INSTANT

Short | Fast | Focused

Ember.js Application Development How-to

Your first step in creating amazing web applications

Marc Bodmer

[PACKT] PUBLISHING

Instant Ember.js Application Development How-to

Your first step in creating amazing web applications

Marc Bodmer

[PACKT]
PUBLISHING

BIRMINGHAM - MUMBAI

Instant Ember.js Application Development How-to

First published: February 2013

Production Reference: 1050213

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-338-1

www.packtpub.com

Credits

Author
Marc Bodmer

Reviewer
Diego Muñoz Escalante

Acquisition Editor
Martin Bell

Commissioning Editor
Maria D'souza

Technical Editor
Hardik Soni

Copy Editor
Alfida Paiva

Project Coordinator
Michelle Quadros

Proofreader
Maria Gould

Graphics
Aditi Gajjar

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

Cover Image
Sheetal Aute

About the Author

Marc Bodmer is a recent graduate with an honors degree in Computer Science. He is based in Toronto, Ontario and will be working as a frontend developer at 500px (`www.500px.com`) from May 2013. Marc loves experimenting with all kinds of various web frameworks as well as creating and contributing to open source projects. He loves attending developer conferences to keep himself updated on web technologies and meeting developers with great ideas.

I would like to acknowledge Dhiren Audich and Robert Kuncewicz for inspiration and for offering advice on various web development frameworks.

About the Reviewer

Diego Muñoz Escalante holds a Masters in Computer Science from the University of Huelva, Spain where, he was born and raised. Well versed in Python, Django, PHP, JavaScript, Ember.js, Angular.js, and others, Diego has contributed to many open source projects. He has taught and completed research at both the University of Huelva as well as the University of Western Ontario in Canada. Most recently he has been focused on Ember and Angular.js and maintains an adapter that connects Django backends with Ember applications. Diego is also collaborating to improve the features of ember-touch to develop touch-ready mobile applications with Ember. At Shiny Ads, where he currently works, Diego is translating a classic PHP + jQuery platform into a modern Angular.js set of applications. Passionate about programming, research, and open source technologies, Diego currently lives in the city of Toronto, Ontario.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

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 access, read and search across 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 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 nine entirely free books. Simply use your login credentials for immediate access.

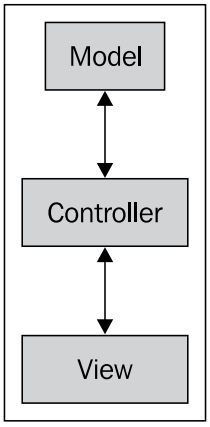
Table of Contents

Preface

Ember.js is a frontend MVC JavaScript framework that runs in the browser. It is for developers who are looking to build ambitious and large web applications that rival native applications. Ember.js was created from concepts introduced by native application frameworks, such as Cocoa. Ember.js helps you to create great experiences for the user. It will help you to organize all the direct interactions a user may perform on your website. A common use case for Ember.js is when you believe your JavaScript code will become complex; when the code base becomes complex, problems about maintaining and refactoring the code base will arise.

MVC stands for model-view-controller. This kind of structure makes it easy to make modifications or refactor changes to any part of your code. It will also allow you to adhere to Don't Repeat Yourself (DRY) principles. The model is responsible for notifying associated views and controllers when there has been a change in the state of the application. The controller sends CRUD requests to the model to notify it of a change in state. It can also send requests to the view to change how the view is representing the current state of the model. The view will then receive information from the model to create a graphical rendering.

If you are still unclear on how the three parts interact with each other, the following is a simple diagram illustrating this:



Ember.js decouples the problematic areas of your frontend, enabling you to focus on one area at a time without worrying about affecting other parts of your application. To give you an example of some of these areas of Ember.js, take a look at the following list:

- **Navigation**: Ember's router takes care of your application's navigation
- **Auto-updating templates**: Ember view expressions are binding-aware, meaning they will update automatically if the underlying data ever changes
- **Data handling**: Each object you create will be an Ember object, thus inheriting all Ember.object methods
- **Asynchronous behavior**: Bindings and computed properties within Ember help manage asynchronous behavior

Ember.js is more of a framework than a library. Ember.js expects you to build a good portion of your frontend around its methodologies and architecture, creating a solid application architecture once you are finished with it. This is the main difference between Ember and a framework like Angular.js. Angular allows itself to be incorporated into an existing application, whereas an Ember application would have had to have been planned out with its specific architecture in mind. Backbone.js would be another example of a library that can easily be inserted into existing JavaScript projects. Ember.js is a great framework for handling complex interactions performed by users in your application. You may have been led to believe that Ember.js is a difficult framework to learn, but this is false. The only difficulty for developers lies in understanding the concepts that Ember.js tries to implement.

This book will teach you Ember.js beginner-level conventions that will provide you with solid ground knowledge of the framework. Do not worry if you do not understand everything all at once. It is a lot to take in and you should give yourself time to learn Ember.js conventions. Ember.js, in particular, favors convention over configuration.

What this book covers

Setting up Ember.js (Simple) will explain what software will have to be installed in order to work with Ember.js. It will also explain what backend structure Ember.js works well with. It will show you how to set up the starter kit for Ember.js, and will explain the basic file structure of your application.

Creating an Ember model/object (Simple) will explain the Ember object model. You will learn how to represent data in a structured way for your application to easily perform actions on them.

Enhancing an Ember object (Simple) will show you some common ways of extending your Ember objects. How to create computed properties, mixins, and observers for your objects will be described. The method of setting and getting objects will be shown as well.

Creating an Ember controller (Simple) will show you how to create controllers and connect them with the rest of your application. The recipe will show you basic properties associated with the controller. The different categories that controllers fall under will be explained for you as well.

Handlebar HTML templates (Simple) explains Handlebars, the default templating engine that Ember.js uses. How to create templates and what their purpose is will be explained as well as the syntax of Handlebars and how to create dynamic HTML templates. Various included helpers in Handlebars will be shown.

Creating an Ember view (Simple) will explain to you how views in Ember work. It will also explain the responsibilities of a view and how views handle user events. Useful properties that are included with views will be shown as well.

