1>
<ul>
  {{#each comments}}
    <li>{{user}}: {{body}}</li>
  {{/each}}
</ul>
```

As you might have noticed, the template needs to display the title of the comments' post. To do this, it needs to be able to access a loaded post in the post controller. By specifying a dependency to the post controller, the comments controller will be able to access the post controller in its `controllers` object property. For example:

```
App.CommentsController = Ember.Controller.extend({
  needs: ['post']
});
```

Then, the comments template will be updated to the following:

```
{{! comments template}}
<h1>Comment listing for {{controller.controllers.post.title}}</h1>
<ul>
  {{#each comments}}
    <li>{{user}}: {{body}}</li>
```

```
    {{/each}}  
</ul>
```

You might be wondering whether this can lead to an infinite dependency loop. Well, controllers can depend on each other without suffering from this fate:

```
App.AController = Ember.Controller.extend({  
  needs: ['b']  
});  
App.BController = Ember.Controller.extend({  
  needs: ['a']  
});
```

This association serves as the correct channel of communication between the different components of the application.

State transitions in controllers

In [Chapter 3](#), *Routing and State Management*, we learned that routes can transition the state of an application into other routes by invoking their `transitionTo` method, as shown in the following code:

```
App.IndexRoute = Ember.Route.extend({
  redirect: function(){
    this.transitionTo('posts');
  }
});
```

Likewise, controllers also have this capability through the use of the provided `transitionToRoute` method. For example, we can change states in a controller's action handler as:

```
App.PostsNewController = Ember.ObjectController.extend({
  actions: {
    cancel: function(post){
      this.transitionToRoute('posts');
    }
  }
});
```

Summary

This has been an exciting chapter that helped us understand the primary purpose of controllers, which is data representation. We learned how controllers are defined based on the defined application routes. We also learned how to use object and array controllers to represent models. Lastly, we learned how to set up dependencies between controllers, which might handle different concerns of the application. At this point of the book, we really should be ready to start thinking of ways to architect Ember.js applications. The next chapter will cover the view layer, for which a good amount of knowledge in the use of controllers will be required.

Chapter 6. Views and Event Management

We discussed templates and controllers in the previous two chapters. We noted that controllers present models that templates render to users. We also learned that when users interact with applications, templates usually propagate these events back to controllers using action template helpers. In reality, these action expressions are views that delegate the events initially to controllers, and later to routes. Therefore, in this chapter, we will learn how to integrate views right into templates, especially when the following application needs arise:

- A section of the application requires sophisticated event management
- There's a need to build reusable components
- The application needs to integrate third-party libraries

Therefore, by the end of this chapter, you should be able to:

- Define a view
- Create a view instance
- Customize a view
- Manage events in views
- Use built-in views
- Use third-party libraries

As mentioned in the first chapter, we will rarely need to define views unless we really need a tight control over the DOM structure. In the next chapter, we will discuss how to use ember components that are a higher level construct of views.

Defining views

Views in an application manage templates through data binding and delegation of user-initiated events. Just like controllers, a view class is defined from the base `Ember.View` class as:

```
var View = Ember.View.extend({});
```

An instance of the preceding view can then be created by calling the view's `create` method:

```
var view = View.create();
```

We can still create additional views from the already defined class, as illustrated in the following code:

```
var UserView = View.extend({
  isLoggedIn: true,
  isAdmin: false
});
```

By now, we already know that Ember.js classes can accept any number of mixins as shown in the preceding example. However, instances created with mixins always use the `createWithMixins` method:

```
var mixinA = Ember.Mixin.create({
  isLoggedIn: true
});

var mixinB = Ember.Mixin.create({
  isAdmin: false
});

var userView = View.createWithMixins(mixinA, mixinB);
```

Accessing a view's controller

Views are usually backed by an instance of the corresponding controller. Once a view is inserted into the DOM, the corresponding controller can be accessed by the `controller` property, as shown in the following example:

```
view.get('controller').getSortedBooks();
```

Specifying a view's template

Every view renders a template into the DOM. Views can be assigned the template to use in a number of ways. For example, let's consider the following router:

```
App.Router.map(function(){
  this.route('new');
});
```

Ember.js will expect a defined `App.NewRoute` class for the new route as:

```
App.NewRoute = Ember.Route.extend({
  model: function(){
    return Em.Object.create();
  }
});
```

Any visit to this route will use the following Ember.js objects, if defined:

- The `App.NewController` object
- The `Ember.TEMPLATES.new` template
- The `App.NewView` object

The default behavior is where the name of the template dictates the view to use. If we wish to use a different template, say `Ember.TEMPLATES.form`, we will need to implement it in the `renderTemplates` hook of the route as:

```
App.NewRoute = Ember.Route.extend({
  model: function(){
    return Em.Object.create();
  },
  renderTemplate: function(controller, model){
    this.render('form');
  }
});
```

As expected, this route will use the `App.FormView` view class. The template used is usually included into the application as a script tag. For example, we can define the new template that will be used by an `App.NewView` as:

```
<script type='text/x-handlebars' id='new'>
<input name='name' >
```



```
<input name='gender' >
<button type='submit'>save</button>
</script>
```

Note that the previous template can still be accessed as `Ember.TEMPLATES.new`. Therefore, an alternative way to specify a view's template is to pragmatically update its value with the desired compiled template, as shown in the following code:

```
var template = [
  '<input name='name' >',
  '<input name='gender' >',
  '<button type='submit'>save</button>'
].join('');
```

```
Ember.TEMPLATES['new'] = Ember.Handlebars.compile(template);
```

In a production environment, it's advised to compile these templates server side, and then bundle them up for performance reasons. We can still specify the template using the `templateName` property on the view class, for example:

```
<script type='text/x-handlebars' id='form'>
  <input name='name' >
  <input name='gender' >
  <button type='submit'>save</button>
</script>
```

```
App.NewView = Ember.View.extend({
  templateName: 'form'
});
```

Specifying a view's element tag

A view's template is usually wrapped in a `div` element by default, as shown in the following example:

```
<div>{{name}}</div>
<div>{{gender}}</div>
```

This yields the following:

```
<div id='ember10' class='ember-view'>
  <div>Jon Doe</div>
  <div>Male</div>
</div>
```

This element type can be altered using the `tagName` property of the view class, as shown in the following code:

```
var View = Ember.View.extend({
  templateName: 'user',
  tagName: 'header'
});
```

The preceding snippet will yield something like the following:

```
<header div id='ember10' class='ember-view'>
  <div>Jon Doe</div>
  <div>Male</div>
</header>
```

Updating a view's element class attribute

In the previous section, we learned that views are usually wrapped in a configurable DOM element. The element's class attribute can also be specified statically using the view's `classNames` array property. For example, a Twitter Bootstrap button can be created as:

```
var view = Ember.View.extend({
  tagName: 'button',
  classNames: ['btn', 'btn-primary']
});
```

This will yield something like the following:

```
<button class='btn btn-primary'>Button</button>
```

The element's classes can also be altered dynamically using the view's `classNameBindings` array property, as shown in the following code:

```
var View = Ember.View.extend({
  tagName: 'button',
  classNames: ['btn'],
  classNameBindings: ['btnWarning'],
  btnWarning: true
});
```

This example yields the following:

```
<button class='btn btn-warning'>Button</button>
```

These class names are dasherized as per the Ember.js naming conventions. Hence, the `btnWarning` property is mapped to the `btn-warning` class name.

Sometimes, you might want to specify the class name to be used based on a given state. This is something we learned in [Chapter 4, Writing Application Templates](#), for example:

```
var View = Ember.View.extend({
  tagName: 'button',
  classNames: ['btn'],
  classNameBindings: ['warn:btnWarning'],
  warn: true
});
```

In the preceding example, the btn-warning class will be added to the class attribute of the element, based on the view's warn property.

Lastly, we can add different classes depending on a certain state. For example, imagine we want to display different states of our Bootstrap button. This is possible with the following signature:

```
classNameBindings: ['property:truthyClassName:falsyClassName'],
```

For example:

```
var View = Ember.View.extend({
  tagName: 'button',
  classNames: ['btn'],
  classNameBindings: ['controller.warn:btnWarning:btnPrimary'],
});
```

In the preceding example, when the view controller's warn property becomes true, the following will be yielded:

```
<button class='btn btn-warning'>checkout</button>
```

Otherwise, the other class will be used instead:

```
<button class='btn btn-primary'>checkout</button>
```

By now, you will have noticed that the binding behavior is similar to the one we learned in [Chapter 4, Writing Application Templates](#).

Updating other views' element attributes

In addition to class attributes, all other attributes of a view's element can be altered dynamically. For example, let's create a thumbnail view, as shown in the following code:

```
var thumb = Ember.View.create({
  tagName: 'img',
  attributeBindings: ['width', 'height', 'src'],
  width: 50,
  height: 50,
  src: 'http://www.google.com/doodles/new-years-day-2014'
});
```

This yields the following:

```
<img src='http://www.google.com/doodles/new-years-day-2014'
width='50' height='50'>
```

The attributes' presence can be altered using bound Boolean properties. For example, we can disable a save button of a form if the corresponding input has not been filled, as shown in the following example:

```
// view

App.FormButton = Em.View.extend({
  tagName: 'button',
  attributeBindings: ['disabled'],
  disabled: function(){
    return !this.get('controller.model.title');
  }.property('controller.model.title')
});

// route

App.NewRoute = Em.Route.extend({
  model: function(){
    return Em.Object.create({
    });
  }
});

{{! new template }}

<form {{action 'save' model on='submit'}}>
```

```
{{input value=model.title}}  
{{#view App.FormButton}}save{{/view}}  
</form>
```

When the model's title property is undefined, the view's disabled property will come true. Hence, the view's element will acquire the disabled attribute and vice versa. This example lets users submit the form only when it's valid. Note that the defined view can be reimplemented as:

```
App.FormButton = Em.View.extend({  
  tagName: 'button',  
  attributeBindings: ['modelIsValid::disabled'],  
  modelIsValid: function(){  
    return !this.get('model.title');  
  }.property('model.title')  
});
```

This example shows that any property, in this case, modelIsValid, can be used to provide the attribute to show or hide when the state changes, as long as it's specified using the following signature:

```
"propertyName:attributeWhenTrue:attributeWhenFalse"
```

Inserting views into DOM

We just learned that views have templates that they render into DOM. Applications that need to do this manually will need to utilize the view instance `appendTo` method, as shown in the following example:

```
view.appendTo('#header');
```

This method takes a jQuery query selector that we are already used to, as shown in the following examples:

```
view.appendTo('header');  
view.appendTo('#header');  
view.appendTo('.header');  
view.appendTo('body > header');
```

Note that only one matched element is used. Therefore, in the third example, the view will be inserted into the last header element found.

As a convenience, views have an `append` method that can be used to insert the views directly into the body section of DOM:

```
view.append(); // appends view to the body section
```

You might also want to remove a view from DOM using the `remove` method as:

```
view.remove();
```

Note that a view is automatically removed from DOM if destroyed, as shown in the following line of code:

```
view.destroy();
```

Inserting views into templates

Views are hierarchical, and hence, they can be inserted into the templates of other views constituting the template hierarchy we discussed in [Chapter 4](#), *Writing Application Templates*. For example, consider the following application template:

```
<script type='text/x-handlebars' id='application'>
  {{view App.HeaderView}}
  {{view App.FooterView}}
</script>
```

As shown, defined views are inserted into a desired template using the view expression. These view expressions can also be wrapped into block clauses, as shown in the following code. Additional views can then be inserted into these block expressions:

```
{{! application template}}
<script type='text/x-handlebars' id='application'>

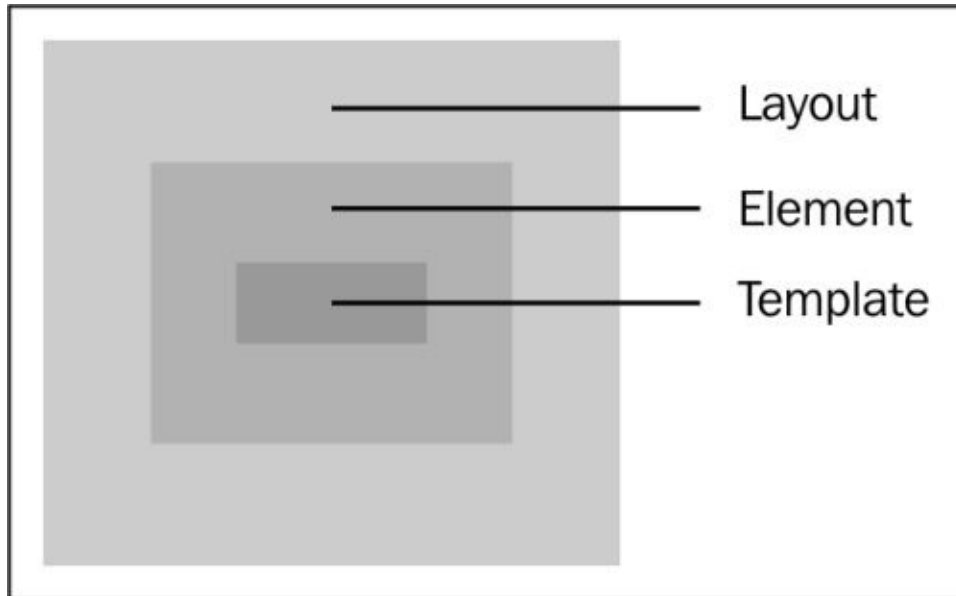
  {{view App.HeaderView}}

  {{#view App.ContentView}}
    {{view App.SideView}}
    {{view App.PaneView}}
  {{/view}}

  {{view App.FooterView}}
</script>
```


Specifying view layouts

We already learned that a view's template is wrapped in an element, which is usually specified by the `tagName` property. In addition, this template can be wrapped by another template, as illustrated in the following figure:



A template is marked as a layout by adding a `yield` expression, as shown in the following code:

```
<script type='text/x-handlebars' id='container'>
  <div id='container'>
    {{yield}}
  </div>
</script>
```

Just like the `outlet` expression, the `yield` expression serves as the portion that the template being wrapped will be inserted into. We then specify this layout in the view as:

```
var View = Ember.View.extend({
  tagName: 'section',
  layoutName: 'container',
  templateName: 'book'
});
```

Suppose our book template is:

```
<script type='text/x-handlebars' id='book'>
  <p>Author: Jon Doe</p>
</script>
```

This will yield the following:

```
<div id='container'>
  <section>
    <p>Author: Jon Doe</p>
  </section>
</div>
```

It is important to note that views with self-closing HTML elements cannot have layouts. These views include `<input>` and ``.

Registering event handlers in views

Views can register event handlers on events emitted from elements in their rendered templates, in addition to the use of the action template expressions. For example, let's reuse an example from [Chapter 4, Writing Application Templates](#):

```
<button {{action 'checkout'}}>checkout</button>
```

This example can easily be reimplemented as a view, as shown in the following code:

```
App.CheckoutButton = Ember.View.extend({
  tagName: 'button',
  click: function(event){
    this.get('controller').send('checkout');
  }
});

{{! template }}
{{#view App.CheckoutButton }}checkout{{/view}}
```

In this example, we created a custom button view that registers a click event handler.

Every view manages only the events invoked from their templates. However, child views usually bubble events to parent views up to the root element up until the events get handled.

Ember.js supports the following events:

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

		doubleClick	input	drop
		mouseMove		dragEnd
		focusIn		
		focusOut		
		mouseenter		
		mouseleave		

Now is a good opportunity to try and write views that use some of these events.

Emitting actions from views

We already learned that views have a reference to the context controller via the `controller` property. A view can use the controller's `send` method to delegate user-initiated events to the corresponding routes, as shown in the following example:

```
App.CheckoutButton = Ember.View.extend({
  tagName: 'button',
  click: function(event){
    this.get('controller').send('checkout');
  }
});
```

Using built-in views (components)

In [Chapter 4](#), *Writing Application Templates*, we promised to discuss the built-in views that Ember.js provides. Most of these are high-level views (components) from controls that guarantee painless design of forms.

Textfields

The textfield view is used to create a bound text input in a form. It's usually created from the `Ember.TextField` class. We can subscribe to the input's value changes by implementing the view's change event handler, as shown in the following code:

```
App.InputView = Ember.TextField.extend({
  change: function(event){
    console.log(this.get('value'));
  }
});
```

Just like any other view, we can insert this view into a template as:

```
{{view App.InputView name='name' valueBinding='controller.name'}}
```

In this example, we created a text input that updated the context controller's name property whenever its value changes. This is one of the many use cases of such views.

Textareas

A textarea is very similar to a textfield, both accept some additional attributes such as rows and cols, for example:

```
{{view Ember.TextArea name='content' valueBinding='content' rows=10  
cols=10}}
```


Select menus

Another common form control is the select menu. Ember.js provides an `Ember.Select` class that can be used to create this control. For example, let's create a select menu that prompts a user to choose their favorite fruit in this control:

```
// controller

App.ApplicationController = Ember.Controller.extend({
  selectedFruit: null,
  fruits: [{
    id: 1,
    name: 'mango'
  }, {
    id: 2,
    name: 'apple'
  }],
});

{{! template }}

{{view Ember.Select
  prompt='Select a fruit:'
  contentBinding='fruits'
  selectionBinding='selectedFruit'
  optionLabelPath='content.name'
  optionValuePath='content.id'}}
```

In the preceding example, the user is presented with two fruits to select from. They are first prompted with a **Select a fruit** prompt that was passed during definition. The view's content property is usually an array of the choices that should be displayed, and the selection property holds the selected choice. Often, these choices are usually objects rather than strings, as seen in the preceding example. Therefore, additional customization needs to be done using two properties:

- The `optionLabelPath` property: This specifies the choice's label
- The `optionValuePath` property: This specifies the value to be looked up for the selected choice

Therefore, the preceding example specified the fruit's name as the property to

display and the ID as the property to determine the selection.

Checkboxes

Checkboxes can also be implemented in the same way using the `Ember.Ceckkbox` view class. These controls enable the user to select various choices from a given set, for example:

```
{{view Ember.Checkbox name='is-complete'
valueBinding='isComplete'}}
{{view Ember.Checkbox name='is-done' valueBinding='isDone'}}
{{view Ember.Checkbox name='is-empty' valueBinding='isEmpty'}}
```

This will yield something like the following:

```
<input type='checkbox' name='is-complete' checked >
<input type='checkbox' name='is-done' >
<input type='checkbox' name='is-empty' >
```

The bound value of an instance of this view is usually a Boolean.

The container view

We already learned that a view can be inserted into other views using the view template helper, as shown in the following lines of code:

```
{{#view App.ContentView}}
  {{view App.SideView}}
  {{view App.PaneView}}
{{/view}}
```

In some cases, we might want the parent view, in this case, `App.ContentView`, to be able to manually manage child views. `Ember.ContainerView` is an enumerable view that an application can pragmatically add or remove child views from, as shown in the following example:

```
var sideView = Ember.View.create();
var paneView = Ember.View.create();
var contentView = Ember.ContainerView.create();
contentView.pushObjects([
  sideView, paneView
]);
```

These child views are usually contained in the `childViews` property. You can, therefore, implement the preceding example as:

```
var compile = Em.Handlebars.compile;
var contentView = Ember.ContainerView.create({
  childViews: ['sideView', 'paneView'],
  sideView: Ember.View.create({
    template: compile('Side')
  }),
  paneView = Ember.View.create({
    template: compile('Pane')
  })
});
```

This yields something like the following:

```
<div>
  <div>Side</div>
  <div>Pane</div>
</div>
```

It is important to note that since container views house other views, they cannot

have templates or layouts. Therefore, specified templates or layouts will be ignored.

Other HTML form controls can be abstracted to achieve simpler views. Therefore, as an exercise, create an `Ember.Radios` view class that displays a set of HTML radio buttons. Note that the implementation will be very similar to that of `Ember.Select`.

Integrating with third-party DOM manipulation libraries

Many jQuery libraries primarily manipulate DOM in order to achieve a desired effect. We all know that you need to initialize these libraries only when DOM is ready:

```
$(document).ready(function(event){  
  // initialize library  
  $('#menu').dropdown();  
});
```

jQuery is an Ember.js dependency, and it's thus very easy to integrate such libraries into applications. Imagine we had a menu view that we wanted to apply to this plugin. Views have the `willInsertElement` and `didInsertElement` hooks that we can use to implement such needs, as shown in the following code:

```
// view  
App.MenuView = Ember.View.extend({  
  didInsertElement: function(){  
    this._super();  
    Ember.run.schedule('afterRender', this, function() {  
      this.$().dropdown();  
    });  
  }  
});  
  
{{! template}}  
{{view App.MenuView}}
```

The `didInsertElement` hooks guarantee that the view has been inserted into DOM, and therefore, we might apply any plugin to it. Note that calling `this.$()` returns a jQuery element selector relative to the view. Also, note that we take care to call the `_super` method, as there can be parent implementations we cannot afford to lose. We also *schedule* this code to be run after the element is rendered into DOM.

At a later point in time, we might decide to remove the view from DOM. It will, therefore, be necessary that we remove any event that the plugin set up before removing the view. Ember.js provides the `willDestroy` hook that can be used to accomplish this:

```
App.MenuView = Ember.View.extend({
  willDestroy: function(){
    this._super();
    this.$().tearDownDropdown();
  }
});
```

Summary

In this chapter, we discussed how views are defined and created as well as how they can be customized. We also learned how events in these views can be managed. Lastly, we explored the different components Ember.js provides and how third-party libraries such as jQuery plugins can be integrated into an application through views.

The end of this chapter marked the completion of the core concepts of Ember.js. In the next chapters, we'll start building complete sample applications as we explore more features. You should, therefore, be well versed in the following Ember.js concepts and objects:

- Objects
- Routes
- Templates
- Controllers
- Views

Chapter 7. Components

Up until this chapter, we learned the basic Ember.js concepts that equipped us with the necessary tools that enabled us to create full-fledged applications. From this chapter onwards, we will be guided through creating all sorts of sophisticated applications as we explore more advanced Ember.js features. This chapter will introduce us to Ember.js components, which enable us to create custom reusable elements, and will cover the following topics in this regard:

- Understanding components
- Defining components
- Customizing components
- Using components as template layouts
- Defining actions inside a component
- Interfacing a component with the rest of the application

Understanding components

A web component is a reusable custom HTML tag. The World Wide Web Consortium is already working on custom web element (web components) specifications (<http://www.w3.org/TR/components-intro/>) that will allow developers to create these custom HTML elements with custom behaviors as opposed to always relying on the provided standard HTML elements. This specification is still being worked on, but there are a number of JavaScript open source projects (shims) that can get you started before this specification is complete:

- **Polymer**, available at <http://www.polymer-project.org/>
- **Facebook React**, available at <http://facebook.github.io/react/docs/component-api.html>
- **Ember.js components**

Ember.js provides mechanisms that will allow developers to create and complete these components in the near future of web technology. Once web components are standardized, Ember.js will continue enabling the easy creation of these custom elements. It's therefore an added advantage to start utilizing the Ember.js component APIs.

Defining a component

Components are a higher-level construct of Ember.js views, and therefore, to define one, we'll need to define either or both of the following two Ember.js objects:

- The component's class
- The component's template

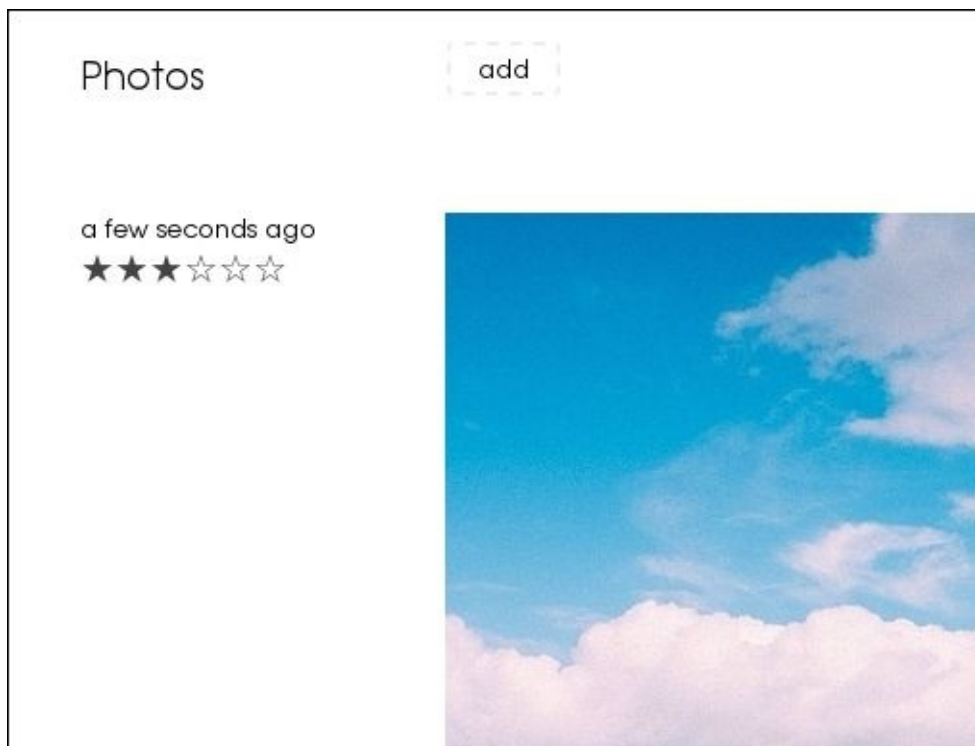
The class is usually extended from the `Ember.Component` class in the following signature:

```
MyAppNamespace.ComponentNameComponent = Ember.Component.extend();
```

The component template is then defined and named using Ember.js conventions. For example, the template for the preceding component will be named as:

`components/component-name`

The bundled sample of this chapter includes a simple application that utilizes several components. This application allows users to upload and rate photos as shown in the following screenshot:



The application defines the following components:

- Post input component
- Post date component
- Post rating component
- User post component
- Post photo component

We already noticed that some of these components were defined by either a class or a template. For example, the user-post component did not define a class. Also, the template names were namespaced using a hyphen, while the class names were camelized. Therefore, it would have been incorrect to register the user-post component template as userpost. This rule corresponds to one of the following component attributes described by W3C:

- Component custom elements must be namespaced by a hyphen.
- Components are sandboxed but can communicate through events.
Therefore, the component and the host DOM JavaScript cannot manipulate each other.

Hence, the post-input component class and template, for example, were defined as:

```
// class
App.PostInputComponent = Ember.Component.extend({
});

{{! template}}
<script type="text/x-handlebars" id="components/post-input">
add
</script>
```

Once defined, a component can be included into any application template using the Handlebars expressions. For example, the first component is a button that is used to prompt the user to select an image from the disk:

```
<script type="text/x-handlebars" id="components/post-input">
add
</script>
```

Now, our router defines the photos route that handled requests at the home path as:

```
App.Router.map(function() {  
  this.resource('photos', {path: '/'})  
});
```

Therefore, all we need to do is include the component in the corresponding photos template as:

```
<script type="text/x-handlebars" id="photos">  
  
  ...  
  {{post-input posts=model}}  
  
  ...  
</script>
```

This will result in the components template being swapped, resulting in:

```
<script type="text/x-handlebars" id="photos">  
  ...  
  <button>add</button>  
  ...  
</script>
```

Do not worry about how the resulting element is a button. The important thing to note is that we just defined and used a custom HTML element without worrying about its underlying implementation.

Differentiating components from views

Before we proceed, you might be wondering why components and views are different since both wrap templates. Well, components are indeed a subclass of views, but their controller context is isolated from the rest of the application. While application controllers can be assigned to any view, the defined component classes cannot be assigned to other components or views. Components define an interface the desired context must implement, and they are therefore more reusable and modular, as we will see in the next sections.

Passing properties to components

While we just mentioned that components are isolated from the rest of the application, there is room for them to communicate with the host application in several ways. First, they're able to bind to properties in the host template context. For example, we just mentioned that the preceding button component is used to prompt the user to upload images. The component requires that an enumerable property, which will act as the photo store, be bound to its `posts` property:

```
{{post-input posts=model}}
```

This way, the component will be able to store the provided photos, as we will discuss in a later section. These selected photos will then be displayed to the user in the same template as:

```
<script type="text/x-handlebars" id="photos">
...
<div class='posts'>
  <ul>
    {{#each model}}
      <li>{{user-post post=this}}</li>
    {{/each}}
  </ul>
</div>
...
</script>
```

Here, we used yet another component, `user-post`, which rendered a given photo into the included page portion. Again, we did not need to worry about the underlying implementation of the component. We only needed to satisfy its interface's requirement of binding a photo to its `post` property.

To understand how these components used the bound properties, let's consider the `post-date` component that was used by the just discussed `user-post` component to display a humanized format of the post's date. This component contains a single expression that displays the formatted date as:

```
<script type="text/x-handlebars" id="components/post-date">
  {{formattedDate}}
</script>
```

The expression is a computed property that uses the Moment.js library (<http://momentjs.com>) to format the date and is defined in the corresponding class as:

```
App.PostDateComponent = Ember.Component.extend({
  formattedDate: function(){
    return moment(this.get('date')).fromNow();
  }.property('date')
});
```

The dependent date property is then bound in the user-post component as:

```
{{post-date date=post.date}}
```


Customizing a component's element tag

Since the W3C component specification is still being worked on, Ember.js components utilize the existing standard HTML elements. In [Chapter 3, Routing and State Management](#), we learned that a view's template is wrapped around an element, which by default is `div`. This element can then be customized using the view's `tagName` property. A component's template is also wrapped in the same way in a customizable element. For example, we promised to discuss how the `post-input` component mentioned previously was rendered into DOM.

All we need to do is define the property in the corresponding class as:

```
App.PostInputComponent = Ember.Component.extend({  
  ...  
  tagName: 'button',  
  ...  
});
```

Customizing a component's element class

Since components are views, their element's class can be specified statically or dynamically using the `classNames` and `classNameBindings` array properties on the component's class. For example, the `post-input` component defines a static class as:

```
App.PostInputComponent = Ember.Component.extend({
  classNames: ['post-input'],
});
```

This results in the component being rendered as:

```
<button class='post-input'>add</button>
```

In the sample application, we mentioned that users are able to rate uploaded photos. The `user-post` component uses the `post-rating` component that serves as the rating widget:

```
{{post-rating content=post}}
```

Each of the stars in the latter part is also a component (`post-rating-item`) and is listed horizontally to compose the widget as:

```
<script type="text/x-handlebars" id="components/post-rating">
  <ul>
    {{#each rating in ratings}}
      <li>{{post-rating-item controller=post content=rating}}</li>
    {{/each}}
  </ul>
</script>
```

As expected, the colored stars represent the range of the rating, and so, we use the `active` class in this case to style them:

```
App.PostRatingItemComponent = Ember.Component.extend({
  classNameBindings: ['active'],
  active: function() {
    ...
  }.property('parentView.selected'),
});
```

This is an example of a dynamic class where the component will only acquire

the class if the defined computed active class evaluates to True. We will discuss how this rating works in a later section, but one last thing to note is that in the case of dynamic classes, we can specify the class name to use. For example, we can implement the preceding case as:

```
App.PostRatingItemComponent = Ember.Component.extend({
  classNameBindings: ['isActive:active'],
  isActive: function() {
    ...
  }.property('parentView.selected'),
});
```

We can also implement it as:

```
App.PostRatingItemComponent = Ember.Component.extend({
  classNameBindings: ['isActive:active:not-active'],
  isActive: function() {
    ...
  }.property('parentView.selected'),
});
```

In the preceding snippet, the not-active class will be acquired by nonactive stars.

Customizing a component's element attributes

A component's element attribute values can also be bound to properties using the `attributeBindings` property. For example, consider our `post-photo` component that displays the images as:

```
App.PostPhotoComponent = Ember.Component.extend({
  tagName: 'img',
  classNames: ['avatar'],
  attributeBindings: ['src'],
  src: Ember.computed.oneWay('photo')
});
```

First, we use the `tagName` property to specify that its element is an image tag. We also specify that this element will have an `src` attribute that will be aliased to the bound `photo` property. The `user-post` component then uses this component to display the images as:

```
{{post-photo photo=post.photo}}
```

Be sure to compare how the following element concepts are customized across views, components, and even templates:

- Tag names
- Class attributes
- Attributes

Managing events in components

Just like views, components can catch user-generated events such as those from the keyboard, mouse, and touch devices.

There are two ways in which the handlers to these events can be defined, the first of which is to attach the `.on` function to the event-subscriber method. For example, the `post-input` component uses this function to define two handlers. This button component implements a file-picker dialog that can be opened from an invisible file input, as described at <https://github.com/component/file-picker>. As soon as the component gets rendered, the event is fired, which results in the hidden form element containing a single input file to be appended into DOM as:

```
createHiddenForm: function(){
  var tpl = [
    '<form class="post-input-form">',
    '<input type="file" style="top: -1000px; position: absolute"
    aria-hidden="true">',
    '</form>'
  ].join('');

  Em.$('body').append(Em.$(tpl));
}.on('didInsertElement'),
```

This form will be used later to upload the images. Next, we define the handler that will initiate the file dialog to open. Note that we use the `.on` method to subscribe to the button's click event:

```
upload: function(){
  ...
}.on('click')
```

Inside this handler, we set up a listener that will get invoked when the user selects an image file, as shown:

```
var input = Em.$('.post-input-form input');
input.one('change', upload);
```

Here is the handler which does the upload:

```
function upload(event){
```

```
var file = input[0].files[0];
var reader = new FileReader;
reader.onload = post.bind(this, reader);
reader.readAsDataURL(file);

};
```

In the preceding snippet, we create a `FileReader` instance and pass the uploaded image to it. We then read the `dataUrl` representation of the image, which then gets sent to the final bound handler:

```
function post(reader){

    var data = {
        photo: reader.result,
        date: new Date
    }
    self.get('posts').pushObject(data);
}
```

This last handler adds the image to the photo controller as a new post. Note that we don't check the mime type of the uploads, so the user might upload other media types, such as videos. This check is left out as an implementation exercise to the reader.

Secondly, we subscribe to these events to implement a method whose name corresponds to the target event, as shown by the rating widget component previously discussed. This component keeps count of the following two properties:

- `selected`: This is the selected/hovered star position
- `_selected`: This is the cached position of the last clicked star

We mentioned that the widget is composed of the `post-rating-item` components that represent each of the stars. When the user hovers over any of them, we updated the `selected` property of the parent component as:

```
mouseenter: function(e) {
    var selected = this.get('content');
    this.set('parentView.selected', selected);
}
```

As shown, we defined a method that corresponded to the `mouseenter` event. This handler sets the active property of all the rating item components to the left to `True`, since the trick here is to apply the style, as expected, to all stars to the left-hand side of the currently selected one:

```
classNameBindings: ['active'],

active: (function() {
  var content = this.get('content');
  var selected = this.get('parentView.selected');
  return ~~content <= ~~selected;
}).property('parentView.selected'),
```

On the other hand, those to the right-hand side lose the active class because their active property gets recalculated to `False`.

If the user doesn't click on any of the stars, they expect the earlier rating to be restored. Therefore, leaving the currently focused component uses the cached `_selected` property to reset the selected property, as shown in the following code:

```
mouseleave: function(e) {
  var selected = this.get('parentView._selected');
  this.set('parentView.selected', selected);
}
```

Again, we only need to implement the `mouseleave` event hook. Lastly, clicking on any of the components gives us the actual rating:

```
click: function(e){
  var content = this.get('content');
  this.set('parentView.selected', content);
  this.set('parentView._selected', content);
}
```

Note that we cache the `_selected` property of the parent component since this will be used in the preceding checks. The active class updates the components state appropriately as:

```
.rating:before{
  content: "☆"
}
```

```
.rating.active:before{  
  content:"★"  
}
```


Defining component actions

We mentioned that components define classes that act as their controllers that are isolated from the rest of the application. For example, an application controller cannot define a component class as a dependency in its needs property.

However, since they are considered as controllers, they can define handlers to action expressions defined in their corresponding templates in an actions object property. For example, let's define a message box component that can be used in any application that needs to implement the chat functionality:

```
{{! template }}}  
<form {{action 'save' on='submit'}}>  
  
  {{input value=message}}  
</form>  
  
// component classApp.MessageBoxComponent =  
Ember.Component.extend({  
  message: '',  
  classNames: ['message-box'],  
  actions: {  
    save: function(){  
      var message = this.get('message').trim();  
      if (message === '') return;  
      var content = this.get('content');  
      content.pushObject(message);  
      this.set('message', '');  
    }  
  }  
});
```

To use this component, one simply needs to provide the Messages container in which new messages will be stored. Here's a possible example:

```
<script type="text/x-handlebars" id="messages">  
  <h1>Messages</h1>  
  <ul>  
    {{#each model}}  
      <li>{{this}}</li>  
    {{/each}}  
  </ul>  
  <div>{{message-box content=model}}</div>  
</script>
```

The component form defines an action that binds the save action handler of the component class to the form's `submit` event. When the user submits the form by hitting the *Enter* key, the handler sanitizes the message before pushing it to the provided container. You'll realize that these actions are similar to the ones we learned earlier in [Chapter 4](#), *Writing Application Templates*. However, there's no event bubbling in components. A failure to locate the handler in the class will lead to an appropriate error being thrown.

Interfacing a component with the rest of the application

Components, as mentioned earlier, are not completely sandboxed, but they can interact with the rest of the application in the following ways:

- Bind to properties
- Send actions

We have already seen how components are able to bind to other application properties by passing the properties in the template expressions:

```
{{post-input posts=model}}
```

Components also have the ability to send their actions to controllers in an application. To demonstrate this, let's create a simple checkout button for an e-commerce site:

```
{{! template}}  
<script type="text/x-handlebars" id="components/checkout-button">  
add to cart  
</script>
```

```
// add to cart component  
App.CheckoutButtonComponent = Ember.Component.extend({  
  tagName: 'button',  
  click: function(){  
    this.sendAction();  
  }  
});
```

```
{{! cart template}}  
<ul>  
  {{#each products}}  
    <li>  
      {{name}}  
      {{price}}  
      {{checkout-button}}  
    </li>  
  </ul>
```

```
// cart controller  
App.CartController = Ember.ArrayController.extend({  
  actions: {
```

```

        click: function(product){
            this.pushObjects(product);
        }
    }
});

```

In the preceding example, we intend to add a product to cart whenever its corresponding checkout button is clicked via an event handler. We utilize the component's `sendAction` method to bubble this action to the parent controller. However, there are two things we need to fix in order to realize this. First, we need to rename our event handler in the controller to something descriptive. Moreover, the same `click` event handler can catch events from other elements as:

```

// cart controller
App.CartController = Ember.ArrayController.extend({
    actions: {
        addToCart: function(product){
            this.pushObjects(product);
        }
    }
});

```

Next, we need to send the selected product to the `addToCart` handler with a little modification of the template:

```

{{! cart template}}
<ul>
  {{#each products}}
    <li>
      {{name}}
      {{price}}
      {{checkout-button product=this action='addToCart'}}
    </li>
  </ul>

```

This just lets the component be able to access the product. Finally, we send the product to the controller event handler as:

```

// add to cart component
App.CheckoutButtonComponent = Ember.Component.extend({
    tagName: 'button',
    template: Ember.Handlebars.compile('add to cart'),
    click: function(){

```

```
        this.sendAction('action', this.get('product'));  
    }  
});
```

Note that the first argument to `sendAction` is always `action` followed by the object(s) we wish to send.

Components as layouts

A component's template can act as a layout for other application templates. These layouts are not specified in the view layer; they use block expressions instead. Additional content can then be inserted inside these templates without losing scope. For example, imagine we wish to create a component that will use the `content-editable` element. This kind of component will need to wrap a section of some HTML content as:

```
{{#content-editable}}  
<p>Tweet content</p>  
{{/content-editable}}
```

As shown, the component uses custom Handlebars tags that match its namespaced template name. The content inside and outside the component will still enjoy the same scope. Can you guess how this component will be implemented? One implementation will be to turn the wrapped content into `content-editable` when double-clicked or focused, and back to `div` when the mouse leaves the element as:

```
App.ContentEditableComponent = Ember.Component.extend({  
  attributeBindings: ['isEditing:contenteditable'],  
  doubleClick: function(){  
    this.set('isEditing', true);  
  }  
  focusOut: function(){  
    this.set('isEditing', false);  
  }  
});
```

The two event-defined handlers toggle the `isEditing` property, which then results in the `content-editable` attribute to be added or removed from the element accordingly.

To make things a little interesting, imagine we want to upgrade our `content-editable` component into a WYSIWYG editor. We will need to define a template that is used to host the different controls to manipulate the content, as shown in the following example:

```
<script type="text/x-handlebars" id="components/content-editable">  
  <div class="controls-toolbar">
```

```
<button>bold</button>
<button>italic</button>
<button>underline</button>
<button>strike-through</button>
</div>
<div class="content">
  {{yield}}
</div>
</script>
```

First, we define a toolbar that will house standard editor controls. We can bind action handlers, as discussed in the previous sections, to perform the manipulations; this will be a worthy attempt by the reader. In the content section, we use the `yield` expression we discussed in the previous chapter to tell the component to render the wrapped content in this portion of the template. With this powerful feature, both the component and the wrapped content can define expressions that bind to isolated contexts.

Summary

You'll find components useful when you want to modularize your application. There are various open source tools that will enable you to ship these components to your application. In the last chapter of this book, we will learn how to ship these components and mixins into your application using the component's (<http://github.com/component>) asset manager and build. Therefore, it's best practice to abstract modular objects in your application into either mixins or components. The following are some of the things we learned about components:

- Defining components
- Customizing component elements and attributes
- Managing actions inside components
- Interfacing components with the rest of the application

In the next chapter, we will learn how to sync data between Ember.js applications and REST backends. We will particularly learn how to use Ember.js data to simplify this need.

Chapter 8. Data Persistence through REST

Up to this point, we've been dealing with frontend aspects of Ember.js-powered applications. Your typical application will, however, need to connect to backend services such as databases. Ember.js makes this simple by integrating solutions for such needs. This chapter assumes no knowledge of server-side technologies, but it will attempt to explain any samples that contain server-side code as clearly as possible. Ensure that you also attempt the given exercises in order to understand the following:

- Making Ajax requests
- Understanding Ember-data
- Creating data stores
- Defining models
- Declaring model relations
- Creating records
- Updating records
- Deleting records
- Persisting data
- Finding records
- Defining a store's adapter
- Creating REST APIs
- Customizing a store's serializer

Making Ajax requests

Most web applications communicate with backend services through either of the following technologies:

- Web sockets
- Ajax

This chapter will mainly deal with Ajax, which enables client applications to send asynchronous requests to remote services through the use of XMLHttpRequests. Web sockets will be handled in a later chapter, but we'll find that many concepts used will be related. Here's an example of a POST request to a music catalog endpoint:

```
var data = JSON.stringify({
  album: 'Folie A Deux',
  artiste: 'Fall Out Boy'
});

function onreadystatechange(event){
  if (event.target.readyState !== 4) return;
  console.log('POST /albums %s', event.target.status);
}

var xhr = new XMLHttpRequest();
xhr.onreadystatechange = onreadystatechange;
xhr.open('POST', '/albums');
xhr.setRequestHeader('Content-type', 'application/json');
xhr.send(data);
```

This is obviously boilerplate code, and jQuery makes this as simple as:

```
$
  .post('/albums', data)
  .then(function(albut){
    console.log('POST /albums 200');
  });
```

There are numerous ways we could integrate this into an Ember.js application. For example, if this was to be initiated after a form submission, we could implement it in a save action, as follows:

```
<script type='text/x-handlebars'>{{outlet}}</script>
```

```

<script type='text/x-handlebars' id='index'>

  {{#with model}}
  <form {{action 'saveAlbum' this on='submit'}}>
    {{input value=artiste}}
    {{input value=album}}
    {{input value='save' type='submit'}}
  </form>
  {{/with}}

</script>

<script>

  App = Ember.Application.create();

  App.Router.map(function(){});

  App.IndexRoute = Ember.Route.extend({
    model: function(){
      return {};
    },
    actions: {
      saveAlbum: function(album){
        var data = {
          album: album.album,
          artiste: album.artiste
        };
        $.post('/albums', data)
        .then(function(album){
          console.log('POST /albums 200');
        }, function(response){
          alert('failed!');
        });
      }
    }
  });
</script>

```

When a user submits the provided form, the index controller's saveAlbum action would be called to post the album to the server using jQuery. Ideally, we could create an album class to separate concerns, as follows:

```

App.Album = Ember.Object.extend({
  toJSON: function(){

```

```

    return {
      album: this.get('album'),
      artiste: this.get('artiste')
    };
  },
  save: function(){ // function to persist album to the server
    $.post('/albums', this.toJSON())
    .then(function(event){
      console.log('POST /albums 200');
    }, function(event){
      alert('failed!');
    });
  }
});

```

With this class, we could then do a final refactor, as follows:

```

App.IndexRoute = Ember.Route.extend({
  model: function(){
    return App.Album.create();
  },
  actions: {
    saveAlbum: function(album){
      album.save();
    }
  }
});

```

In addition, we may wish to load the saved albums from the server to present to the user. A simple way would be to implement a `find` class method to load these albums, as follows:

```

Album.reopenClass({
  find: function(){
    return $.getJSON('/albums');
  }
});

```

This example adds a static class method to the `Album` class, which would then be used to query the backend for albums, as follows:

```

App.AlbumsRoute = Ember.Route.extend({
  model: function(){
    return App.Album.find();
  }
});

```

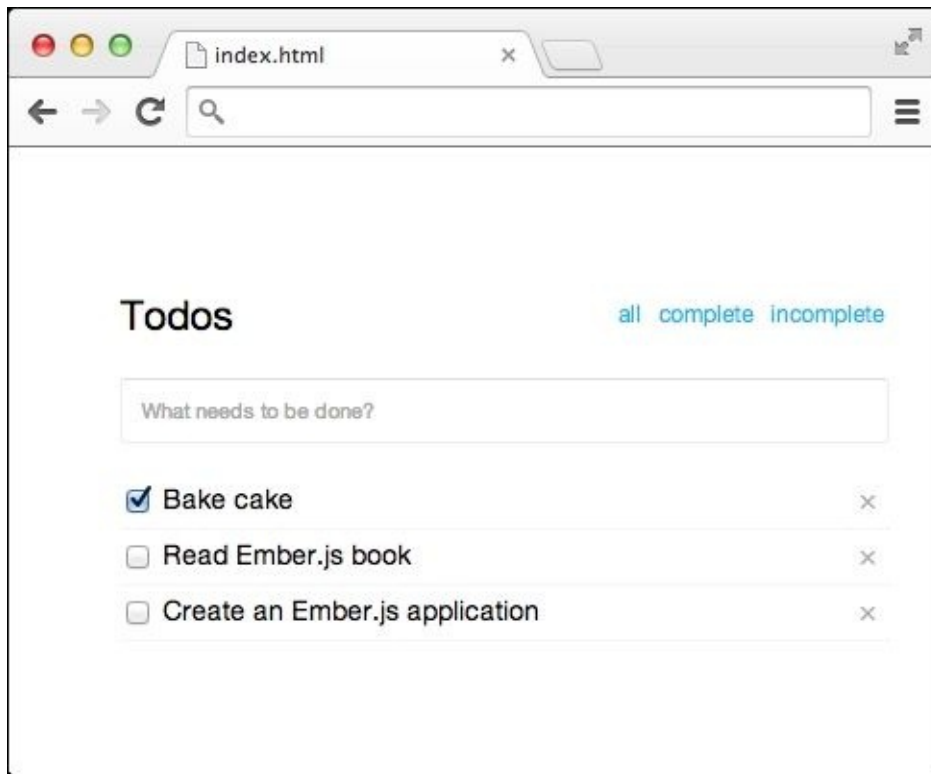
```
});
```

As this is a common practice, the Ember.js community also maintains another project called **Ember-data** (<http://github.com/emberjs/data>) and that aims to abstract such needs. This chapter is, therefore, going to walk us through using Ember-data in RESTful applications. These applications use **REST** (**Representational State Transfer**), which, as we know, lets us consume APIs that use some of the following HTTP verbs:

- GET
- POST
- PUT
- DELETE

Understanding Ember-data

Ember-data is another ambitious, opinionated, open source project used to develop applications that need to communicate with backend database services. A suitable version can be downloaded from <http://builds.emberjs.com/>. In our case, we will be using Version 1.0.0-beta.9 build that is already included in the chapter samples. These samples define a simple **Todos** application implementation of <http://github.com/component/todo>:



We will be using the `fixtures-adapter` sample first, which you can load via the `index.html` file. This application enables users to do the following:

- Load saved todos
- Create and save new todos
- Filter loaded todos by state (complete versus incomplete todos)

Ember-data namespace

The Ember-data library utilizes its own global namespace named `DS`, from which we will be referencing commonly-used classes such as `DS.Store` and `DS.Model`.

Creating a data store

Applications that use Ember-data usually use a single storage repository that stores all records that are available to the application. This store is defined from the `DS.Store` class, as follows:

```
App.ApplicationStore = DS.Store.extend({
});
```

The preceding code is automatically executed by Ember.js, and so we do not need to do anything. Just like the router, this class is usually autoinstantiated and made accessible to all routes and controllers as a `store` property. Here's an example that demonstrates a route that accesses the application's store:

```
App.BooksRoute = Ember.Route.extend({
  model: function(){
    return this.store.find('book');
  }
});
```

Do not worry about what this does. The important thing to learn from the preceding snippet is how the store instance is accessed.

Defining models

In the introductory chapters, we learned how to organize application objects into reusable classes called `models`. Ember-data provides support for defining such models that extend `DS.Model`, and from which records can then be created. For example, let's review the `Todo` model that was defined in the samples:

```
App.Todo = DS.Model.extend({
  title: DS.attr('string'),
  complete: DS.attr('boolean', {
    defaultValue: false
  })
});
```

As illustrated, the model was defined by extending the `DS.Model` class. We then defined two attributes using the `DS.attr` class methods that take two arguments:

- The name of the attribute
- An optional options object

The attribute's type is usually one of the following:

- String
- Number
- Date
- Boolean

However, we learn later that it's possible to define other custom types. The options object commonly contains a `defaultValue` property, which could be a value or a function that evaluates to the value to be used as the default.

Declaring relations

Records in our application may be related; thus, Ember-data supports defining some of the following common relations:

- One-one
- One-many
- Many-many

One to one

In this type of relation, only one model can belong to the other. For example, we can define two objects, a person and a passport, where the person only owns one passport:

```
App.Person = DS.Model.extend({
```

Finding records

Ember-data provides various ways to query loaded records as well to pull new ones from backend services. To find all records of a particular model, we can simply utilize the store's `find` method, as follows:

```
// GET /todos

store.find('todo');
```

This method loads all todos from the server via a promise that we then consume, shown as follows:

```
store
  .find('todo')
  .then(function(todos){
    todos.map(function(todo){
      todo.set('complete', false);
      return todo;
    });
  });
```

If we, however, want to only query loaded records, we can use the store's `all` method, as follows:

```
store.all('todo');
```

Similarly, we may want to query a record by a given id, as follows:

```
// GET /todos/1

store.find('todo', id);
```

Querying records by search terms is also as easy as:

```
// GET /todos?complete=true

store.find('todo', {
  complete: true
});
```

Defining a store's adapter

Every store needs an adapter that sits at the network layer, where it makes the actual API request calls. This is what differentiates the two variants of our Todos application, where each of its stores defines an adapter that communicates with different remote data stores. For example, the first sample defines its adapter as follows:

```
App.ApplicationAdapter = DS.FixtureAdapter;
```

All adapters need to implement the following methods:

- `find`
- `findAll`
- `findQuery`
- `createRecord`
- `updateRecord`
- `deleteRecord`

These adapters enable applications to stay in sync with various data stores such as:

- Local caches
- A browser's local storage or indexeddb
- Remote databases through REST
- Remote databases through RPC
- Remote databases through WebSockets

These adapters are, therefore, swappable in case applications need to use different data providers. Ember-data comes with two built-in adapters: the `fixtures-adapter` and the `rest-adapter`.

The fixtures adapter uses an in-browser cache to store the application's records. This adapter is especially useful when the backend service of the project is either inaccessible for testing or is still being developed. When using this adapter, it may be necessary to add initial data called fixtures to mock out the existing records. These records can be loaded into the application's store by adding them in the `FIXTURES` property of the affected model, shown as follows:

```
App.TODO.FIXTURES = [  
  { id: 1, title: 'Bake cake', complete: true },  
];
```

Creating REST APIs

Once satisfied with the workings of the models, we may then swap out the fixtures adapter with the rest -adapter, that as you guessed, communicates with remote data stores through REST. The second sample includes a simple Node.js server (Server.js) that uses Express.js (<http://expressjs.com>) to demonstrate the use of this adapter. To test the application, you will need to install Node.js by following these steps:

1. Download your platform's Node.js binary at <http://nodejs.org/download/>.
2. Un-archive the downloaded package.
3. Add the location of the bin directory inside the un-archived directory into your environment PATH setting.
4. Test out the installation by running node in a terminal.

To start the application, navigate to the rest -adapter sample directory and then simply run the following two commands in your shell emulator:

```
npm install
node server
```

Then, visit <http://localhost:5000> in your browser. We notice that the two applications are different in that the latter persists data to the running backend. If we add new todos and visit a new tab, we will realize that the new changes are reflected. The application, however, does not persist the changes in a real database as this is out of the scope of this book. Therefore, as an exercise, try and reimplement this sample in your favorite server-side stack.

The rest-adapter makes a few assumptions that our todos server API must adhere to, shown in the following table:

Action	Request HTTP verb	Request URL	Request JSON payload	Response JSON data
Create	POST	/todos	{todo: data}	{todo: data} or id
Find all	GET	/todos	None	{todo: data}

Find query	GET	/todos?complete=true	None	{todo: data}
Find one	GET	/todos/1	None	{todo: data}
Update	UPDATE	/todos/1	{todo: data}	{todo: data} or None
Delete	DELETE	/todos/1	None	None

This implementation can be found in the `api.js` module. Therefore, it is wise to use such a format when creating new APIs that are primarily consumed by Ember-data applications. This convention is also documented at <http://jsonapi.org/>, which may be a great resource for you.

Related objects can also be loaded in a similar way. For example, in our *tweet-retweet* case, we could load retweets of a particular tweet, as follows:

```
{ tweet: {
  id: 1,
  title: 'New book out',
  retweets: [{
    id: 1,
    title: 'RT New book out',
    user: 'Jon',
  }, {
    id: 2,
    title: 'RT New book out',
    user: 'Doe'
  }]
}}
```

Notice that Ember-data expects that the property in the response data that contains the related objects should be named as the pluralized form of the related model:

```
retweets: DS.hasMany('retweet')
```

```
retweet => retweets
person => people
```

Alternatively, the API can only send the IDs of the related objects, as follows:

```
{ tweet: {
  id: 1,
```



```
    title: 'New book out',
    retweets: [1, 2, 3]
  }}
```

Ember-data will then *side-load* the corresponding objects into the data store.

Sometimes, a model may have more than one relation of the same model. For example, a typical Facebook user has followers and followings, whose model can then be defined as follows:

```
App.User = DS.Model.extend({
  followers: DS.hasMany('user'),
  followings: DS.hasMany('user')
});
```

Ember-data will then expect the response data to contain a list of related objects named users. However, since there is more than one attribute that depends on the user model, we can easily resolve this by using the *inverse* option, as follows:

```
App.User = DS.Model.extend({
  followers: DS.hasMany('user', { inverse: 'followers' }),
  followings: DS.hasMany('user', { inverse: 'followings' })
});
```

With this, we can then return a response as follows:

```
{ user: {
  id: 1,
  followers: [],
  following: []
}}
```

If the application consumes APIs from different endpoints, we will need to define different adapters for each of the models, as follows:

```
App.BookAdapter = DS.RESTAdapter.extend({
  namespace: 'v3/',
  host: 'http://books.example.com'
});
```

```
App.PenAdapter = DS.RESTAdapter.extend({
  namespace: 'v3/',
  host: 'http://pens.example.com'
```

```
});
```

As shown in the preceding code, adapters can be customized in many different ways in order to meet the needs of your APIs and domain logic. This ensures that the existing APIs can still be consumed easily rather than having to build separate API endpoints reserved for Ember-data applications.

Customizing a store's serializer

In addition to a store's adapter, all stores have a serializer that serializes and deserializes data that comes in and out of the application. For example, if the data models in our backend use primary keys other than `id`, we can easily do this:

```
DS.RESTSerializer.reopen({  
  primaryKey: 'key'  
});
```

Note that this could also be specified per model, as follows:

```
App.PhoneSerializer = DS.RESTSerializer.extend({  
  primaryKey: 'phone_id'  
});
```

Creating custom transformations

Transformations are the different types of model attributes. Application authors are not limited to the built-in ones, so they may define their own transformations easily. For example, one of our backend services may represent Booleans as zeros and ones:

1 - true
0 - false

We could create a transformation that resolves these values where necessary:

```
App.BinaryBoolean = DS.Transform.extend({
  serialize: function(boolean){
    return (!boolean)
      ? 0
      : 1;
  },
  deserialize: function(binary){
    return (!!!binary)
      ? false
      : true;
  }
});
```

We just created a new transformation by extending `DS.Transform` and then defining the following two methods that act on the attribute's value:

- `serialize`: This converts the attribute value to a form acceptable by the server
- `deserialize`: This converts the value loaded from the server into what the application will use

We can then easily use this new type as follows:

```
App.TODO = DS.TODO.extend({
  complete: DS.attr('binaryBoolean')
});
var todo = store.createRecord('todo', {
  complete: true
});
todo.save(); // POST /todos {'todo': {complete: 1}}
```

Summary

In this chapter, we've discussed how to use Ember-data to create applications that need to communicate with backend storage services through REST. We have learned how to create records from defined models as well as updating and deleting them. We have also learned the different customizations we would need to make in order to consume existing APIs as much as possible. We should, therefore, be comfortable enough to start writing any client-side applications backed by REST APIs. As we proceed to the other exciting chapters, we should start thinking of how web sockets, JSONP, and RPC can be integrated with Ember-data seamlessly.

Chapter 9. Logging, Debugging, and Error Management

Until now, we have learned the basics of architecting and building Ember.js applications. In this chapter, we will learn how to debug these applications in order to not only reduce development time, but also to make development more fun. We will, therefore, cover the following topics:

- Logging
- Tracing events
- Debugging errors
- Using the Ember.js inspector

Logging and debugging

Ember.js can be downloaded in two formats that are meant to be used in development and production environments accordingly. The development (magnified) build is recommended to be used during the application development period for easier debugging. There are various ways to log and inspect objects created inside an application. We will discuss how to log and debug each of these objects in detail.

Objects

Besides the logging functions already provided by the browser's console object, Ember.js provides the following `Ember.Logger` logging utilities that are specifically meant to log Ember.js objects:

- `assert`
- `debug`
- `error`
- `info`
- `log`
- `warn`

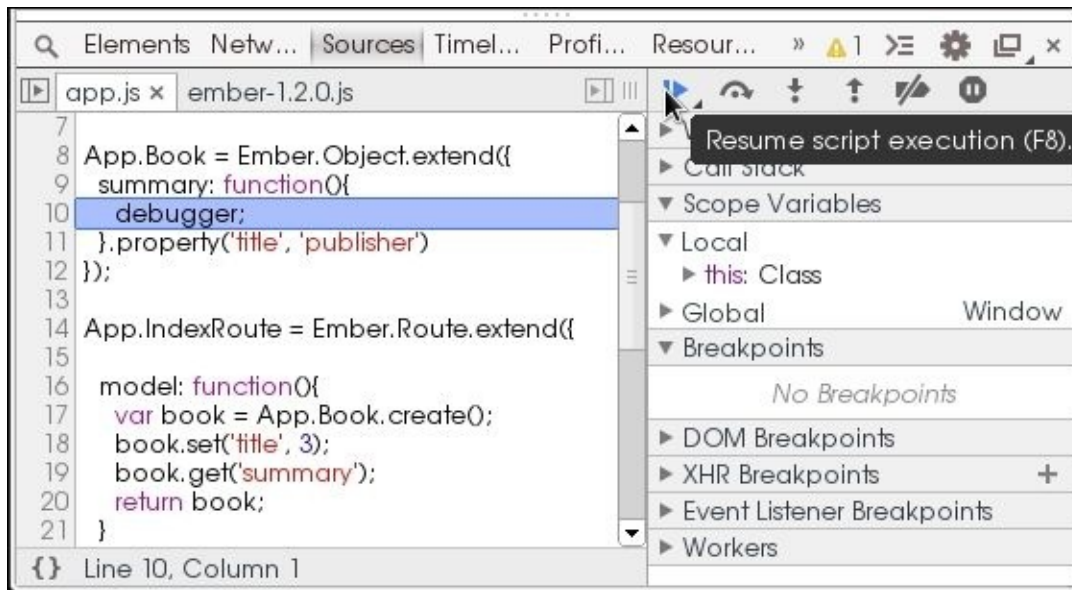
Ember.js bindings can be logged as they occur. To enable this logging, add the following code to a program before the application is initialized:

```
Ember.LOG_BINDINGS = true;
```

Most browsers allow setting breakpoints at predetermined points in an application. Breakpoints pause the execution of a program using the debugger keyword. Pausing a program can help troubleshoot problems as well as trace events. For example, we could set a breakpoint that will let us know whether a property was computed as expected:

```
App.Book = Ember.Object.extend({
  summary: function(){
    debugger;
  }.property('title', 'publisher')
});
```

This creates a breakpoint, as shown in the following screenshot:



Hitting the *F8* key resumes the execution of the application. Multiple breakpoints could be set up to trace the execution of an event. The sidebar on the right-hand side of the developer tools could then be used to enable, disable, or inspect these points.

Router and routes

When an application transitions from one route to another, it may be necessary to trace these events in the case of misbehavior. Enabling this behavior is easy:

```
var App = Em.Application.create({
  LOG_TRANSITION: true
});
```

More detailed logging can be enabled by additionally passing the `LOG_TRANSITIONS_INTERNAL` option as `true`:

```
var App = Em.Application.create({
  LOG_TRANSITIONS_INTERNAL: true
});
```

Even with this simple application, running it will log the following transition:

```
Transitioned into 'index'
```

The application controller houses two useful pieces of information about the current application state. To get the current application route name, we will reference this from the application controller, shown as follows:

```
var currentRouteName = this
  .controllerFor("application")
  .get("currentRouteName");
Ember.Logger.log(currentRouteName); // post.like
```

The full path of this current route could be looked up appropriately, as follows:

```
var currentPath = this
  .controllerFor("application")
  .get("currentPath");
Ember.Logger.log(currentPath); // user.post.like
```

Any instantiated route can be referenced from the application container, shown as follows:

```
App.__container__.lookup("route:index");
```

Templates

As we've seen time and again, templates can be looked up from the `Ember.TEMPLATES` object; for example:

```
Ember.TEMPLATES['index'];
```

Breakpoints can also be set right from templates! For example, consider that we have an index template defined as follows:

```
<script type='text/x-handlebars' id='index'>
  {{#link-to 'books'}}books{{/link-to}}
  {{#link-to 'pens'}}pens{{/link-to}}
</script>
```

We may want to inspect the rendering of this template by using the debugger expression:

```
<script type='text/x-handlebars' id='index'>
  {{#link-to 'books'}}books{{/link-to}}
  {{debugger}}
  {{#link-to 'pens'}}pens{{/link-to}}
</script>
```

Logging from the template is also possible using the `log` expression:

```
{{log model}}
```

This logs the route's model to the browser's console.

Controllers

A specific controller can be looked up globally via the main application container:

```
App.__container__.lookup("controller:index");
```

This application container registers classes to be instantiated by the application, which can, in turn, be referenced. Note that the preceding example should be used for debug purposes only. Controller dependency should instead be used to access other controllers from routes and controllers, as shown in the following example:

```
App.ApplicationController = Em.Controller.extend({
  title: 'My app'
});

App.IndexController = Em.Controller.extend({
  needs: [
    'application'
  ],
  actions: {
    save: function(){
      var title =this.get('controllers.application');
      console.log(title);
    }
  }
});
```

Finally, we can enable logs that will indicate generation of controllers by passing another option during application instantiation, as shown in the following example:

```
App = Ember.Application.create({
  LOG_ACTIVE_GENERATION: true
});
```

Views

Instantiated views have unique IDs and can therefore be looked up accordingly, as follows:

```
Ember.Logger.log(Ember.View.views['ember1']);
```

Just as with routes, we can also log view events on route transitions. This may be useful in cases where we need to verify whether registered view classes are being used. This behavior can be enabled as follows:

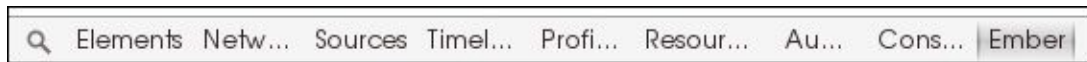
```
var App = Ember.Application.create({  
  LOG_VIEW_LOOKUPS: true  
});
```

Using the Ember.js inspector

An Ember.js application can be inspected via a browser extension that is available for Chrome, Opera, and Firefox. This extension lets you inspect objects in your application from an Ember.js tab that is created in the developer tools. To get started in Chrome, you'll need to do the following:

1. Visit `chrome://flags` and ensure **Experimental Extension APIs** is enabled.
2. Install the extension at <https://chrome.google.com/webstore/detail/ember-inspector/bmdblncegkenkacieihfhpfpppoconhi>.
3. Restart Chrome.
4. Open your Ember.js application and press the `Ctrl + U` keys to launch the developer tools.

An **Ember** tab should have been created next to the **Console** tab, as shown in the following screenshot:



From the sidebar, clicking on **View Tree** gives detailed information about the current state of the application, as shown in the following screenshot:



The next tab shows all the routes, views, controllers, and templates registered in the application. Here is a screenshot taken from the **Todos** application used in the previous chapter:

View Tree	Route Name	Route	Controller	Template	URL
/# Routes	application	Applicati...	Applicati...	application	
Data	loading	Loading...	LoadingController	loading	#/loading
Promises	error	ErrorRoute	ErrorController	error	#!/unused_dummy_error_path_route_a...
Info	todos	TodosR...	TodosC...	todos	
	todos.loading	TodosLo...	TodosLoadingCont	todos/loading	#/loading
	todos.error	TodosEr...	TodosErrorControl	todos/error	#!/unused_dummy_error_path_route_to...
	todos.complete	TodosC...	TodosC...	todos/complete	#/complete
	todos.incomplete	TodosIn...	TodosIn...	todos/incomplete	#/incomplete
	todos.index	TodosIn...	TodosIn...	todos/index	#!/

If an application uses Ember.js data, the **Data** tab will display all the loaded models:

View Tree	Model Type	Id	Title	Complete
/# Routes	App.Todo (3)	1	Bake cake	true
Data		2	Read Ember.js book	false
Promises		3	Create an Ember.js application	false
Info				

Client-side tracing

When developing an Ember.js or any other MVC application, it may be wise to trace events that occur in the application. Tracing events has the benefit of yielding data that becomes meaningful when presented as graphs. A simple tracer could be implemented by logging the timestamp of predetermined points of an ongoing event. For example, let's create an application that traces progress in loading models from the server:

```
App.ApplicationRoute = Ember.Route.extend({
  trace: function(event){
    var timestamp = Date.now();
    var data = {
      timestamp: timestamp,
      event: event
    };
    Ember.$.ajax('/logs', {
      type: 'POST',
      data: data
    });
  },
  model: function(){
    this.trace('load-application-model');
    return this.getJSON('/books');
  },
  setupController: function(controller, model){
    this._super(controller, model);
    this.trace('load-application-model');
  }
});
```

This will produce logs similar to the following code:

```
{
  timestamp: 1396376883120,
  event: 'load-application-model'
}
{
  timestamp: 1396376883130,
  event: 'load-application-model'
}
```

Graphing this data could help us gain an insight into the performance of our application.

Error management

In addition to saving logs back to the server, we could also POST any errors that could occur in the application by the following signature:

```
Ember.onerror = function(error) {  
  var data = {  
    stack: error.stack,  
    event: 'error'  
  };  
  Ember.$.ajax('/logs', {  
    type: 'POST',  
    data: data  
  });  
};
```

Summary

We just learned how to log events as well as debug bottlenecks in our Ember.js applications. A lot of development time could be saved as a result of the proper logging and tracing of events in client-side applications. In the next chapter, we'll learn how to write and run tests for our applications.

Chapter 10. Testing Your Application

Testing is an important activity conducted on any software project. Tests automate bug checks and ensure that new features not only work as expected but also don't introduce undesired behavior. Ambitious Ember.js projects, therefore, need to be well tested in order to guarantee their stability and ensure satisfying user experiences. Therefore, by the end of this chapter, we should be able to:

- Test object-computed properties
- Test object observers
- Test controllers
- Test views
- Test components
- Test user journeys

Writing tests

Ember.js supports writing the following two common types of tests:

- Unit
- Integration

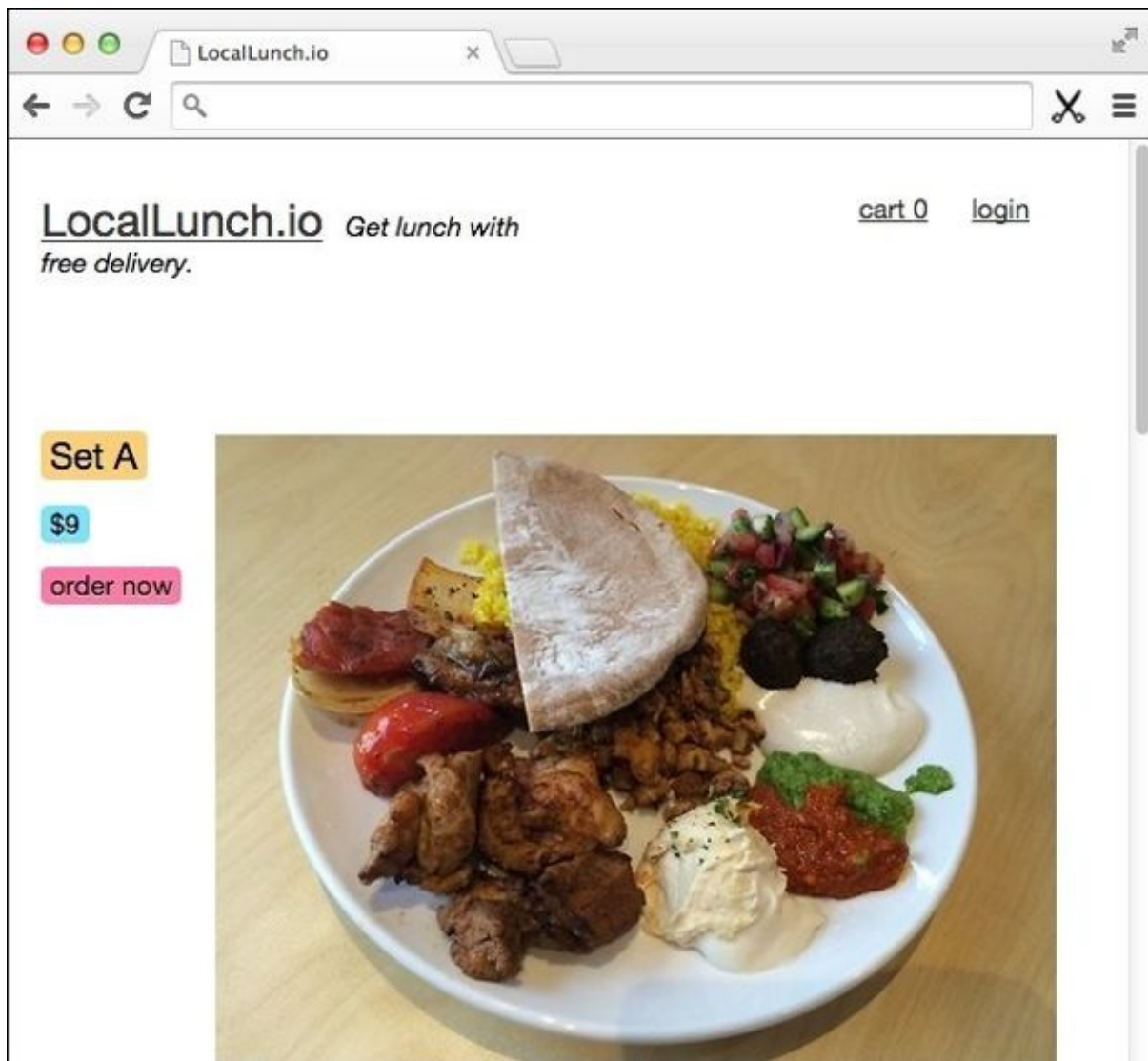
Unit tests test a specific attribute of a class (or instance) defined in an application. For example, think of scenarios such as the following:

- A created user object has a name
- A user's full name is computed properly from their first and last name
- A book model is validated properly before being saved to the server
- An observer reacts to changes correctly

Integration tests, on the other hand, test user journeys and important application workflows; for example:

- Only authenticated users can access the app
- Submitting a form persists the form data to a store, and redirects the user back to a listing
- Clicking on a checkout button adds the product to a cart

In this chapter, we will be testing a simple implementation of a typical e-commerce site that you can load via the `index.html` file located in the chapter sample.



The site in the preceding screenshot has the following features:

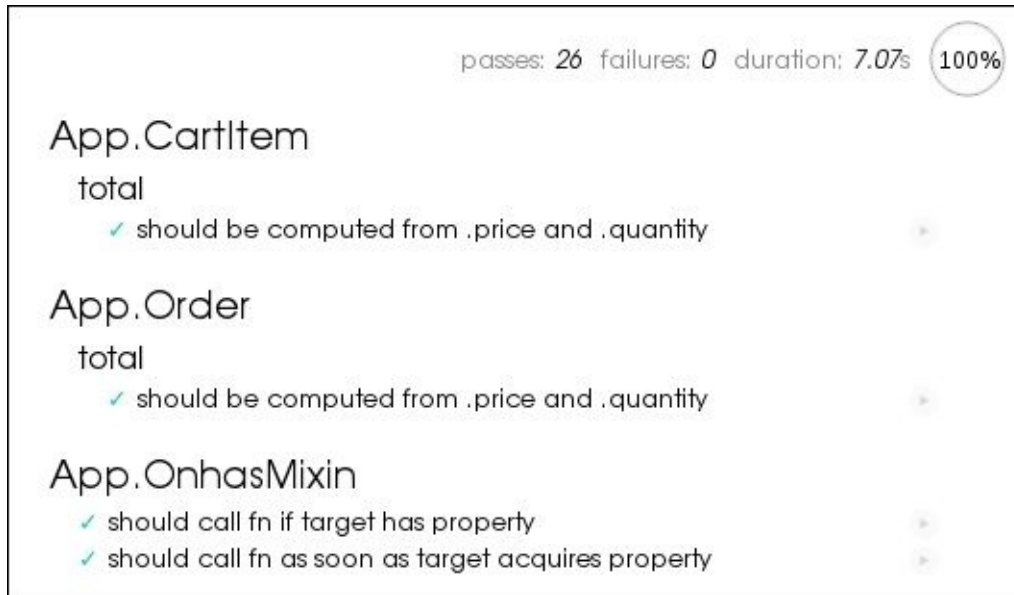
- Users can order meals to be delivered at specified locations
- Admins can log in to the site to add new meals
- Admins can see orders

To test the admin interface, log in with the username admin and password pass.
The application has been tested with the help of the following libraries:

- Mocha.js (<http://visionmedia.github.io/mocha>): This is a testing library
- Sinon.js (<http://sinonjs.org/>): This is the spies, stubs, and mocks library

- Chai.js (<http://chaijs.com>): This is an assertion library
- The Ember mocha adapter (<https://github.com/teddyzeenny/ember-mocha-adapter>)

Loading `test.html` in the browser runs the two types of tests located at `test.integration.js` and `test.unit.js`, as shown in the following screenshot:



If we examine the test loader file content, we see that the testing framework required the following element to be created:

```
<div id='mocha'></div>
```

This is the element in which the test report was rendered. We also needed to create our application's root element in the same way:

```
<div id='ember'></div>
```

Near the bottom of the file, the application scripts were loaded in order, as follows:

```
<script src="lib/jquery.js"></script>
<script src="lib/bootstrap.min.js"></script>
<script src="lib/handlebars.js"></script>
<script src="lib/ember.js"></script>
```

```
<script src="lib/ember-data.js"></script>
<script src="app.js"></script>
<script src="fixtures.js"></script>
```

Finally, the testing libraries were loaded:

```
<script src='lib/mocha.js'></script>
<script src='lib/ember-mocha-adapter.js'></script>
<script src='lib/chai.js'></script>
<script src='lib/sinon.js'></script>
```

Ember.js comes with test utilities that help write these tests. These test helpers are meant to be used with any testing library of your choice. In our case, we used Mocha.js, a popular, easy-to-use library. The first task we needed to do was set up the test environment. This was done by first defining the root element of the Ember.js application. This ensured that the Ember.js application was only executed within the element and didn't affect other parts of the testing environment:

```
App.rootElement = '#ember';
```

We then needed to run the `setupForTesting` method of the application. This deferred the readiness of the application for later execution during testing. It also prevented the tests from manipulating the window's location:

```
App.setupForTesting();
```

We also needed to call another method, `injectTestHelpers`, which injected Ember.js test helpers into the test environment:

```
App.injectTestHelpers();
```

We finally loaded and executed the tests included in the two files, as follows:

```
Em.$(function() {
  mocha.run();
});

<script src='test.unit.js'></script>
<script src='test.integration.js'></script>
```

You will notice that, for each of the tests, we defined `beforeEach` and `afterEach` hooks that got called before and after the test was executed, respectively:

```
beforeEach(App.beforeEach);  
afterEach(App.afterEach);
```

For example, the default pre-test run hook used the `visit` helper to transition the app into the index route, as follows:

```
App.beforeEach = function() {  
  visit('/');  
}
```

The post-test hook, on the other hand, resets the application state after each test by destroying such instances as the Ember-data store:

```
App.afterEach = function() {  
  App.reset();  
}
```

Ember.js comes with a number of test helpers that we will be using to help us in writing tests.

Asynchronous test helpers

These helpers are used to perform asynchronous operations. This means that, if used, the next bunch of tests needs then to be wrapped in a run loop function.

They include the following:

- `visit(url)`: This performs an asynchronous application transition into the provided URL route.
- `fillIn(selector, text)`: This is used to asynchronously set the value of an input that matches the given selector with the given text.
- `click(selector)`: This is used to trigger a click event on an element that matches the given selector. This is good for the `triggerEvent(selector, "click")` helper.
- `keyEvent(selector, type, keyCode)`: This is used to trigger a key event on the given selector.
- `triggerEvent(selector, type, options)`: This is used to trigger other DOM events on the given selector.

Synchronous test helpers

These helpers are used to perform synchronous operations. They include the following:

- `find(selector, context)`: This helper is used to perform an element selection within the given optional context
- `currentPath()`: This is used to get the current application's route path
- `currentRouteName()`: This is used to get the name of the current application route
- `currentURL()`: This is used to get the URL of the current route

Wait helpers

There is currently only one helper of this type: `andThen`. It is used to run a block of test operations after the previous asynchronous operations have been completed.

Writing unit tests

Unit testing involves testing object-computed properties, observers, and method calls.

Testing computed properties

Let's start by looking at the first tested object-computed property in the chapter sample, the `total` property of `App.CartItem`, that is applied from

`App.TotalMixin`:

```
App.CartItem = Em.Object.extend(App.TotalMixin, {
  product: null,
  quantity: null,
  price: null
});
```

When a user clicks on the order button of a meal, we expect the cart to be filled by the new item. We also expect the item's `total` property to be incremented, which is ascertained as follows:

```
describe('App.CartItem', function() {

  describe('total', function(){

    beforeEach(App.beforeEach);

    afterEach(App.afterEach);

    it('should be computed from .price and .quantity', function() {

      andThen(function(){
        var order = App.CartItem.create();
        order.get('total').should.equal(0);
        order.set('price', 10);
        order.get('total').should.equal(0);
        order.set('quantity', 1);
        order.get('total').should.equal(10);
        order.set('quantity', 2);
        order.get('total').should.equal(20);
        order.set('price', 20);
        order.get('total').should.equal(40);

      });

    });

  });

});
```

```
});
```

First, we created an order and verified that its total defaulted to zero. We then updated its quantity and price and ascertained that the total was computed properly in each case. We also did the same for the `App.Order` model, this time creating the order from the store:

```
var store = App.__container__.lookup('store:main');  
var order = store.createRecord('order', {});
```

Note how the store instance was referenced from the main application container.

Testing method calls

The app contains a mixin `App.OnhasMixin` that defines a function `onhas` that invokes a given callback if the property being bound is defined or as soon as it gets set. This is comparable to a situation where you can either buy a new computer now if you have the money or wait till you get a pay check. We started by testing the first case:

```
var object = Em.Object.createWithMixins(App.OnhasMixin, {
  id: 1
});
object.onhas('id', done);
```

In this case, the function fired as the object already contained the `id` property. We simply needed to pass the callback `done` provided by the Mocha test runner. The second test case asserts that the callback is called only when the bound property gets set as follows:

```
var object = Em.Object.createWithMixins(App.OnhasMixin, {
  id: null
});
object.onhas('id', andThen.bind(null, done.bind(null, null)));
object.set('id', 1);
```

Testing observers

One of the observers, `App.UserController#storeUser`, stores the username of the current logged-in user to the local storage:

```
storeUser: function(){
  var username = this.get('content.username');
  window.localStorage.setItem('user', username);
}.observes('content.username')
```

To test the observer, we first cleared any stored user in the local storage in the pre-test hook:

```
window.localStorage.setItem('user', null);
```

We then created the user controller, set the user, and asserted that the user is actually stored in the browser's local store:

```
var controller = App.UserController.create({
  content: {}
});

// spy
var spy = sinon.spy(controller, 'storeUser');

controller.set('username', 'username-1'); // 1
window.localStorage.getItem('user').should.equal('username-1');
```

We also set up a spy using **Sinon.js** to record the observer invocations:

```
spy.callCount.should.to.equal(2);
spy.restore();
```

The spy, as shown in the preceding code, enabled us to verify that the observer subscribed properly to the property changes.

Writing integration tests

We now already know that integration tests test the import workflows and user journeys in an application. The Ember.js framework is well tested; reviewing these tests can help a great deal in learning how to write tests for your application. Many features such as bindings and observers are already tested, so you will be writing more integration tests than unit tests. The integration tests in the chapter sample cover virtually all user interactions in the application. We will first go through all consumer-related cases, the first of which ascertains that users can see the available meals on the first page:

```
find('.products li').length.should.equal(4);
```

This example shows the use of the synchronous `find` helper, which returned the elements that listed the meals. We also checked whether the meal attributes were being displayed correctly to the user:

```
// name label

find('.products li:nth-of-type(1) .product-name')
  .text()
  .trim()
  .should
  .equal(product.get('name'));

// price label

find('.products li:nth-of-type(1) .product-price')
  .text()
  .trim()
  .should
  .equal('$'+product.get('price'));

// order button

find('.products li:nth-of-type(1) .product-add-to-cart')
  .text()
  .trim()
  .should
  .equal('order now');
```

Our next check verified that the cart link on the navigation bar indicated that the cart was empty:

```
find('.nav-cart').text().should.equal('cart 0');
```

Next to the cart link, we also ascertained that users were able to see the login link:

```
find('.nav-login').text().should.equal('login');
```

Finally, we checked whether the customer was also able to see the site branding:

```
find('.brand').text().should.equal('LocalLunch.io');
```

The next set of test cases verified that users would be able to add products to the cart. Before each test case, we triggered a click event on the checkout button of the first meal to add the meal to the cart:

```
click('.products li:nth-of-type(1) .product-add-to-cart');
```

The first test checked whether the desired meal was actually added to the cart:

```
var store = App.__container__.lookup('store:main');  
var controller = App.__container__.lookup('controller:cart');
```

```
store  
  .find('product', 1)  
  .then(function(product){  
  
    controller.get('length').should.equal(1);  
  
    var item = controller.objectAt(0);  
    item.get('product.id').should.equal(product.get('id'));  
    item.get('price').should.equal(product.get('price'));  
    item.get('quantity').should.equal(1);  
  
  });
```

We first queried this product from the store and then asserted that it was indeed added to the cart controller. We also asserted that the cart item contained the expected attributes: product, price, and quantity.

By then, we expected the cart link counter indicator to be incremented:

```
find('.nav-cart').text().should.equal('cart 1');
```

We also expected the user to have been transitioned to the cart page:

```
currentPath().should.equal('cart');
```

The proceeding tests tested the cart page to which our user has transitioned. First, we tested that viewing an empty cart page presents an appropriate message:

```
find('.message').text().should.equal('Cart is empty.');
```

We then tested the page for when the cart was filled, by first adding two items to the cart in the pre-test hook:

```
var controller = App.__container__.lookup('controller:cart');  
var store = App.__container__.lookup('store:main');
```

```
store  
  .find('product', 1)  
  .then(function(product){  
    controller.addItem(product);  
    controller.addItem(product);  
  });
```

We used the `controller.addItem` method to add the two products to the cart. Just as in the preceding test case, we asserted that the cart link indicated that two cart items had been added to the cart, as shown in the following code:

```
find('.nav-cart').text().should.equal('cart 2');
```

The cart page contains a table that shows the cart details. It also contains additional information to the left; one of these is the cart total:

```
var store = App.__container__.lookup('store:main');  
store  
  .find('product', 1)  
  .then(function(product){  
  
    var total = product.get('price') * 2;  
    find('label.checkout')  
      .text()  
      .trim()  
      .should  
      .equal(total.toString());  
  
  });
```

The `find` helper returned the label that displayed the order total. We got its `textContent` and asserted that it was equal to the order total. We then proceeded to test the cart table by first ensuring that the relevant table header cells were displayed in the right order:

```
[
  'Product',
  'Price',
  'Quantity'
].forEach(function(cell, i){
  find(find('thead td')[i]).text().should.equal(cell);
});
```

Then we tested the table body:

```
find(find('tbody td')[0])
  .text()
  .should
  .equal(product.get('name'));
find(find('tbody td')[1])
  .text()
  .should
  .equal(product.get('price').toString());
find(find('tbody input'))
  .val()
  .should
  .equal('2');
```

The preceding code tests the first row of the table by asserting that it displays the expected values. For the third cell, we test that the input contains the expected product quantity. The user is free to adjust the product amount using this input. We test this in the next test case. First, we update the input to a new value, as follows:

```
fillIn(find('tbody input'), 4);
```

We then allow any triggered operations to finish before we assert that the quantity in the cart was updated:

```
andThen(function(){
  find('.nav-cart')
    .text()
    .should
```

```
.equal('cart 4');  
});
```

This assertion shows how to use the `fillIn` helper to asynchronously update the value of an input. We also asserted that the cart total was updated as a result:

```
var total = product.get('price') * 4;  
find('label.checkout')  
  .text()  
  .trim()  
  .should  
  .equal(total.toString());
```

Once the user is satisfied with the order details, they should be able to go to the payment page by clicking on the pay button:

```
click('.pay');  
andThen(function(){  
  currentPath().should.equal('checkout');  
});
```

Once the user is on the checkout page, if their cart is empty an appropriate message will be shown:

```
find('.message').text().should.equal('Cart is empty.');
```

If the cart is not empty, the user sees a table that summarizes the order:

```
[  
  'Name',  
  'Quantity'  
].forEach(function(cell, i){  
  find(find('thead td')[i]).text().should.equal(cell);  
});
```

The table body lists the order's product name alongside its quantity:

```
[  
  product.get('name'),  
  '2'  
].forEach(function(cell, i){  
  find(find('tbody td')[i]).text().should.equal(cell);  
});
```

The table also displays the order total:

```
var total = product.get('price') * 2;
find('.checkout').text().trim().should.equal(total.toString());
```

The table tested sits on the left-hand side of the page. The main page contains a form in which the user enters their payment information before submission. We use the `triggerEvent` helper to invoke this event:

```
triggerEvent('.form-pay', 'submit');
```

A success message is displayed to the user, notifying them that the order will be delivered to them shortly:

```
find('.message')
  .text()
  .should
  .equal('Success! Your order will arrive in 20 minutes. Thank
you.');
```

This tests the payment action defined in the checkout controller:

```
pay: function(model){
  var self = this;
  var controller = self.get('controllers.cart');
  controller.forEach(function(item){
    self.store.createRecord('order', item).save();
  });
  model.set('success', true);
  controller.set('content', []);
}
```

As shown, the cart items are converted to actual orders that the shop admin can see:

```
store
  .find('order')
  .then(function(orders){

    orders.get('length').should.equal(1);

    var order = orders.objectAt(0);
    order.get('product.id').should.equal(product.get('id'));
    order.get('quantity').should.equal(2);
    order.get('price').should.equal(product.get('price'));
```

```
});
```

As a recap, you can see the preceding test cases tested the user's journey from the index page to the checkout page through the cart page. This is what constitutes integration tests.

The next case will test the admin's user journey. Admins can get access to the admin dashboard by logging into the app on the login page with the right credentials:

```
fillIn('.input-username', user.get('username'));
fillIn('.input-password', user.get('password'));
triggerEvent('.form-login', 'submit');

andThen(function(){
  currentPath().should.equal('index');
});
```

A successful login redirects the user to the index route. We also tested the case when a user tries to access the admin dashboard with the wrong credentials:

```
fillIn('.input-username', 'username');
fillIn('.input-password', 'password');
triggerEvent('.form-login', 'submit');

andThen(function(){
  currentPath().should.equal('login');
});
```

We expect the user to remain in the login route on login failure.

Once logged in, the admin should be able to add meals as well as edit or remove existing ones. These are covered in the next tests where we first log in as an admin into the site via the login helper defined at the bottom of the file:

```
fillIn('.input-username', username);
fillIn('.input-password', 'pass');
triggerEvent('.form-login', 'submit');

andThen(function(){
  if (done){
    done();
  } else {
```

```

        visit('/');
    }
});

```

The application contains a hidden form whose file input prompts the user to select a meal image when the list-product component is clicked:

```

<script type="text/x-handlebars" id="components/list-product">
    add
</script>

<script type="text/x-handlebars">
    <form class="list-product-form">
        <input
            class='list-product-form-input'
            type="file"
            style="top: -1000px; position: absolute"
            aria-hidden="true">
        </form>
    </script>

```

The corresponding `App.ListProductComponent` defines a click event handler that gets invoked when the component is clicked:

```

var self = this;

// submit

var input = Em.$('.list-product-form input');
input.one('change', upload);

// show file chooser

input.click();

// upload

function upload(event){

    var file = input[0].files[0];
    var reader = new FileReader;
    reader.onload = post.bind(this, reader);
    reader.readAsDataURL(file);

};

```



```
// post

function post(reader){
  self.sendAction('action', reader.result);
}
```

Once the user selects a file, the preceding post method is called with a `FileReader` instance argument, from which we get the URL representation of the selected image for storage. You would obviously upload this file to a storage service such as S3 or Google Cloud Storage in an actual application, instead. We finally create the product, which gets automatically listed in the page. Since it's not possible to update the value of a file input, our next test spec partially confirms this by manually adding a product to the cart:

```
Em.run(function(){

  var controller = App.__container__.lookup('controller:index');
  controller.send('createProduct', 'data:');

  Em.run.next(function(){
    find('.products li').length.should.equal(5);
    done();
  });

});
```

Here, we triggered the `createProduct` action on the index controller, which adds a product to the cart. Note that this is the same action that is called by the `post` function. In the next run loop, we asserted that a new product was added to the listing. This product's name and price are undefined, so the next test case checks whether the admin can actually edit these products:

```
var name = 'test';
var price = 1000;

fillIn('.products li:nth-of-type(1) .product-name', name);
fillIn('.products li:nth-of-type(1) .product-price', price);

andThen(function(){
  product.get('name').should.equal(name);
  product.get('price').should.equal(price.toString());
});
```

The preceding test case showed that the admin is able to update the product's name and price via the provided corresponding inputs. In an actual app, you would probably add a save button that persisted the changes to the backend.

In the navigation bar, there's an orders link that the user can click on to transition to the orders page:

```
{{#link-to "orders" class='nav-orders'}}orders{{/link-to}}
```

The next test ascertains that clicking on the link transitions the user to the orders page:

```
click('.nav-orders');

andThen(function(){
  find('.message')
    .text()
    .should
    .equal('No new orders have been made.');
```

Lastly, the admin is presented with a message if no orders have been made yet. However, if orders have been made, the admin should be able to see a table that lists them:

```
visit('/orders');

[
  'ID',
  'Product',
  'Price',
  'Quantity',
  'Total'
].forEach(function(cell, i){
  find(find('thead td')[i]).text().should.equal(cell);
});

[
  product.get('name'),
  product.get('price').toString(),
  order.get('quantity').toString(),
  order.get('total').toString()
].forEach(function(cell, i){
  find(find('tbody td')[++i]).text().trim().should.equal(cell);
});
```

});

Summary

We just learned the various test techniques you can use to ensure the stability of your application. Since you will mostly be writing integration tests, the best approach would be to break your app into clear user journeys. Then, test any expected interactions as well as transitions. This would obviously be done when developing new features. When testing against external resources, you can use libraries such as Sinon to stub these services or extend the test timeout, shown as follows:

```
describe('visit /orders', function(){  
  this.timeout(5000);  
});
```

In the next chapter, we will learn how to build Ember.js applications that are backed by external real-time data and service resources. We will specifically learn how to use the popular **socket.io** library within Ember.js applications.

Chapter 11. Building Real-time Applications

Up until now, the applications we created did not require any real-time capabilities. However, today's modern applications strive to offer the best user experience through reduced page reloads, efficient data transfers, and improved performances. In addition, these applications might also need to send data and receive updates to and from the server as quickly as possible. There are a number of web technologies that can be used to accomplish this need:

- Adobe Flash sockets
- JSONP polling
- XHR long polling
- XHR multipart streaming
- ActiveX HTMLFile
- Web sockets
- Server-sent events
- WebRTC

In this chapter, we will learn how to use the **Socket.io** (<http://socket.io>) library, which enables bidirectional communication between web clients and servers. It does this by providing a similar API between the mechanisms just mentioned, excepting the last two. Additionally, it selects the best mechanism to use depending on a number of factors, such as browser support, among others.

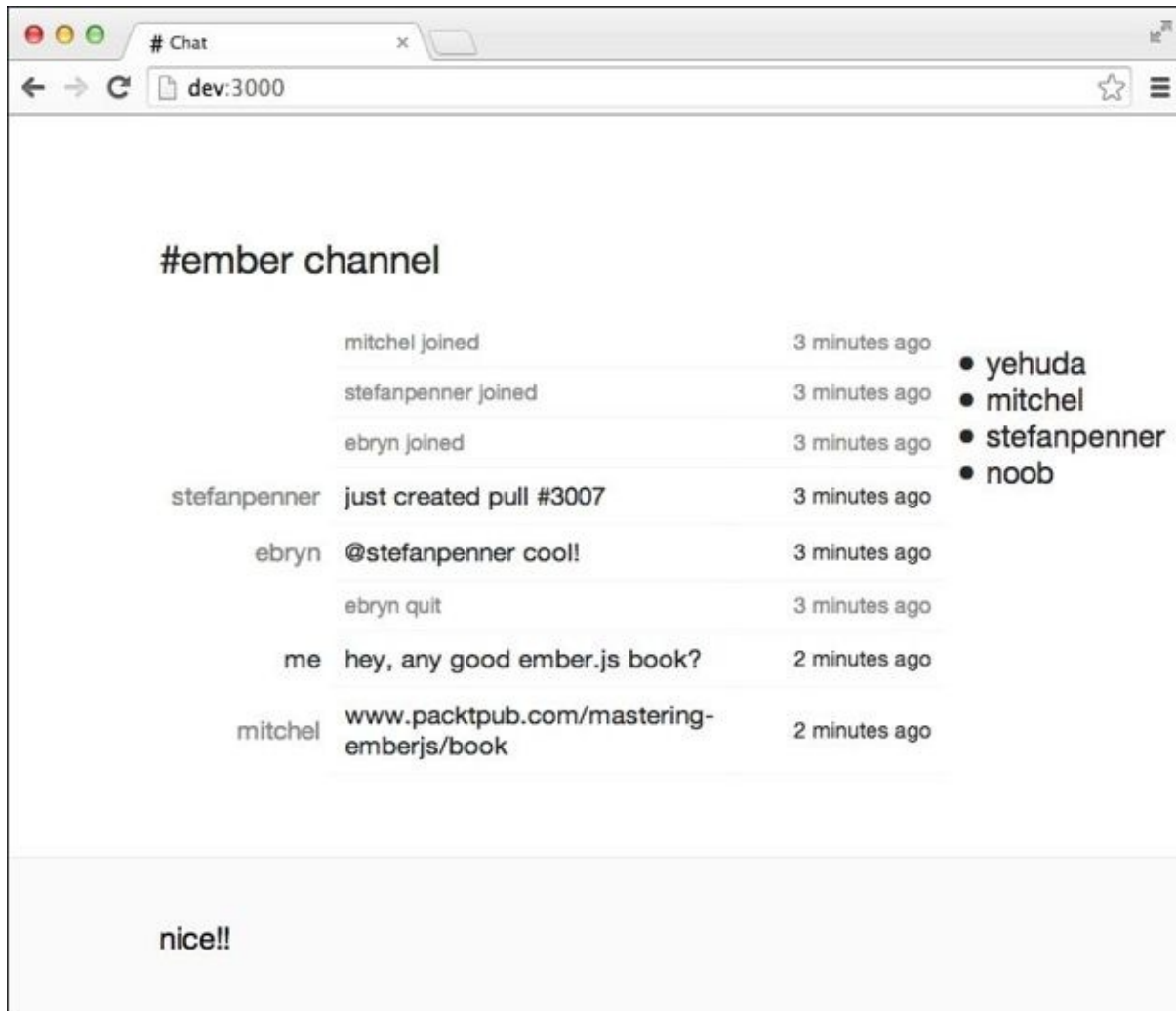
Before diving into using Socket.io, it's worth noting that server-sent events are a good option if the client app is meant to constantly receive updates from the backend while performing little or no pushes. Facebook newsfeeds and Twitter timelines are good examples of use cases that can benefit from this technology. The following resources can aid in the development of such an application:

- **Sse client library** available at <https://github.com/segmentio/sse>
- **Sse Node.js library** available at <https://github.com/segmentio/sse-stream>

WebRTC is a good choice for applications that require peer-to-peer communication, such as audio and video streaming.

Setting up Socket.io

To aid in mastering Socket.io, we will explore the bundled chapter sample that is a simple IRC-style chat application and the backend of which is built in Node.js, as shown in the following screenshot:



The only prerequisite is Node.js, which can be downloaded from <http://nodejs.org/download>. The following tests can then be performed to verify the installation:

```
node --version
v0.10.29
```

```
npm --version  
1.4.14
```

Then, boot the application with the following commands in order:

```
npm install  
node server.js
```

The backend uses the **Express.io** (<http://express.io>) library, which integrates the popular Express.js (<http://expressjs.com>) web application framework with Socket.io to reduce the setup boilerplate code to the following:

```
var app = require('express.io')();  
app.http().io();  
app.listen(3000);
```

Express.io ships with the Socket.io client library and serves it at `/socket.io/socket.io.js` when started. We therefore only need to load it together with the rest of the application files as:

```
<script src="/socket.io/socket.io.js"></script>  
<script src="/public/moment.js"></script>  
<script src="/public/jquery-1.10.2.js"></script>  
<script src="/public/handlebars-1.1.2.js"></script>  
<script src="/public/ember-1.2.0.js"></script>  
<script src="/public/app.js"></script>
```

The client library can, of course, alternatively be downloaded from <http://socket.io/download/> and served likewise. The client library is usually initialized by invoking the `connect` method on the exposed `io` global variable, as shown in the following code:

```
App = Ember.Application.create({  
  rootElement: '#wrap'  
});  
App.io = io.connect();
```

Note that there are other Socket.io libraries implemented in other languages that can be used instead of the official Node.js library.

Connecting the user

When the application is loaded at `http://localhost:3000`, the user is required to specify the handle to use before joining the chat by submitting the desired nick in the `/join <desired nick>` format. The `App.MessageField` view delegates this event to the index controller chat action as:

```
{{view App.MessageField
  required="required"
  placeholder="message"
  action="chat"
  id="message-input"
  value=controller.message}}

App.MessageField = Em.TextField.extend({
  insertNewline: function(){
    this.triggerAction();
  }
});
```

This is something that we learned in [Chapter 6, Views and Event Management](#), under the *Emitting actions from views* section. As we shall see later, the rest of the application's visibility is obscured until the user connects successfully for the first time. In the chat action of the corresponding controller, `App.IndexController`, we first make sure that the submitted message is not empty:

```
var message = self.get('message');
if (message) message = message.trim();
if (!message || message === '') return;
```

We then check to see if the user intends to join the chat:

```
var match = message.match(/\/join (\w+)/);
if (match){
  if (nick){
    self.send('tip', 'Already connected!');
  } else {
    self.send('join', match[1], view);
  }
}
```

Of course, if the user is already connected, we notify them through a tooltip.

This tooltip is displayed via the `tip` action, which takes the message to be displayed as its only argument:

```
$('.tooltip')  
  .text(msg)  
  .show()  
  .click(function(){  
    $(this).fadeOut();  
  });
```

If a nick is matched, we pass it the `join` action, which first subscribes to three server updates. The first update is triggered when a new user joins the chat, and it adds this particular user to the controller's `nicks` array to be displayed on the user, as shown in the following code:

```
App.io.on('join', function(data){  
  self.get('nicks').pushObject(data.nick);  
});
```

The next handler, on the other hand, removes a disconnected user from the collection:

```
App.io.on('quit', function(data){  
  self.get('nicks').removeObject(data.nick);  
});
```

We also subscribe to incoming messages and store them in the controller's `messages` property:

```
App.io.on('chat', function(data) {  
  self.get('messages').pushObject(App.Message.create(data));  
});
```

The `Socket.io .on` instance method is used to perform pull operations by subscribing to events emitted from the backend, as shown in the preceding three handlers. The `.emit` method, on the other hand, is used to push data to the backend. In this case, we notify the server that a new user wishes to join the chat as:

```
App.io.emit('ready', { nick: nick }, function(data) {  
});
```

The preceding call sends a `ready` event to the backend alongside the `nick`. In the

same way, the backend also sets up event handlers, the first of which listens to the ready event:

```
// server.js
app.io.route('ready', function(req) {
});
```

The listener first checks to see if the nick has been taken by an existing user and whether an existing user has taken the nick:

```
var success = nicks.indexOf(req.data.nick) === -1;
```

If it has indeed been taken, we notify the user of this failure through another update:

```
req.io.respond({
  success: success
});
```

All that the client app needs to do is display the notice via the tooltip:

```
if (data.success){
} else {
  self.send('tip', 'Nick is taken');
}
```

However, if the nick is not taken, we first store the nick in session for later reference:

```
req.session.nick = req.data.nick;
```

In a group chat application, we expect users already logged in to the chat to be notified when others join, as demonstrated by loading the app on different tabs. Therefore, when the user connects, we first add an appropriate message to the message store:

```
var message = {
  isjoin: true,
  nick: req.data.nick,
  message: req.data.nick+' joined',
  date: (new Date).toISOString()
};
messages.push(message);
```

We then *broadcast* this event to the other logged-in users:

```
req.io.broadcast('join', req.data);
req.io.broadcast('chat', message);
```

Socket.io provides a means of sending events to all active connections, except the current connection, via the `.broadcast` method. The preceding broadcast events result in the user being added to the nick listing to the right-hand side of the client application, along with the accompanying join notice:

```
<ul class="nicks">
  {{#each nicks}}
  <li> {{this}}</li>
  {{/each}}
</ul>
```

We finally notify the connecting user of the successful connection using the `respond` method:

```
req.io.respond({
  success: success,
  nicks: nicks,
  messages: messages.slice(0, -1)});
```

In addition, we send them a list of the currently logged-in users, as well as the five most recent messages. Once connected, we clear the message field and populate the users and nicks collections as:

```
self.set('message', '');
self.set('connected', true);
self.set('nick', nick);
self.get('nicks').pushObjects(data.nicks);
self.get('messages').pushObjects(data.messages.map(function(data){
  var message = App.Message.create(data);
  if (message.get('nick') === nick) message.set('isme', true);
  return message;
})));
```

When messages stream in, we make the bottom of the message list visible to the viewport by use of an observer:

```
messagesLengthDidChange: function(){
  Em.run.debounce(this, 'send', 'scrollToBottom', 200);
}.observes('messages.length'),
```

What this does is trigger the controller's `scrollToBottom` action once in a window of 200 milliseconds, as shown:

```
$('#html, body')  
  .animate({ scrollTop: $(document).height() }, 'slow');
```

Once the user connects, the rest of the application visibility is enabled, thanks to bindings:

```
<div id="content" {{bind-attr class="connected:show"}}></div>
```

In this case, the message container element acquires the `show` property, thereby resetting the opacity to 1:

```
#title,  
#content{  
  opacity: 0.1;  
}
```

```
#title.show,  
#content.show{  
  opacity: 1;  
}
```

Subsequent messages are relayed to the server and broadcasted to the other users via the `chat` action discussed:

```
if (!nick) return self.send('tip', '/join <nick>');  
  
var msg = App.Message.create({  
  isme: true,  
  message: message,  
  nick: nick  
});  
self.get('messages').pushObject(msg);  
App.io.emit('chat', msg.toJSON());  
self.set('message', '');
```

Again, on the backend, we subscribe to this event and relay the message to the other users:

```
app.io.route('chat', function(req) {  
  messages.push(req.data);  
  req.io.broadcast('chat', req.data);  
});
```

In the client app, conveniently use a table, for convenience, to list the messages:

```
{{#each messages}}
<tr {{bind-attr class="isnotice:notice isjoin:join isquit:quit
isme:me"}}>
  <td {{bind-attr class=":nick"}}>
    <span>{{#if isme}}me{{else}}{{nick}}{{/if}}</span>
  </td>
  <td class="message">
    <span>{{message}}</span>
  </td>
  <td class="date" data-timestamp="{{timestamp}}">
    <span>{{message-date date=date}}</span>
  </td>
</tr>
{{/each}}
```

You will notice that we define a message-date component that displays a message's date formatted with the aid of the Moment.js library:

```
<script type="text/x-handlebars" id="components/message-date">
  {{formattedDate}}
</script>
```

```
App.MessageDateComponent = Ember.Component.extend({

  tagName: 'span',

  didInsertElement: function(){

    var self = this;

    var id = setInterval(fn, 15000);
    this.set('intervalId', id);
    fn();
    function fn(){
      self.set(
'formattedDate', moment((new Date(self.get('date')))).fromNow()
    );
    }

  },

  willDestroyElement: function(){
    clearInterval(this.get('intervalId'));
  }
});
```

```
}  
});
```

In line with a better real-time experience, the component updates the formatted date every 15 seconds to illustrate aging.

Finally, when the user disconnects, we remove their handle from the nicks collection and notify the rest of the users:

```
app.io.route('disconnect', function(req) {  
  
  var nick = req.session.nick;  
  if (!nick) return;  
  
  var message = {  
    isquit: true,  
    nick: nick,  
    message: nick+' quit',  
    date: (new Date).toISOString()  
  };  
  messages.push(message);  
  
  var index = nicks.indexOf(nick);  
  if (index > -1) {  
    nicks.splice(index, 1);  
  }  
  
  req.io.broadcast('chat', message);  
  req.io.broadcast('quit', {  
    nick: nick  
  });  
});
```

You will notice that we referenced the saved user's handle from session.

There are a few improvements that can be made to the application, such as multichannel, emoticon, and user-avatar support, among others.

Summary

This was a short introduction to building Ember.js applications that integrate with aiding real-time web technologies. It demonstrated how easy it is to use the third-party library without spending too much time making trivial choices. We learned how to initialize the Socket.io library and subscribe and emit updates to and from the server. In the next chapter, we will learn how to componentize our applications into reusable components.

Chapter 12. Modularizing Your Project

Many Ember.js projects get complex, and so it may be necessary to modularize the project for maintainability through a combination of any of the following:

- Split the project into a number of script files and load them individually.
- Concatenate the script files into one build file. This reduces the number of requests the browser needs to make to the backend; hence, the page load time is reduced.
- Maintain reusable components that can be used on a number of projects.

There are a number of open source tools that can be used to perform such tasks. These tools may contain the following:

- A package manager that installs external reusable components
- A build process that intelligently concatenates all the project files into a single build file

The following are some of these popular tools:

- Grunt (a build tool)
- Gulp (a build tool)
- Bower (a package manager)
- NPM (a package manager)
- Browserify (a build tool)
- Ember CLI (a build tool)
- Broccoli (a build tool)
- Ember add-ons (<http://emberaddons.com>)
- Duojs (build tool and package manager based on Component and Browserify)
- Component (a build tool and package manager)

Any of these can be used to get the job done in any combination. In this chapter, we will be discussing how to use **Component** to easily manage complex Ember.js projects. It's worth noting that the ES6 module feature is being worked on to tackle these problems. Luckily, some of these tools are compliant with the specifications, thereby enabling easy migration.

Installing the Component build tool

Component is both a package manager and a build tool for client-side web projects. It provides the ability to install external project dependencies as well as organize a project into several local components that later can be built into a single build file. The chapter sample uses the tool to install and use the required project dependencies. To run the sample, we first need to install Node.js, which can be downloaded from <http://nodejs.org/download>. To run the project, simply execute make in a terminal shell. If the system doesn't have make installed, the project can still be run with the following commands, the first of which installs the tool:

```
npm install
```

Component is an NPM package and can therefore be installed by adding the dependency to `package.json`:

```
{
  "name": "2048",
  "private": true,
  "dependencies": {
    "component": "^0.19.9"
  }
}
```

Next, we install the external component dependencies and build the file using the following code:

```
./node_modules/component/bin/component install
./node_modules/component/bin/component build -n public -o public
```

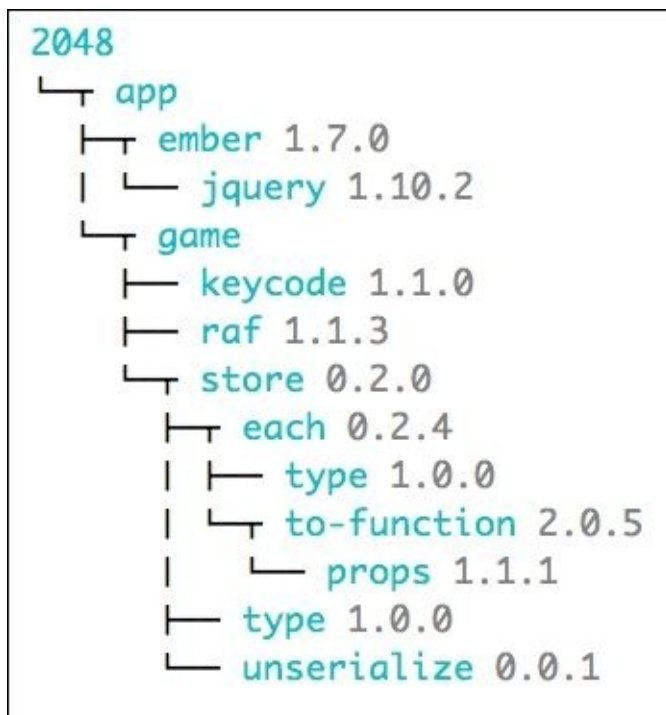
The app can finally be opened by loading `index.html` in the browser. This app is an implementation of the popular 2048 game (<http://gabrielecirulli.github.io/2048/>) by Gabriele Cirulli, and takes advantage of the Ember.js' runtime.

The open source game, shown in the following screenshot, is a good introduction to web game development:



Code organization

The Component tool requires a project to be organized into components. A component is a reusable module that can contain scripts, styles, images, templates, and fonts defined in a component.json configuration file. It can also optionally define dependencies that are themselves components. These dependencies can be local or remote. Therefore, a project can be thought of as a tree of components, as shown in the following screenshot:



In this project, the root of the project defines a component 2048, shown as follows:

```
{
  "name": "2048",
  "local": ["app"],
  "paths": ["lib"]
}
```

This component doesn't include any scripts or styles because these are contained in child components, and so we do not have to specify them. It does, however, specify that it depends on a local component named app that is found in the lib

relative directory, which in turn defines yet another local component dependency
game:

```
"local": [  
  "game"  
],
```

Installing components

A component can define remote components as dependencies. A remote component is an external Git repository that is hosted in either Github or BitBucket. Some versions of components allow the installation of components from other remote components as long as they adhere to a <username>/<repo> format. In this case, the app component defines one dependency that will be installed from <http://github.com/kelonye/ember>:

```
"dependencies": {  
  "kelonye/ember": "1.7.0"  
}
```

The game component, on the other hand, also defines three dependencies:

```
"dependencies": {  
  "yields/keycode": "1.1.0" ,  
  "component/raf": "1.1.3",  
  "yields/store": "0.2.0"  
}
```

These components can be installed into the components/ relative directory by invoking the following in the project's root directory:

```
./node_modules/component/bin/component install
```

We could also simply ask `component install` if the module is installed globally with `npm install -g component` or if `./node_modules/.bin` is added to the bash profile PATH.

Building components

Once the remote components have been installed, the project can be built by invoking the following command:

```
./node_modules/component/bin/component build
```

This concatenates scripts and styles into a `builds` folder. By default, the folder is named `build` but can be changed by passing the `-out` or `-o` flag. Also, by default, the built files are named `build.js` and `build.css`, but this can be altered using the `-name` or `-n` flag, for example:

```
component build -o public -n public
```

Loading the built files

The built files are referenced from the build folder as follows:

```
<link rel="stylesheet" href="public/public.css">

<script src="public/public.js"></script>
<script>require('app');</script>
```

Notice that the application is booted by *requiring* the app component. The app component includes an `index.js` file that gets executed in the process. Every requireable component needs to specify this file in its `.scripts` property in the configuration file. It may also specify the main file via the `.main` flag, so we can name the `index.js` file as `app.js` and then set the main flag as `"app.js"`. The script requires the remotely-installed `ember.js` component, creates the application, and defines the router:

```
require('ember');

window.App = Em.Application.create();

require('game');

App.Router.map(function(){
  this.route('game');
});
```

A good way to organize local components is to separate them by the routes or resources defined in the router; in this case, the app and game components. Each of the components contains corresponding controllers, views, models, routes, and template scripts, explicitly defined in the configuration files:

```
"scripts": [
  "index.js",
  "views.js",
  "models.js",
  "routes.js",
  "templates.js",
  "controllers.js"
],
```

These are required in the main script as follows:

```
require('./templates');
require('./models');
require('./views');
require('./controllers');
require('./routes');
```

The app component defines two templates that are required accordingly:

```
// component.json
"templates": [
  "templates/application.html",
  "templates/index.html"
],

// template.js
function compile (template){
  return Em.Handlebars.compile(require(template));
};
[
  'application',
  'index'
].forEach(function(tmpl){
  Em.TEMPLATES[tmpl] = compile('./templates/'+tmpl+'.html');
});
```

The preceding snippet registers the application and index templates to `Em.TEMPLATES`. The application, index controllers, and routes are then required accordingly, as follows:

```
App.ApplicationController = Em.Controller.extend({
});

App.IndexController = Em.Controller.extend({
  needs: ['application']
});

App.ApplicationRoute = Em.Route.extend();

App.IndexRoute = Em.Route.extend({
  redirect: function(){
    this.transitionTo('game');
  }
});
```

Notice that the index route redirects the application state to the game route.

Game logic

Before we discuss the game component, it's a good idea to discuss the game logic:

- The game is a 4 by 4 grid whose moves are made by sliding tiles using the keyboard arrow keys
- The tiles merge if they are of equal magnitude
- Each move generates a new random tile
- The objective of the game is to slide these tiles until one of them merges to a value of 2048

The grid cells are represented by the `App.Cell` model located in `models.js`:

```
App.Cell = Em.Object.extend({  
  
  x: null,  
  y: null,  
  value: null,  
  
  move: function(cell){  
    var value = this.get('value');  
    var nvalue = cell.get('value');  
    var score = nvalue+this.get('value');  
    cell.set('value', score);  
    this.set('value', 0);  
    if (value && nvalue) return score;  
    return 0;  
  },  
  
  isTile: function(){  
    return !!this.get('value');  
  }.property('value'),  
  
  is2048: function(){  
    return this.get('value') == 2048;  
  }.property('value'),  
  
});
```

A cell is considered a tile if its value is defined. When the game begins, we first populate the game with the cells:

```
App.GameRoute = Em.Route.extend({
```

```

    setupController: function(controller, model) {
        this._super();
        controller.addStartTiles();
    }
});

```

The game controller houses the cells and so we populate it accordingly:

```

App.GameController = Em.ArrayController.extend({
    needs: ['application'],
    size: 4,
    score: 0,

    addStartTiles: function () {

        var size = this.get('size');
        var rows;

        this.setProperties({
            model: [],
            tiles: []
        });

        for (var x = 0; x < 4; x++){
            for (var y = 0; y < 4; y++){
                var cell = App.Cell.create({
                    x: x,
                    y: y
                });
                this.addObject(cell);
            }
        }
    }
});

```

At this stage, we restore the saved game from the local storage using the store component:

```

// restore

var tiles = store('tiles');
var score = store('score');
var restored = tiles && score;
if (restored){

    tiles.forEach(function(tile){

```

```

        var x = Em.get(tile, 'next.x') || Em.get(tile, 'prev.x');
        var y = Em.get(tile, 'next.y') || Em.get(tile, 'prev.y');
        var value = Em.get(tile, 'next.value') || Em.get(tile,
'prev.value');
        var cell = self.find(function(_cell){
            return _cell.get('x') == x && _cell.get('y') == y;
        });
        if (cell) cell.set('value', value);
    });

    this.get('tiles').pushObjects(tiles);
    this.set('score', Number(score));

    store('tiles', undefined);
    store('score', undefined);
}

```

In addition, we cache the traversals for the four possible game move directions that we will need to make at this stage. For example, when the user makes an upward move, we will traverse cells from the left to the right and downwards:

```

var traversals = Em.Object.create();

// 'up'

rows = [];
for (var x = 0; x < size; x++){
    var row = [];
    for (var y = 0; y < size; y++){
        var cell = this.find(function(_cell){
            return _cell.get('x') == x && _cell.get('y') == y;
        });
        row.pushObject(cell);
    }
    rows.pushObject(row);
}
traversals.set('up', rows);

```

Next, we will generate the first two tiles of the game if its state is not restored:

```

// generate 2 random tiles

if (!restored){
    for (var i = 0; i < 2; i++) {
        this.get('tiles').pushObject(this.getRandomTile());
    }
}

```

```
}
```

The random tile generator simply picks a random cell and converts it into a tile by setting its value to either 2 or 4 if the game hasn't ended:

```
getRandomTile: function () {
  if (this.hasAvailableCells()) {
    var value = Math.random() < 0.9
      ? 2
      : 4;
    var tile = this.getRandomAvailableCell();
    tile.set('value', value);
    return {
      prev: tile
    };
  }
},
```

The game view sets up a listener to play the moves:

```
App.GameView = Em.View.extend({
  didInsertElement: function(){
    var self = this;
    this._super();
    $(document).on('keydown', function(event){
      event.preventDefault();
      self.get('controller').send('onMove', event.which);
    });
  },
});
```

When the user makes a play, we send the key code of the pressed key to the game controller's `onMove` action. If the game hasn't ended, the action will first determine which direction the play was made in, with the help of the `keycode` component:

```
if (self.hasEnded()) return self.endGame(false);

var dir = [
  {
    name: 'up',
    vector: {x: 0, y: -1}
  }, {
    name: 'right',
    vector: {x: 1, y: 0}
```

```

    }, {
      name: 'down',
      vector: {x: 0, y: 1}
    }, {
      name: 'left',
      vector: {x: -1, y: 0}
    }
  ].find(function(_dir){
    return keycode(_dir.name) == code;
  }));

```

The next step is to find the traversal matrix that corresponds to the direction of the play:

```

if (dir){
  var traversals = self.get('traversals.'+dir.name);
}

```

We then traverse through the cells and attempt to move the tile, if eligible:

```

try {
  var tiles = [];
  var moved;
  var traversals = self.get('traversals.'+dir.name);
  traversals.forEach(function(row){
    row.forEach(function(cell){
      if (cell.get('isTile')){

      }
    });
  });
} catch (e) {
  self.endGame(e.won);
}

```

For each of these tiles, we find the farthest new cell that it can occupy, by calling the `getNewFarthestCell` controller method:

```

var ncell = self.getNewFarthestCell(cell, dir.vector, 0);

```

This method takes the cell to move the trajectory vector and the magnitude of the movement:

```

getNewFarthestCell: function(cell, dir, mag){

```

```
},
```

The method is meant to be recursive, that is, we incrementally move the cell in the direction of the game until we find the farthest cell. First, we find the position of the current, previous, and next cells, as follows:

```
++mag;

var traversals = this.get('traversals.up');
var x = cell.get('x');
var y = cell.get('y');
var value = cell.get('value');

var nx = x + dir.x * mag;
var ny = y + dir.y * mag;

var px = x + dir.x * (mag - 1);
var py = y + dir.y * (mag - 1);
var pcell = traversals[px][py];
```

We then check to see whether the cell is off the grid:

```
var nrow = traversals[nx]; // cell is x outbound
if (!nrow) return ret();

var ncell = nrow[ny]; // cell is y outbound
if (!ncell) return ret();
```

If the new cell is indeed outbound, we return the previous cell as the new position of the tile. However, if we encounter a tile, we test whether the two can be merged:

```
// cell cannot be merged
var nvalue = ncell.get('value');
if (nvalue && value && nvalue != value) return ret();
```

If the preceding conditions aren't met, we proceed to test the next cell:

```
return this.getNewFarthestCell(cell, dir, mag);
```

Next, we check to see whether the cell can actually move to the new position:

```
if (ncell && ncell != cell){
}
```

Then, we need a state that determines whether a cell moved within that iteration for later reference:

```
if (!moved){  
    moved = true;  
}
```

If a merge has already occurred, then we need to use the cell just before it:

```
if (merged && ncell.get('isTile')) {  
    ncell = pcell;  
}
```

Finally, we move the tile and increment the game's score:

```
var score = cell.move(ncell);  
self.set('score', self.get('score') + score);
```

Since the tile has moved, we add it to the new set of tiles that would be rendered on repaint:

```
tiles.pushObject({  
    prev: cell,  
    next: ncell  
});
```

For each merge in the iteration, we need to check whether the game has been won or lost:

```
if (self.hasEnded()){  
    var err = new Error;  
    err.type = 'end-game';  
    err.won = false;  
    throw err;  
} else if (ncell.get('is2048')){  
    var err = new Error;  
    err.type = 'end-game';  
    err.won = true;  
    throw err;  
}
```

As per Gabriele's game specification, a new random tile can only be generated if any of the existing tiles move after the end of the iterations:

```
if (moved){
```

```

    tiles.pushObject(self.getRandomTile());
    self.set('tiles', tiles);
}

```

Finally, we need to check if the game has ended by catching the raised exception:

```

if (e.type == 'end-game'){
    self.set('tiles', tiles);
    self.endGame(e.won);
} else {
    console.err(err);
}

```

The method that is called displays a message overlay, indicating whether the game has been won or not:

```

endGame: function(won){
    var classes = ['game-won', 'game-over'];
    var type    = won ? classes[0] : classes[1];
    var message = won ? 'You win!' : 'Game over!';
    $('.game-message')
        .removeClass(classes)
        .addClass(type)
        .html('<p>'+message+'</p>')
        .show();
},

```

The game template, found in `templates/game.html`, reacts to changes in the bound controller. First, the New Game button's click event is bound to the `createNewGame` action:

```
<a class="restart-button" {{action 'createNewGame'}}>New Game</a>
```

```

createNewGame: function(){
    this.addStartTiles();
    this.set('score', 0);
    $('.game-message').hide();
}

```

The score label is bound to the controller's score property:

```
<div class="score-container">{{score}}</div>
```

The grid is composed of cells overlaid by tiles, and so we first lay out the 16

cells as follows:

```
<div class="grid-container">
  {{#each row in traversals.up}}
  <div class="grid-row">
    {{#each cell in row}}
    <div class="grid-cell"></div>
    {{/each}}
  </div>
  {{/each}}
</div>
```

Next, we lay out the tiles:

```
<div class="tile-container">

  {{#each tiles}}
  {{#view App.TileView prevBinding='prev' nextBinding='next'}}
  <div class="tile-inner">{{view.value}}</div>
  {{/view}}
  {{/each}}

</div>
```

The main role of `App.TileView` is to animate the tile movement as well as showing the different tile shades based on their values. Now, getting a smooth slide transition is a bit tricky. Once a tile view has been inserted into the DOM, we first set the initial position of the tile in `didInsertElement`:

```
var prev = classes('prev');
var next = classes('next');
var hasNext = !!self.get('next.value');

if (!hasNext) prev.pushObject('tile-new');

self.set('tileClasses', prev.join(' '));
```

If the tile is in motion, we use the `requestAnimationFrame` component to set the new position before the next browser repaint:

```
if (hasNext){
  raf(function(){
    self.set('tileClasses', next.join(' '));
  });
}
```

Serving images and fonts

Images and fonts specified in styles can be referenced using their corresponding relative paths, as shown in the first line of the game component's style sheet:

```
@import url(fonts/clear-sans.css);
```

Component automatically resolves these paths:

```
@import url("game/styles/fonts/clear-sans.css");
```

Notice that the assets are symbolically linked to the build folder. If needed, we can prefix these paths with the required static root. For example, if Django was used to server these files, we would add a prefix flag to the build command:

```
component build -prefix /static
```

This would result in a path such as this:

```
@import url("/static/game/styles/fonts/clear-sans.css");
```

On some platforms, symbolical links may cause problems, so you can pass the copy flag to copy the files instead of linking them, as follows:

```
component build --copy
```

Summary

Component is a great tool that you can use to organize your Ember.js project. It's a great tool that can be used to install and reuse tiny components hosted in Github. As a rule of thumb, publish Ember.js components as either mixins or those that extend the `Em.Component` class. For example, the component at <http://github.com/kelonye/ember-link> is an example of a good minimal component that can be extended into any project view:

```
require('ember');

module.exports = Em.Mixin.create({
  tagName: 'a',
  href: 'javascript:',
  attributeBindings: 'href target'.w()
});
```

This way, the Ember.js community can benefit from commonly used snippets that result in faster project developments.

Index

A

- action bubbling / [Handling event actions](#)
- actions
 - emitting, from views / [Emitting actions from views](#)
- addObject(object) method
 - about / [addObject\(object\)](#)
- addObjects(objects) method
 - about / [addObjects\(objects\), pushObjects\(objects\), and removeObjects\(objects\)](#)
- addObserver() method / [Defining property observers](#)
- afterModel hook
 - about / [Redirecting state](#)
- Ajax requests
 - creating / [Making Ajax requests](#)
- App.MessageField view / [Connecting the user](#)
- App.TileView / [Game logic](#)
- application router
 - creating / [Creating the application's router](#)
- application template
 - about / [Rendering the route's template](#)
- array controller
 - about / [An array controller](#)
 - addObject(object) method / [addObject\(object\)](#)
 - pushObject(object) method / [pushObject\(object\)](#)
 - removeObject(object) method / [removeObject\(object\)](#)
 - addObjects(objects) method / [addObjects\(objects\), pushObjects\(objects\), and removeObjects\(objects\)](#)
 - pushObjects(objects) method / [addObjects\(objects\), pushObjects\(objects\), and removeObjects\(objects\)](#)
 - removeObjects(objects) method / [addObjects\(objects\), pushObjects\(objects\), and removeObjects\(objects\)](#)
 - contains(object) method / [contains\(object\)](#)
 - compact() method / [compact\(\)](#)
 - every(callback) method / [every\(callback\)](#)

- filter(object) method / [filter\(object\)](#)
- filterBy(property) method / [filterBy\(property\)](#)
- find(callback) method / [find\(callback\)](#)
- findBy(key, value) method / [findBy\(key, value\)](#)
- insertAt(index, object) method / [insertAt\(index, object\), objectAt\(index\), and removeAt\(index, length\)](#)
- removeAt(index, length) method / [insertAt\(index, object\), objectAt\(index\), and removeAt\(index, length\)](#)
- objectAt(index) method / [insertAt\(index, object\), objectAt\(index\), and removeAt\(index, length\)](#)
- map(callback) method / [map\(callback\)](#)
- mapBy(property) method / [mapBy\(property\)](#)
- forEach(function) method / [forEach\(function\)](#)
- uniq() method / [uniq\(\)](#)
- sortProperties method / [sortProperties and sortAscending](#)
- sortAscending method / [sortProperties and sortAscending](#)
- asynchronous routing
 - about / [Asynchronous routing](#)
- asynchronous test helpers
 - about / [Asynchronous test helpers](#)
 - visit(url) / [Asynchronous test helpers](#)
 - fillIn(selector, text) / [Asynchronous test helpers](#)
 - click(selector) / [Asynchronous test helpers](#)
 - keyEvent(selector, type, keyCode) / [Asynchronous test helpers](#)
 - triggerEvent(selector, type, options) / [Asynchronous test helpers](#)

B

- bound expressions
 - writing, in templates / [Writing bound and unbound expressions](#)
- bubbling action / [Registering DOM element event listeners](#)
- built-in views (components)
 - using / [Using built-in views \(components\)](#)
 - textfield / [Textfields](#)
 - textarea / [Textareas](#)
 - select menu / [Select menus](#)
 - checkboxes / [Checkboxes](#)
 - container view / [The container view](#)
 - third-party DOM manipulation libraries, integrating with / [Integrating with third-party DOM manipulation libraries](#)
- built files
 - loading / [Loading the built files](#)
- {{bind-attr .. }} helper / [Writing template bindings](#)

C

- Chai.js
 - URL / [Writing tests](#)
- checkboxes
 - about / [Checkboxes](#)
- Chrome
 - URL / [Using the Ember.js inspector](#)
- classes
 - reopening / [Reopening classes and instances](#)
- class instance methods
 - defining / [Defining class instance methods](#)
- client-side tracing
 - about / [Client-side tracing](#)
- code organization
 - about / [Code organization](#)
- comments
 - adding, in templates / [Adding comments in templates](#)
- compact() method
 - about / [compact\(\)](#)
- component
 - about / [Component](#)
- component actions
 - defining / [Defining component actions](#)
- Component build tool
 - about / [Installing the Component build tool](#)
 - installing / [Installing the Component build tool](#)
- components
 - about / [Understanding components](#)
 - defining / [Defining a component](#)
 - post input component / [Defining a component](#)
 - post date component / [Defining a component](#)
 - post rating component / [Defining a component](#)
 - user post component / [Defining a component](#)
 - post photo component / [Defining a component](#)
 - differentiating, from views / [Differentiating components from views](#)
 - properties, passing to / [Passing properties to components](#)

- element tag, customizing / [Customizing a component's element tag](#)
- element class, customizing / [Customizing a component's element class](#)
- element attributes, customizing / [Customizing a component's element attributes](#)
- events, managing in / [Managing events in components](#)
- interfacing, with rest of application / [Interfacing a component with the rest of the application](#)
- using, as template layouts / [Components as layouts](#)
- installing / [Installing components](#)
- building / [Building components](#)
- computed properties
 - defining / [Defining computed properties](#)
 - used, for rendering dynamic data from controllers / [Computed properties](#)
 - testing / [Testing computed properties](#)
- conditionals
 - writing, in templates / [Writing conditionals](#)
- contact.edit route template
 - about / [Rendering the route's template](#)
- contact/edit template / [Switching contexts](#)
- contact template
 - about / [Rendering the route's template](#)
- container view
 - about / [The container view](#)
- contains(object) method
 - about / [contains\(object\)](#)
- contexts
 - switching, in templates / [Switching contexts](#)
- controller
 - about / [Controller](#)
- controller, views
 - accessing / [Accessing a view's controller](#)
- controller dependencies
 - specifying / [Specifying controller dependencies](#)
 - about / [Specifying controller dependencies](#)
- controllers
 - defining / [Defining controllers](#)

- providing, with models / [Providing controllers with models](#)
- dynamic data, rendering from / [Rendering dynamic data from controllers](#)
- array controller / [Object and array controllers](#), [An array controller](#)
- object controller / [An object controller](#)
- state transitions / [State transitions in controllers](#)
- debugging / [Controllers](#)
- logging / [Controllers](#)
- createWithMixins method / [Using mixins](#), [Defining controllers](#), [Defining views](#)
- custom helpers
 - defining, in templates / [Defining custom helpers](#)
- custom transformations
 - creating / [Creating custom transformations](#)

D

- data-template-name attribute / [Registering templates](#)
- data store
 - creating / [Creating a data store](#)
- debounce function
 - reference link / [Observables](#)
- debugging
 - objects / [Objects](#)
 - routes / [Router and routes](#)
 - router / [Router and routes](#)
 - templates / [Templates](#)
 - controllers / [Controllers](#)
 - views / [Views](#)
- didInsertElement hook / [Integrating with third-party DOM manipulation libraries](#)
- didInsertElement hooks / [Integrating with third-party DOM manipulation libraries](#)
- DOM
 - views, inserting into / [Inserting views into DOM](#)
- DOM element event listeners
 - registering / [Registering DOM element event listeners](#)
- dynamic data, rendering from controllers
 - properties, using / [Properties](#)
 - computed properties, using / [Computed properties](#)
 - observables, using / [Observables](#)

E

- @each helper / [Defining computed properties](#)
- element attributes, components
 - customizing / [Customizing a component's element attributes](#)
- element attributes, views
 - updating / [Updating other views' element attributes](#)
- element class, components
 - customizing / [Customizing a component's element class](#)
- element class attribute, views
 - updating / [Updating a view's element class attribute](#)
- element tag, components
 - customizing / [Customizing a component's element tag](#)
- element tag, views
 - specifying / [Specifying a view's element tag](#)
- Ember-data
 - URL / [Making Ajax requests](#)
 - about / [Understanding Ember-data](#)
 - download link / [Understanding Ember-data](#)
 - DS namespace / [Ember-data namespace](#)
 - data store, creating / [Creating a data store](#)
 - models, defining / [Defining models](#)
 - relations, declaring / [Declaring relations](#)
 - one to one relation / [One to one](#)
 - records, searching / [Finding records](#)
 - store adapter, defining / [Defining a store's adapter](#)
 - store serializer, customizing / [Customizing a store's serializer](#)
 - custom transformations, creating / [Creating custom transformations](#)
- Ember.ArrayProxy
 - reference link / [sortProperties and sortAscending](#)
- Ember.Evented mixin
 - about / [Event subscription](#)
 - on method / [Event subscription](#)
 - off method / [Event subscription](#)
 - one method / [Event subscription](#)
 - trigger method / [Event subscription](#)
 - has method / [Event subscription](#)

- Ember.js
 - origin / [The origin of Ember.js](#)
 - URL / [The origin of Ember.js](#)
 - about / [The origin of Ember.js](#)
 - downloading / [Downloading Ember.js](#)
 - objects, creating / [Creating objects in Ember.js](#)
- Ember.js application
 - requirements / [Downloading Ember.js](#)
 - creating / [Creating your first application](#)
 - router / [Router](#)
 - route / [Route](#)
 - controller / [Controller](#)
 - views / [View](#)
 - template / [Template](#)
 - component / [Component](#)
 - initializing / [Initializing the application](#)
 - embedding / [Embedding Ember.js applications](#)
 - testing / [Writing tests](#)
 - admin interface, testing / [Writing tests](#)
- Ember.js components
 - about / [Understanding components](#)
- Ember.js inspector
 - about / [Using the Ember.js inspector](#)
 - using / [Using the Ember.js inspector](#)
- Ember.ObjectProxy
 - reference link / [An object controller](#)
- Ember mocha adapter
 - URL / [Writing tests](#)
- enumerable data
 - rendering, in templates / [Rendering enumerable data](#)
- error management
 - about / [Error management](#)
- event actions
 - handling / [Handling event actions](#)
- event handlers
 - registering, into views / [Registering event handlers in views](#)
- events

- managing, in components / [Managing events in components](#)
- event subscription
 - about / [Event subscription](#)
- every(callback) method
 - about / [every\(callback\)](#)
- Express.io
 - URL / [Setting up Socket.io](#)
- Express.js
 - URL / [Setting up Socket.io](#)
- Expressjs
 - URL / [Creating REST APIs](#)
- {{#each}} ... {{/each}} block expression / [Rendering enumerable data](#)

F

- Facebook React
 - URL / [Understanding components](#)
- file-picker
 - URL / [Managing events in components](#)
- filter(object) method
 - about / [filter\(object\)](#)
- filterBy(property) method
 - about / [filterBy\(property\)](#)
- find(callback) method
 - about / [find\(callback\)](#)
- findBy(key, value) method
 - about / [findBy\(key, value\)](#)
- forEach(function) method
 - about / [forEach\(function\)](#)
- form inputs
 - writing, in templates / [Writing form inputs](#)

G

- game logic
 - about / [Game logic](#)
- getJSON() method / [Providing controllers with models](#)
- getProperties method / [Accessing object properties](#)

H

- Handlebars
 - about / [Downloading Ember.js](#)
 - URL / [Writing out templates](#)
- HashLocation class
 - reference link / [Configuring the router](#)
- HistoryLocation class
 - reference link / [Configuring the router](#)
- HTTP verbs / [Making Ajax requests](#)

I

- if...else conditional / [Writing conditionals](#)
- if conditional / [Writing conditionals](#)
- images and fonts
 - serving / [Serving images and fonts](#)
- insertAt(index, object) method
 - about / [insertAt\(index, object\), objectAt\(index\), and removeAt\(index, length\)](#)
- instances
 - reopening / [Reopening classes and instances](#)
- integration tests
 - about / [Writing tests](#), [Writing integration tests](#)
 - writing / [Writing integration tests](#)

J

- jQuery
 - about / [Downloading Ember.js](#)

L

- logging
 - about / [Logging and debugging](#)
 - objects / [Objects](#)
 - router / [Router and routes](#)
 - routes / [Router and routes](#)
 - templates / [Templates](#)
 - controllers / [Controllers](#)
 - views / [Views](#)
- `{{#link-to}}...{/link-to}}` helper / [Defining route links](#)

M

- map(callback) method
 - about / [map\(callback\)](#)
- mapBy(property) method
 - about / [mapBy\(property\)](#)
- method calls
 - testing / [Testing method calls](#)
- mixins
 - about / [Using mixins](#)
 - using / [Using mixins](#)
- Mocha.js
 - URL / [Writing tests](#)
- models
 - controllers, providing with / [Providing controllers with models](#)
 - defining / [Defining models](#)
- Moment.js library
 - about / [Passing properties to components](#)
 - URL / [Passing properties to components](#)

N

- named outlets helper / [Extending templates](#)
- Node.js binary
 - download link / [Creating REST APIs](#)
- Number.toFixed helper
 - implementing / [Creating subexpressions](#)

O

- objectAt(index) method
 - about / [insertAt\(index, object\), objectAt\(index\), and removeAt\(index, length\)](#)
- object properties
 - accessing / [Accessing object properties](#)
- objects
 - creating / [Creating objects in Ember.js](#)
 - logging / [Objects](#)
 - debugging / [Objects](#)
- observables
 - used, for rendering dynamic data from controllers / [Observables](#)
- observer function / [Defining property observers](#)
- observers
 - testing / [Testing observers](#)
- observers function / [Defining property observers](#)
- on('init') method / [Defining property observers](#)
- optionLabelPath property
 - about / [Select menus](#)
- optionValuePath property
 - about / [Select menus](#)
- {{outlet}} expression / [Expressing variables](#)

P

- partial helper / [Extending templates](#)
- Polymer
 - URL / [Understanding components](#)
- promise
 - about / [Initializing the application](#)
- promises
 - about / [Asynchronous routing](#)
- properties
 - used, for rendering dynamic data from controllers / [Properties](#)
 - passing, to components / [Passing properties to components](#)
- property() function / [Defining computed properties](#)
- property bindings
 - creating / [Creating property bindings](#)
- property observers
 - defining / [Defining property observers](#)
- pushObject(object) method
 - about / [pushObject\(object\)](#)
- pushObjects(objects) method
 - about / [addObjects\(objects\), pushObjects\(objects\), and removeObjects\(objects\)](#)

Q

- queryTermDidChange / [Observables](#)

R

- real-time applications
 - Socket.io, setting up / [Setting up Socket.io](#)
- records
 - searching / [Finding records](#)
- redirect hook
 - about / [Redirecting state](#)
- relations
 - declaring / [Declaring relations](#)
 - one to one relation / [One to one](#)
- removeAt(index, length) method
 - about / [insertAt\(index, object\), objectAt\(index\), and removeAt\(index, length\)](#)
- removeObject(object) method
 - about / [removeObject\(object\)](#)
- removeObject(objects) method
 - about / [addObjects\(objects\), pushObjects\(objects\), and removeObject\(objects\)](#)
- removeObserver() method / [Defining property observers](#)
- render helper / [Extending templates](#)
- renderTemplate hook
 - about / [Rendering the route's template](#)
- reopenClass method / [Reopening classes and instances](#)
- reopen method / [Reopening classes and instances](#)
- requestAnimationFrame component / [Game logic](#)
- resource handlers
 - defining / [Defining route and resource handlers](#)
- resources
 - nesting / [Nesting resources](#)
 - serializing / [Serializing resources](#)
- REST
 - about / [Making Ajax requests](#)
- REST APIs
 - creating / [Creating REST APIs](#)
- route
 - about / [Route](#)

- route controller
 - configuring / [Configuring a route's controller](#)
- route handler
 - defining / [Defining route and resource handlers](#)
- route links
 - defining, in templates / [Defining route links](#)
- route model
 - specifying / [Specifying a route's model](#)
- route path
 - specifying / [Specifying a route's path](#)
- router
 - about / [Router](#)
 - configuring / [Configuring the router](#)
 - debugging / [Router and routes](#)
 - logging / [Router and routes](#)
- routes
 - URLs, mapping to / [Mapping URLs to routes](#)
 - debugging / [Router and routes](#)
 - logging / [Router and routes](#)
- route template
 - rendering / [Rendering the route's template](#)
- routing errors
 - catching / [Catching routing errors](#)

S

- script tags / [Registering templates](#)
- select menu
 - about / [Select menus](#)
- serialization
 - about / [Serializing resources](#)
- setProperties method / [Accessing object properties](#)
- setupController hook / [Providing controllers with models](#)
- Sinon.js
 - URL / [Writing tests](#)
 - about / [Testing observers](#)
- Socket.io
 - setting up / [Setting up Socket.io](#)
- sortAscending method
 - about / [sortProperties and sortAscending](#)
- sortProperties method
 - about / [sortProperties and sortAscending](#)
- state
 - redirecting / [Redirecting state](#)
- state transition cycle
 - about / [Understanding the state transition cycle](#)
- state transitions, in controllers / [State transitions in controllers](#)
- store adapter
 - defining / [Defining a store's adapter](#)
- store serializer
 - customizing / [Customizing a store's serializer](#)
- subexpressions
 - creating, in templates / [Creating subexpressions](#)
- synchronous test helpers
 - about / [Synchronous test helpers](#)
 - find(selector, context / [Synchronous test helpers](#)
 - currentPath() / [Synchronous test helpers](#)
 - currentRouteName() / [Synchronous test helpers](#)
 - currentURL() / [Synchronous test helpers](#)

T

- tagName property / [Customizing a component's element attributes](#)
- template
 - about / [Template](#)
- template, views
 - specifying / [Specifying a view's template](#)
- template bindings
 - writing / [Writing template bindings](#)
- template layouts
 - components, using as / [Components as layouts](#)
- templates
 - registering / [Registering templates](#)
 - inserting / [Inserting templates](#)
 - writing / [Writing out templates](#)
 - variables, expressing / [Expressing variables](#)
 - bound expression, writing / [Writing bound and unbound expressions](#)
 - unbound expression, writing / [Writing bound and unbound expressions](#)
 - comments, adding / [Adding comments in templates](#)
 - conditionals, writing / [Writing conditionals](#)
 - contexts, switching / [Switching contexts](#)
 - enumerable data, rendering / [Rendering enumerable data](#)
 - template bindings, writing / [Writing template bindings](#)
 - route links, defining / [Defining route links](#)
 - DOM element event listeners, registering / [Registering DOM element event listeners](#)
 - form inputs, writing / [Writing form inputs](#)
 - extending / [Extending templates](#)
 - custom helpers, defining / [Defining custom helpers](#)
 - subexpressions, creating / [Creating subexpressions](#)
 - views, inserting into / [Inserting views into templates](#)
 - logging / [Templates](#)
 - debugging / [Templates](#)
- tests
 - writing / [Writing tests](#)
 - unit tests / [Writing tests](#)
 - integration tests / [Writing tests](#)

- asynchronous test helpers / [Asynchronous test helpers](#)
 - synchronous test helpers / [Synchronous test helpers](#)
 - wait helpers / [Wait helpers](#)
- textarea
 - about / [Textareas](#)
- textfield
 - about / [Textfields](#)
- third-party DOM manipulation libraries
 - integrating, with built-in views (components) / [Integrating with third-party DOM manipulation libraries](#)
 - integrating with / [Integrating with third-party DOM manipulation libraries](#)

U

- unbound expressions
 - writing, in templates / [Writing bound and unbound expressions](#)
- uniq() method
 - about / [uniq\(\)](#)
- unit tests
 - about / [Writing tests](#)
 - writing / [Writing unit tests](#)
 - computed properties, testing / [Testing computed properties](#)
 - method calls, testing / [Testing method calls](#)
 - observers, testing / [Testing observers](#)
- unless...else conditional / [Writing conditionals](#)
- URLs
 - mapping, to routes / [Mapping URLs to routes](#)
- user
 - connecting / [Connecting the user](#)

V

- variables
 - expressing, in templates / [Expressing variables](#)
- view helper / [Extending templates](#)
- view layouts
 - specifying / [Specifying view layouts](#)
- views
 - about / [View](#)
 - defining / [Defining views](#)
 - controller, accessing / [Accessing a view's controller](#)
 - template, specifying / [Specifying a view's template](#)
 - element tag, specifying / [Specifying a view's element tag](#)
 - element class attribute, updating / [Updating a view's element class attribute](#)
 - element attributes, updating / [Updating other views' element attributes](#)
 - inserting, into DOM / [Inserting views into DOM](#)
 - inserting, into templates / [Inserting views into templates](#)
 - event handlers, registering / [Registering event handlers in views](#)
 - actions, emitting from / [Emitting actions from views](#)
 - components, differentiating from / [Differentiating components from views](#)
 - logging / [Views](#)
 - debugging / [Views](#)

W

- W3C Web Components
 - about / [Component](#)
- wait helpers
 - about / [Wait helpers](#)
 - andThen / [Wait helpers](#)
- web component
 - about / [Understanding components](#)
- willDestroy hook / [Integrating with third-party DOM manipulation libraries](#)
- willInsertElement hook / [Integrating with third-party DOM manipulation libraries](#)
- {{#with}}...{/with}} helper / [Switching contexts](#)