Routing for your application (Medium) will explain how the state of your application will change when using the Ember.js router. It will show how user actions impact the state and how you can incorporate different state changing methods into your application. Transitions and outlets are the main topics explained here.

Common parts of an application (Medium) will show you how to create common parts of a web application now that you have a solid foundation with Ember.js. Action helpers for your views and special keys within controllers will be shown. These parts can also be expanded on once you are finished with the tutorials in this book.

Handling external data (Advanced) will show you how you can incorporate external data from an API into your application. An Ajax method will be used for this recipe. It will also touch upon Ember Data, which is a library the Ember.js is working on to simplify data in more complex applications.

What you need for this book

The programming language used in this book will be JavaScript. The browser that will be used to run the JavaScript, HTML, and so on will be Google Chrome. Other browsers such as Firefox and Safari should work just as well. The operating system that can be used is Windows, Mac OS X, or Linux. A text editor other than Notepad, such as Notepad++, VIM, or Sublime Text should be used for proper formatting of the code.

Who this book is for

The target audience for this book is frontend developers who realize that their frontend code base has gotten too large to maintain effectively and properly, and are looking for a complete framework solution. Ember.js is also a possible solution for implementation of small features in an existing application.

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 are shown as follows: "We create a namespace called `MovieTracker` where we can access any necessary Ember.js components."

A block of code is set as follows:

```
MovieTracker.Router = Ember.Router.extend({
  root: Ember.Route.extend({
    index: Ember.Route.extend({
      route: '/'
    })
  })
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<script type="text/x-handlebars" data-template-name="action_panel">
    <h2>Actions</h2>
      <a class="btn btn-large action_button" {{action
      "toggleWatched" target="actionPanelView"}}>
        <i class="icon-ok"></i>
        Toggle watched
      </a>
  </script>
```

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: "The preceding code should display **A Comedy Movie has a rating of 5** in the console."

Warnings or important notes appear in a box like this.

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`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

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/support`, 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.

Instant Ember.js Application Development How-to

Welcome to *Instant Ember.js Application Development How-to*. This book will walk you through building the frontend structure for a sample application using Ember.js. You will learn how Ember implements the MVC architecture, and key parts of the Ember API will be explained as well.

Setting up Ember.js (Simple)

A good place to start experimenting with Ember.js is the official starter kit created by the developers of Ember.js. This starter kit will provide you with the framework that you need to get up and running with Ember.js quickly. We can use this starter kit on a Windows, Mac OS X, or Linux operating system. A backend for Ember.js is definitely a possibility. The most commonly used backend for Ember, **Rails**, has gems to add Ember.js support. **Node.js** and **Django** have adapters to incorporate Ember.js as well. For the purposes of this book, we will sacrifice proper application architecture in the interest of time.

Getting ready

The browser used for the purposes of this book will be Google Chrome. Any text editor for editing our JavaScript and HTML files will work in this case. Ember provides a starter kit that will help us create our sample application.

Download the latest starter kit from `https://github.com/emberjs/starter-kit/downloads` or use the `starter-kit.zip` file provided. Currently, the latest released version of Ember.js is 1.0.0-pre.2. Any version of this level and above should work.

1. Once downloaded, unzip it to wherever you like. The folder that should be created is named `starter-kit`.
2. Navigate inside that folder and you should, at the minimum, see the subfolders `css` and `js` as well as the file `index.html`.
3. Go to the `js` directory and create folders called `views`, `models`, and `controllers`.

How to do it...

We can now make modifications to the existing files and add new ones as needed to build out our application foundation.

1. We can begin by sorting out our `app.js` file in the `js` folder to get rid of the code that will be put into separate files later on. Go ahead and copy and paste the following code into `app.js`:

```
var MovieTracker = Ember.Application.create();
```

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.

2. Inside the `controllers` folder, we can add a file called `application.js` with the following lines:

```
MovieTracker.ApplicationController =
Ember.Controller.extend();
```

3. In your `index.html` file, include the following line:

```
<script src="js/controllers/application.js"></script>
```

4. Now that we have a good starting point in `app.js`, we can create a simple router. Create a new file called `router.js`. In this file, we can copy and paste the following code:

```
MovieTracker.Router = Ember.Router.extend({
  root: Ember.Route.extend({
    index: Ember.Route.extend({
      route: '/'
    })
  })
});
```

5. In your `index.html` file, include the following line:

```
  <script src="js/router.js"></script>
```

6. The last thing we need to do is create some sort of view for our application. You can copy and paste the following code into a new file called `application.js` inside the `views` folder:

```
MovieTracker.ApplicationView = Ember.View.extend({
  templateName: 'application'
});
```

   In `index.html` add this line:

```
<script src="js/views/application.js"></script>
```

7. Also add the following Handlebars template after your `<body>` tag in `index.html`:

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

How it works...

The `css` folder contains the stylesheet of the application we are going to create. This stylesheet contains **HTML 5 Boilerplate** to normalize styles regardless of what browser you are using.

The `js` folder contains a subfolder named `libs` and the `app.js` file. `libs` is for storing any external libraries that you will want to include into your application. `app.js` is the JavaScript file that contains your Ember application structure.

`index.html` is a basic HTML index file that will display information in the user's browser. We will be using this file as the index page of the sample application that we will be creating.

We create a namespace called `MovieTracker` where we can access any necessary Ember.js components. `Initialize()` will instantiate all the controllers currently available with the namespace. After that is done, it injects all the controllers onto a router.

We then set `ApplicationController` as the rendering context of our views. Your application must have `ApplicationController`, otherwise your application will not be capable of rendering dynamic templates.

`Router` in Ember is a subclass of the Ember `StateManager`. The Ember `StateManager` tracks the current active state and triggers callbacks when states have changed. This router will help you match the URL to an application state and detects the browser URL at application load time. The router is responsible for updating the URL as the application's state changes.

When Ember parses the URL to determine the state, it attempts to find `Ember.Route` that matches this state. Our router must contain `root` and `index`. You can think of `root` as a general container for routes. It is a set of routes.

`index` can be renamed if you wish. If we load a page with the URL following `/`, Ember will match it to `root.index` and will transition your application to the `root` state and then the `index` state within `root`.

An Ember view is responsible for structuring the page through the view's associated template. The view is also responsible for registering and responding to user events.

`ApplicationView` we are creating is required for any Ember application. The view we created is associated with our `ApplicationController` as well. The `templateName` variable is the name we use in our `index.html` file. The `templateName` variable can be changed to anything you wish.

We then reference this `templateName` variable in our view.

The sample application we will be making throughout the book is a movie tracker. It will allow you to store and organize the movies you have watched and want to watch. This application has a simple concept, and will allow you to further extend it once you are done following the recipes in this book.

Creating an Ember model/object (Simple)

An object or a model is a way to manage data in a structured way. In other words, they are a way of representing persistent states in your application. In Ember.js, almost every object is derived from the `Ember.Object` class. Since most objects will be derived from the same base object, they will end up sharing properties with each other. This allows the observation and binding to properties of other objects. That being said, we can see where some of the powerful features of Ember, such as computed properties and bindings, come from.

How to do it...

We will create a basic model object and then create a subclass of this object.

1. In your `models` folder, go ahead and create the following file called `movie.js`.

```
MovieTracker.Movie = Ember.Object.extend({
  id: null,
  title: null,
  watched: false,
  rating: 0
});
```

2. In `index.html` add the following line:

```
<script src="js/models/movie.js"></script>
```

3. We can now test out our model by creating a new movie object in the browser's JavaScript console and printing out the title. Copy and paste the following code in your browser's console. You should see the string **The Action Movie** being printed out to the console.

```
var actionMovie = MovieTracker.Movie.create({
  title: "The Action Movie"
});

actionMovie.get("title");
```

4. We can also extend the descendants of the `Ember.Object` class to create our own class. In `movie.js`, we can append the following code:

```
MovieTracker.ActionMovie = MovieTracker.Movie.extend({
  genre: "action"
});
```

How it works...

We define our models and then create new objects based on those models. In the previous example, we created an object in our application called `Movie`, which has an ID, a title (if it has been watched), and its rating.

Our `ActionMovie` will have the same properties as `Movie`, but will add one extra property. The `genre` property will define what genre the movie falls under. Go ahead and copy and paste the following code into your browser's console:

```
var actionMovie = MovieTracker.ActionMovie.create({
  title: "An Action Movie",
});
```

```
actionMovie.get("genre");
```

The preceding code should print out **action** to the console. It creates a new `ActionMovie` object, which inherits the `genre` property.

The get and set methods

You should always use the Ember `get` and `set` methods when accessing model properties. This ensures that Ember's bindings and observers associated with the `get` and `set` methods can keep the data in sync throughout your application.

Enhancing an Ember object (Simple)

Now that we have our basic Ember objects, we can begin to make them more powerful. The Ember.js documentation promotes three features. Two of these features deal with models. One of them is auto-updating templates and the other two are bindings and computed properties. In this recipe, we are only concerned with bindings and computed properties.

How to do it...

Now that we have some basic Ember objects defined in our application, we can go ahead and add more advanced functionality to them.

1. We can represent actors in our application using bindings, so create a new file in the `models` folder called `actor.js` and paste the following code:

```
MovieTracker.Actor = Ember.Object.extend({
  filmBinding: 'MovieTracker.Movie.title'
});
```

2. In `index.html` add the following line:

```
<script src="js/models/actor.js"></script>
```

How it works...

Bindings are used to maintain synchronization of properties between two different objects.

Computed property functions act like properties of our model. They also work with bindings. Computed properties are used to build new properties by combining other properties. They should not contain any application behavior. In most cases, multiple calls to one computed property should always return the same value.

An observer will trigger when the observed property changes and is especially useful when you need to perform some action after a binding has synchronized.

You will notice there is a property named `filmBinding`. When Ember.js sees a model property ending with the string `Binding`, Ember will automatically create a bound property for you (in this case it is called `film`).

There's more...

Computed properties, mixins, and observers give us different options for interacting with data in our application.

Computed properties

Computed properties allow us to create functions within our models.

```
MovieTracker.Movie = Ember.Object.extend({
  id: null,
  title: null,
  watched: false,
  rating: 0,
  titleAndRating: function() {
       return this.get('title') + ' has a rating of ' +
       this.get('rating');
  }.property()
});
```

Here, we add a function called `titleAndRating` that will combine the title and rating of our movie into one string. Computed properties often have dependencies on other properties. In the example we created, we can tell Ember that our `titleAndRating` function depends on the title and the rating of the movie by adding those parameters within the property parentheses.

```
MovieTracker.Movie = Ember.Object.extend({
  id: null,
  title: null,
  watched: false,
  rating: 0,
```

```
  titleAndRating: function() {
    return this.get('title') + ' has a rating of ' +
    this.get('rating');
  }.property('title', 'rating')
});
```

If we want to test this out, we can attempt to get the title and the rating of a movie we create. Copy and paste the following code into the console:

```
var comedyMovie = MovieTracker.Movie.create({
  title: 'A Comedy Movie',
  rating: 5
});

comedyMovie.get('titleAndRating');
```

The preceding code should display **A Comedy Movie has a rating of 5** in the console.

Mixins

Another useful part of Ember.js is something called mixins. A mixin is an object that defines a set of functions relating to a type. In our case, we can copy and paste the following code into the top of `movie.js`.

```
WatchedMixin = Ember.Mixin.create({
  isWatched: function() {
    var title = this.get('title'),
      watched = this.get('watched');

    return('Has ' + title + ' been watched? ' + watched);
  }
});
```

We can then add the mixin into our `Movie` object by passing it as the first argument to `.extend` or `.create`.

```
MovieTracker.Movie =
Ember.Object.extend(MovieTracker.WatchedMixin, {
  id: null,
  title: null,
  watched: false,
```

In the future, mixins should probably be contained in their own file, as you will most likely be using them for multiple objects.

We can then copy and paste the following into the console to demonstrate our mixin:

```
var watchedMovie = MovieTracker.Movie.create({
  title: 'A Watched Movie',
  rating: 5,
  watched: true
});

watchedMovie.isWatched();
```

The preceding code should print out **Has A Watched Movie been watched? true**. We created a `Watched` mixin object that contains a function that can be added to any Ember object. The extended object will then have these functions accessible to it.

Observers

Ember.js can also make use of something called observers. We can change our Ember `Movie` object to include an observer.

```
MovieTracker.Movie = Ember.Object.extend(MovieTracker.WatchedMixin,{
  id: null,
  title: null,
  watched: false,
  rating: 0,
  titleAndRating: function() {
    return this.get('title') + ' has a rating of ' +
    this.get('rating');
  }.property('title', 'rating'),
  titleChanged: function() {
    console.log('Title changed!');
  }.observes('title')
});
```

Refresh the page to load the new changes, then recreate our comedy movie with the following code:

```
var comedyMovie = MovieTracker.Movie.create({
  title: 'A Comedy Movie',
  rating: 5
});
```

If we were to change the title with something like:

```
comedyMovie.set('title', 'Comedy Movie 2');
```

Then **Title changed!** would be displayed as an output to the console.

We added an observer to the title, so whenever the title changes, the corresponding function (`titleChanged`) is triggered.

Creating an Ember controller (Simple)

A controller is an object capable of storing the application state. Application views connect to controllers and translate the current state of the controller into HTML. Another way of thinking of controllers is that they act as a direct representation of models for your views and can send CRUD commands to the models on behalf of the views.

How to do it...

We can now create a controller for a part of our application.

1. Add the following controller into `application.js` located in the `controllers` folder:

```
MovieTracker.moviesController = Ember.ArrayController.create({
    content: [],
    init: function(){
        this._super();

        var list = [
            MovieTracker.Movie.create({
                title: 'Movie 1',
                rating: 4
            }),
            MovieTracker.Movie.create({
                title: 'Movie 2',
                rating: 5
            })];

        this.set('content', list);
    }
});
```

2. We can also create a controller that will allow us to keep track of a movie that has been selected by the user in our application.

```
MovieTracker.selectedMovieController =
Ember.ObjectController.create({
    selectedMovie: [],

    select: function(item) {
        this.set('selectedMovie', item);
```

```
    },

    toggleWatched: function() {
       this.selectedMovie.toggleProperty('watched');
    }
});
```

How it works...

In Ember.js, controllers are split into three different categories:

- `ArrayController`
- `ObjectController`
- `Controller`

`ArrayController` is used for managing a collection of objects, that is, a collection of movies and actors. Each `ArrayController` uses a `content` property to store its data. Properties and methods in this controller will have a proxy that will allow access to its `content` property. In `moviesController`, we create an empty content array and then populate it with some example data. The `this._super()` call lets you access the `init` function of the parent class that you are overriding.

`ObjectController` is essentially the same as `ArrayController`, except that it is used for one object as opposed to a collection. For our application, we are going to need a controller for changing a movie to *watched* when the user clicks on a corresponding button for this action. In `selectedMovieController`, we are only concerned with one specific `Movie` object. The function inside this controller will change the `watched` property of the associated movie to true if it was false previously and vice versa.

The `Controller` class in Ember is used when you have a controller that is not a proxy. In other words, the controller does not take care of an object or an array.

If we look back at the code in our `application.js` within the `controllers` folder, we added the following:

```
MovieTracker.ApplicationController = Ember.Controller.extend();
```

This line does not need any arrays or objects contained within it, so we simply assign it the `Controller` class. This controller handles the controls at the application level.

We can paste the following line into the browser's console to check that `moviesController` is in proper working order:

```
MovieTracker.moviesController.get('length');
```

The preceding line should print out **2** to the console.

Handlebar HTML templates (Simple)

Handlebars is a logic-less semantic templating engine that will interact with our views to render HTML elements. Handlebars is the default engine for Ember.js. Handlebars is an extension of the **mustache templating engine**. At runtime, Ember.js will compile all Handlebar templates. A Handlebars expression takes the form of `{{contents}}`.

Apart from plain HTML, templates embed:

- **Expressions**: An expression takes data from a controller, view, or context and places it into HTML and automatically keeps the data in sync.
- **Outlets**: Outlets are placeholders for other templates. The router places different templates in your outlets as your application state changes. Handlebars has the `{{outlet}}` helper for doing this.
- **Views**: Views handle user events.

How to do it...

Handlebars are HTML templates with embedded Handlebars expressions. These templates go within the `<script>` tags so that your browser can recognize them.

1. In `index.html`, add the following within the `<body>` tags.

```
<script type="text/x-handlebars" data-template-
name="movie_title">
  <h2>{{title}}</h2>
</script>
```

2. Global paths can also be defined in your templates. We can change the template as shown in the following code snippet:

```
<script type="text/x-handlebars" data-template-
name="movie_title">
  <h2>{{MovieTracker.selectedMovieController.title}}</h2>
</script>
```

3. Handlebars include conditional expressions. A common use for this is when you want to display part of a template only if a property exists. The template can be further modified as follows:

```
{{#if movie}}
  <h2>{{movie.title}}</h2>
{{else}}
  <h2>No movie</h2>
{{/if}}
```

4. We can also render a template only if the value is false, undefined, null, and so on.

```
{{#unless hasBeenWatched}}
  <h2>You should watch this movie!</h2>
{{/if}}
```

5. The default context for our template blocks is `Ember.View` associated with the block. The context is the object where properties are looked upon. If we want to invoke a template with a context other than `Ember.View`, we can use the `{{#with}}` helper.

```
{{#with movie}}
  <h2>{{title}}</h2>
{{/with}}
```

6. At some point in our application, we will most likely want to bind HTML attributes into our templates. Common use cases are when you want an `src` attribute for an image and/or when you want an `href` attribute for a link.

```
<a {{bindAttr href="url"}}>More Details</a>
```

7. We can add a child view to a parent by using the `{{view}}` helper. This helper takes in a path to a `view` class.

```
MovieTracker.ActorView = Ember.View.extend({
  templateName: 'actor'
  mainActor: "John Smith"
});
```

```
{{#with movie}}
  <h2>{{title}}</h2>
  {{view MovieTracker.ActorView}}
{{/with}}
```

```
<script type="text/x-handlebars" data-template-
name="actor">
  {{view.mainActor}}
</script>
```

8. You can also use view templates inline. You can think of this as assigning views to portions of a page. It allows you to encapsulate event handling for one part of a page. We can rewrite the previously mentioned code as the following:

```
MovieTracker.ActorView = Ember.View.extend({
  mainActor: "John Smith"
});
```

```
{{#with movie}}
  <h2>{{title}}</h2>
  {{#view MovieTracker.ActorView}}
    {{view.mainActor}}
  {{/view}}
{{/with}}
```

How it works...

All the features previously described are binding-aware. This means that as values in your templates change, the HTML will be updated automatically. If you do not want a value to automatically update, you can use the unbound view helper.

This basic template has `data-template-name` that is typically used as a reference by a view. The `{{title}}` expression will print the value of the `title` property that we send to the template using a view.

Ember determines whether a path is global or relative to the view by checking if the first letter is capitalized. This is why your `Ember.Application` name should start with a capital letter.

The # symbol within `{{}}` means that it is a block expression. These expressions require a closing expression (such as `{{/if}}` in the previous example). If the `movie` property is not false, undefined, or null, then it will display the movie title.

```
{{#with movie}}
  <h2>{{title}}</h2>
{{/with}}
```

We changed the *context* of our template block to `movie` in this block of code, and thus we were able to reference the title directly.

```
<a {{bindAttr href="url"}}>More Details</a>
```

When we used `bindAttr` for our URL, the template took the `url` property from the view and inserted it as an `href` attribute.

```
<a href="http://moredetails.com">More Details</a>
```

`{{view.mainActor}}` within the `actor` template would render as **John Smith**.

The `{{view}}` helpers in Handlebars can include optional parameters to help in displaying information.

- `class`: Used to assign class names.
- `tagName`: By default, new instances of `Ember.View` create the `<div>` elements. Use `tagName` to override this.
- `contentBinding`: Binds the specified content as the context of the template block.

As you are developing your templates, remember that Handlebars is capable of rendering the following elements from different parts of your application with different scopes:

- `Controller`
- `View`
- `Context`

These parts would look like the following in Handlebars:

```
title from controller: {{controller.title}}
title from view: {{view.title}}
title from context: {{title}}
```

Creating an Ember view (Simple)

Views will help you to translate low-level browser events into semantic events. These semantic events will allow the Ember router to adjust your application into the proper state. Views can be created and destroyed at render time as well as during the run loop of your application. The `Ember.View` class includes a variety of properties for creating different types of views. A few of the common ones that you could use are as follows:

- `tagName`: The default HTML tag is `div`, but it can be changed using `tagName`.
- `classNames`: Sets the HTML class attribute with the provided array of strings.
- `classNameBindings`: Sets the HTML class attribute based on an array of properties for the view. These properties can be computed properties, allowing you to perform logic to determine whether class names should be assigned or not.
- HTML attributes such as `href` can also be added using the `attributeBindings` property.

Views in Ember are more about user interaction and less about markup. As you read on, keep in mind that Ember.js gives you a lot of flexibility in the way you define your templates and views and the interaction between them. For example, you can define the tag name or class names within a view class constructor or you can define them in your Handlebars. As your application grows, you will most likely find better ways of defining classes that are relevant to your needs.

It is up to you the way you want to structure your application depending on what you are doing. If a view *must* be a certain tag, then it should go in the view constructor. If you find that your views are often set to other tags, then defining tags in your Handlebars saves you from having to make lots of extended view objects.

How to do it...

Views in Ember can be created in many different ways. One way to create your view is to define the view directly in your JavaScript, and then display it on the page using a Handlebars view helper.

1. We can create something like this for our `MovieTracker` application. If you recall from earlier, we created `moviesController` as shown in the following code, but with changed movie titles:

```
MovieTracker.moviesController = Ember.ArrayController.create({
    content: [],
    init: function(){
        this._super();

        var list = [
            MovieTracker.Movie.create({
                title:'The Avengers',
                rating: 4
            }),
            MovieTracker.Movie.create({
                title: 'Spiderman',
                rating: 1
            })];

        this.set('content', list);
    }
});
```

2. We can now create a view in `views/application.js` that will send the necessary data to Handlebars for rendering.

```
MovieTracker.MovieListingsView = Ember.View.extend({
    templateName: 'movie_listings',
    controllerBinding: 'MovieTracker.moviesController'
});
```

3. Now that we have our view bound to a controller, we can create the template that will render the data in the view. The following code will go inside `index.html` after the main application `<script>` tags, right before the section where we include all our scripts.

```
<script type="text/x-handlebars" data-template-
name="movie_listings">
<ul class="movie_listings">
{{#each movie in content}}
```

```
        <li>
           <h4>{{movie.title}}</h4>
            <br/>
            <h4>{{movie.rating}}</h4>
         </li>
       {{/each}}
    </ul>
    </script>
```

How it works...

Views have three main responsibilities. They are as follows:

- Render HTML elements
- Handle DOM events
- Trigger actions to change the state of your application

The Handlebars templating engine is used for rendering HTML elements.

```
this.set('content', list);
```

In `moviesController`, we set the `content` key to our `list` array with the preceding command. `set` takes in the parameters `(key, value)`.

`'movie_listings'` is the name of the template that we are going to sync our data to. We can bind this view to our `moviesController` controller.

With the following piece of code, we are using the Handlebars notation we learned about earlier. We iterate through the `moviesController` content object containing a list of movies, and display them as an HTML list.

```
{{#each movie in content}}
   <li>
      <h4>{{movie.title}}</h4>
      <br/>
      <h4>{{movie.rating}}</h4>
   </li>
{{/each}}
```

There's more...

There are multiple ways to handle DOM events. The two main ones are:

- Ember-dispatched events
- Handlebars action helper

For Ember-dispatched events, we can implement a function that will respond to user events such as the following:

```
MovieTracker.ToggleWatched = Ember.View.extend({
  templateName: 'movie-watched',
  click: function(event){
    alert('I pressed movie-watched!');
  }
});
```

The `jQuery.event` object is passed into our function as an argument.

A Handlebars action helper can be used to handle custom events created by you.

```
<div {{action "mark_watched"}}>
  I have watched this movie
</div>
```

This will call the `mark_watched` function of the template associated with `Ember.View` when the `div` tag is clicked. There are various other events that can be linked to the `action` helpers as well.

If we wanted to pass a value from the template context to the handler, we could have done something as follows:

```
{{#each movie in content}}
<div {{action "mark_watched" movie}}>
  I have watched this movie
</div>
{{/each}}
```

The preceding code will call the `mark_watched` method with a `jQuery.Event` object containing a movie.

A view that references the properties of a model should be bound to one controller in charge of those properties. You can also bring in data from elsewhere if needed. You can do this by binding one view's controller to another by using the `connectControllers` function.

Absolute paths

When dealing with Ember views, an absolute path in a class is a path pointing to a global variable. You may see other examples where the view will contain a `contentBinding` property.

```
MovieTracker.MovieTrackerView = Ember.View.extend({
    templateName: 'movie_main_details',
    controllerBinding: 'MovieTracker.moviesController',
    contentBinding: 'controller.content'
});
```

This example will still work technically but it does not work when considering a good design. Taking `contentBinding` out of our view and inserting it into a template instead will solve this problem.

```
{{view MovieTracker.MovieTrackerView
contentBinding="controller.content"}}
```

Moving the `contentBinding` variable into the Handlebars template binds the content of the particular view, `MovieTrackerView`, to the specified controller's `content` property. This promotes reusability in our code and makes testing easier because the view is isolated.

Routing for your application (Medium)

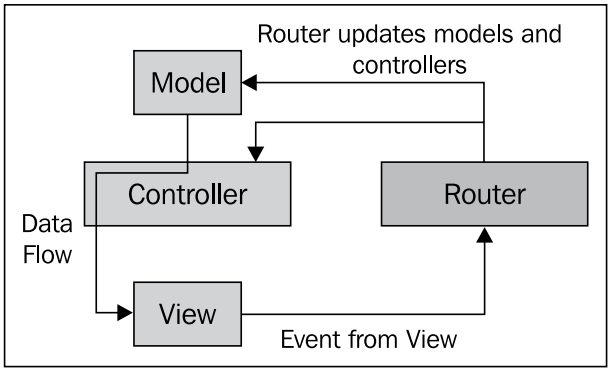
The router in Ember is responsible for changing the state of your application in response to the user's actions. As mentioned earlier, Ember's router is a subclass of the more general purpose `StateManager`. States are the main features of an Ember application. Through the use of states, we can answer questions such as:

- Is there a user logged in?
- What model object is the user currently looking at?
- Is the user currently editing content on the site?

The state of your application can change when one of the following three reasons occur:

- The user fires an event from one of your views
- The user loads the page for the first time or changes the URL
- The data changes, triggering a linked observer and listener

Ember associates a URL for every state in the router.



The preceding diagram shows you the high-level overview of how data flows in your application with the router involved.

Getting ready

As we saw earlier, every Ember application should have a basic router set up. Ember applications without a router are allowed, but not common. Your router should look like the following:

```
App.Router = Ember.Router.extend({
  root: Ember.Route.extend({
    index: Ember.Route.extend({
      route: '/'
    })
  })
});
```

How to do it...

There are a few different ways you can approach routing in your application. It depends on the type of application you are creating and the complexity of it.

Outlets are sections of your view that change during runtime based on the actions of the user. The outlet specifies that the router will make the decision of what to place in that area of the template.

1. Handlebars has an `{{outlet}}` helper that will render a view in response to a state change. Any template can have any number of outlets as long as you name them (such as `{{outlet aName}}`). Create an outlet in your routing using the following code:

```
MovieTracker.Router = Ember.Router.extend({
  root: Ember.Route.extend({
    index: Ember.Route.extend({
      route: '/',
      connectOutlets: function(router){
      router.get('applicationController').connectOutlet
      ('details', MovieTracker.moviesController.content);
      }
    })
  })
});
```

```
MovieTracker.DetailsController =
Ember.ArrayController.extend();

MovieTracker.DetailsView = Ember.View.extend({
    templateName: 'details'
});

<script type="text/x-handlebars" data-template-
name="application">
    {{outlet}}
 </script>

<script type="text/x-handlebars" data-template-
name="details">
    {{#each movie in controller}}
      {{movie.title}}
    {{/each}
  </script>
```

2. The only other thing that is different is the `details` template.

```
{{#each movie in controller}}
    {{movie.title}}
{{/each}}
```

You can notify your application of a change in state by using state transitions.

3. We want to make our application capable of transitioning between states. We can take our previous router, and change our routes as shown in the following code:

```
MovieTracker.Router = Ember.Router.extend({
  root: Ember.Route.extend({
    index: Ember.Route.extend({
      route: '/',
      connectOutlets: function(router){
      router.get('applicationController').connectOutlet
      ('details', MovieTracker.moviesController.content);
      }
    })
  })
});
```

This code will become as follows:

```
MovieTracker.Router = Ember.Router.extend({
  root: Ember.Route.extend({
    index: Ember.Route.extend({
      route: '/',
      redirectsTo: 'movies'
    }),

    movies: Ember.Route.extend({
      route: '/movies',
      showMovie: Ember.Route.transitionTo('movie'),
      connectOutlets: function(router){
      router.get('applicationController').connectOutlet
      ('listings', MovieTracker.Movie.find());
      }
    }),

    movie: Ember.Route.extend({
      route: '/movie',
      connectOutlets: function(router, context){
      router.get('applicationController').connectOutlet
      ('oneMovie', context);
      }
    })
  })
})
```

How it works...

Looking from the top to the bottom, you will notice that we added a `connectOutlet` function to our index router. `connectOutlet` is a callback that allows us to connect `{{outlet}}` in templates to specific views based on the state of the application. The first argument shows that we want to connect our `ApplicationController` with the Details view and controller. The last argument is the data context that we send to the view and controller.

The `connectOutlet` call will do the following task:

- Create a new instance of `DetailsView`
- Set the `content` property of `DetailsController` to the data context argument we pass in (in this case it is a list of movies)
- Make `DetailsController` the controller for `DetailsView`
- Connect `DetailsView` to the outlet in the `application` template

When Ember initializes your application, a single-shared instance of each controller is created. With most cases, these instances are accessed from the router so that they are placed as a property of the router. The only difference is that the names of the controllers are converted to lowerCamelCase. In this case, the instance of `ApplicationController` is stored as `applicationController`.

For the template we created, the content of `DetailsController` is passed along (in which we have a list of movies).

When using state transitions, the first change is the transition into the movies state by using `redirectsTo`. This was just done as a preference and there is nothing wrong with staying in the index state. The movies state now contains a `showMovie` function. `transitionTo` will do exactly as its name implies, that is, it will transition you from one state to another. The function can be triggered using an `action` helper in the template.

```
<a {{action showMovie movie href=true}}>{{movie.title}}</a>
```

The `action` helper goes within the opening tag of an element and takes three arguments. `showMovie` is the name of the action we want to send to the current state in the router and `movie` is the context. In our movie route, the `connectOutlets: function(router, context)` is passed in the router as the first argument and the context from the `{{action}}` helper as the second argument.

If your `action` helper's target does not implement the function you are trying to call, an error will be thrown as follows:

Uncaught TypeError: Cannot call method 'call' of undefined

With the following line inside the `movie` route, we set the `content` property of the `oneMovie` controller as `context`:

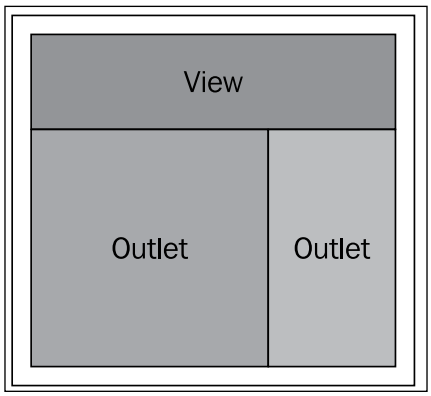
```
router.get('applicationController').connectOutlet('oneMovie',
context);
```

If your application gets large enough, you should consider breaking up your router into separate files. Each of these files would contain one router per state.

Our application will enter into the index state the first time the user navigates to it.

There's more...

Having read about views and outlets, you might be wondering when to use what.



The previous diagram is an example of the views and outlets that may exist in a web application. At the top, the view could represent a navigation bar, which stays mostly static throughout your application. The outlet on the left-hand side is an area of a template that has its child template determined at runtime based on user interaction. For example, this outlet could show a list of items. The outlet on the right-hand side could then show more details about an item when that item is clicked in the list.

Common parts of an application (Medium)

Now that we have talked about most of the basics of Ember, we can start to create some basic elements of an application.

How to do it...

As seen earlier, Ember allows us to make a function act like a property when we add `.property()`. Any parameters of this function will let Ember know that it has to update the value of the property when the specific parameters are updated.

1. In our `moviesController`, we are storing all the movie objects that exist in our application. We can add some *helper properties* to this controller.

```
// Controller to store Movie Objects
MovieTracker.moviesController =
Ember.ArrayController.create({
    content: [],
```

```
    // Property that adds an item to the array
    addItem: function(item) {
        this.addObject(item);
    },

    // Property that returns the length of the data array
    itemCount: function() {
        return this.get('length');
    }.property('@each'),

    unwatchedCount: function() {
      return this.filterProperty('watched',
      false).get('length');
    }.property('@each.watched')
});
```

2. If we wanted to reference any of these properties from our template, we can insert some more Handlebars code into our HTML.

```
{{MovieTracker.moviesController.unwatchedCount}}
```

3. We can also create some basic interactions in our application. We can add an anchor that will fire an event.

```
<script type="text/x-handlebars" data-template-
name="action_panel">
    <h2>Actions</h2>
      <a class="btn btn-large action_button" {{action
      "toggleWatched" target="actionPanelView"}}>
        <i class="icon-ok"></i>
        Toggle watched
      </a>
  </script>
```

4. We can then create a `selectedMovieController` controller as follows:

```
MovieTracker.selectedMovieController =
Ember.ObjectController.create({
    selectedMovie: [],

    select: function(item) {
        this.set('selectedMovie', item);
    },

    toggleWatched: function() {
       this.selectedMovie.toggleProperty('watched');
    }
});
```

5. We also need to add a relevant view.

```
MovieTracker.MovieListingsView = Ember.View.extend({
    click: function(event) {
      var content = this.get('content');
      MovieTracker.selectedMovieController.select(content);
    }
});
```

6. We can now create a view that will render the previous template and receive the `toggleWatched` event.

```
MovieTracker.ActionPanelView = Ember.View.extend({
    templateName: 'action_panel',

    toggleWatched: function(event) {
    MovieTracker.selectedMovieController.toggleWatched();
    }
});
```

How it works...

The `@each` special key will trigger if:

- Any property changes in any of the movie objects within the content array
- An item is added to or removed from the content array
- The content array is set to a different array
- `{{MovieTracker.moviesController.unwatchedCount}}`

  This code block will give us `unwatchedCount`, the number of movie objects with the `watched` property set to `false`.

We create an event using the `{{action}}` helper that will call a `toggleWatched` function in our view to toggle the `watched` property of a movie object.

```
<script type="text/x-handlebars" data-template-name="action_panel">
    <h2>Actions</h2>
      <a class="btn btn-large action_button" {{action
      "toggleWatched" target="actionPanelView"}}>
        <i class="icon-ok"></i>
        Toggle watched
      </a>
  </script>
```

In order to render the current movie, we need some sort of way to keep track of the currently selected movie in our application.

```
MovieTracker.selectedMovieController = Ember.ObjectController.create({
    selectedMovie: [],

    select: function(item) {
        this.set('selectedMovie', item);
    },

    toggleWatched: function() {
       this.selectedMovie.toggleProperty('watched');
    }
});
```

In this block of code we store a selected movie into an array. We create a `select` function that will store the selected movie. The `toggleWatched` function will toggle the `watched` property of the `selectedMovie` object.

Whenever we click on one of our movie listings, the click event will be registered in the `MovieListingsView` view and we can send the clicked object to our `selectedMovieController`. We have now successfully connected our `{{action}}` in our template to `selectedMovieController`.

You will see many different ways of creating parts of your application when looking at other tutorials and resources for Ember. There is no one *correct* way. It all depends on your application and what makes sense to you. All the sample codes have been provided for you. There are parts of the application that you can go on and continue finishing, such as:

- Making some of the expanded templates in `index.html` into reusable templates
- Finding ways to change the ratings of movies (watch out here because rating is defined as a number in the model)
- Navigating through the movie list using the button on the left-hand side and the button on the right-hand side in the navigation bar

Handling external data (Advanced)

Hardcoded data in your application is okay for demo purposes, but will most likely not work when developing real applications. Instead, we can figure out some way to retrieve external data.

How to do it...

Static data is fine for a simple application, but eventually we want to implement a way where we can import dynamic data into our application.

1. We can create a JavaScript file called `helpers.js`. In it, we will have code as shown in the following snippet:

```
MovieTracker.GetMovieItems = function() {
  MovieTracker.moviesController.addItem
  (MovieTracker.Movie.create({
        title: 'The Avengers',
        rating: 4,
        watched: false
    }));
  MovieTracker.moviesController.addItem
  (MovieTracker.Movie.create({
        title: 'Spiderman',
        rating: 1,
        watched: true
    }));
};
```

2. `App.js` then has to be modified to call this function when the application starts up.

```
MovieTracker = Ember.Application.create({
  ready: function() {
     this._super();

    MovieTracker.GetMovieItems();
  }
});
```

3. We can insert a `$.ajax()` function into our function instead, if we want to grab data from another source.

```
MovieTracker.GetMovieItems = function() {
  $.ajax({
    url: yourURL,
      dataType: 'json',
      success : function(data) {
      // Create an Ember object from your data
      // Use the addItem() we created earlier.
    }
  });
};
```

How it works...

The `GetMovieItems()` function creates the new `Movie` objects and inserts them into our content array defined in `moviesController`. This is good because now we have one function that will handle the loading of data into our application.

As you get your data, you have to create an Ember object out of it. Once it is an Ember object, you can add it to a controller. In this case, we added our data to our main data controller, `moviesController`.

At the time of writing this, the Ember.js team was working on a library called **Ember Data**. Ember Data helps out with functionality that is needed in a more complex application. In complex applications, you could be loading in models from a JSON API. You then need some way of updating and saving these models as your application changes the contents of them. Ember Data will give you a nice API to handle these calls. It will also help to do the following:

- Provide stateful data syncing
- Encode and decode properties
- Create communications between transactions, and much more

You can read more about it at `https://github.com/emberjs/data`.

Thank you for buying

Instant Ember.js Application Development How-to

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] PUBLISHING

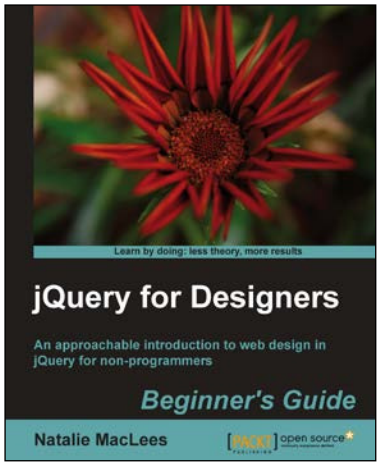
Responsive Web Design with HTML5 and CSS3

ISBN: 978-1-849693-18-9 Paperback: 324 pages

Learn responsive design using HTML5 and CSS3 to adapt websites to any browser or screen size

1. Everything needed to code websites in HTML5 and CSS3 that are responsive to every device or screen size
2. Learn the main new features of HTML5 and use CSS3's stunning new capabilities including animations, transitions and transformations
3. Real world examples show how to progressively enhance a responsive design while providing fall backs for older browsers

---



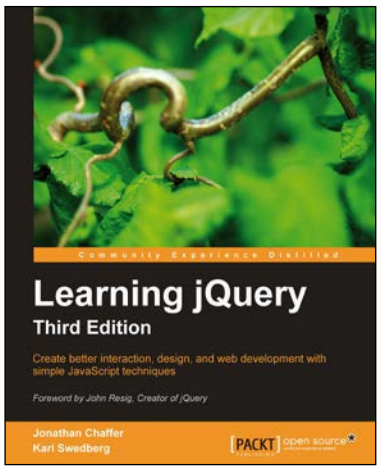
jQuery for Designers: Beginner's Guide

ISBN: 978-1-849516-70-9 Paperback: 332 pages

An approachable introduction to web design in jQuery for non-programmers

1. Enhance the user experience of your site by adding useful jQuery features
2. Learn the basics of adding impressive jQuery effects and animations even if you've never written a line of JavaScript
3. Easy step-by-step approach shows you everything you need to know to get started improving your website with jQuery

[PACKT] PUBLISHING

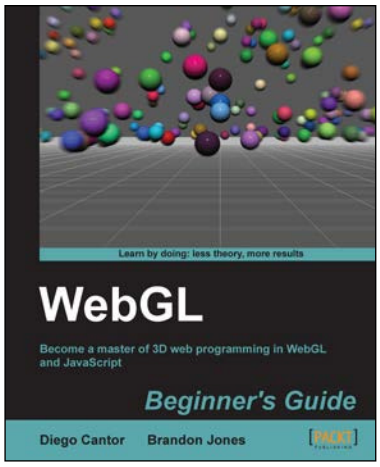


Learning jQuery, Third Edition

ISBN: 978-1-849516-54-9 Paperback: 428 pages

Create better interaction, design, and web development with simple JavaScript techniques

1. An introduction to jQuery that requires minimal programming experience
2. Detailed solutions to specific client-side problems
3. Revised and updated version of this popular jQuery book



WebGL Beginner's Guide

ISBN: 978-1-849691-72-7 Paperback: 376 pages

Become a master of 3D web programming in WebGL and JavaScript

1. Dive headfirst into 3D web application development using WebGL and JavaScript
2. Each chapter is loaded with code examples and exercises that allow the reader to quickly learn the various concepts associated with 3D web development
3. The only software that the reader needs to run the examples is an HTML5 enabled modern web browser. No additional tools needed.
4. A practical beginner's guide with a fast paced but friendly and engaging approach towards 3D web development

Please check **www.PacktPub.com** for information on our titles