Ember.js Web Development with Ember CLI

Ember.js is a JavaScript framework based on the Model View Controller design pattern. It brings proven design principles and practices to modern web-based application development and helps you focus on solving your core business problem.

Ember.js Web Development with Ember CLI is for the next generation of web developers who want to build powerful single-page web applications using the simplicity of Ember CLI and the sophistication of the upcoming Ember 2.0. Starting with an introduction to Ember.js, you will explore its object-oriented pattern, cover classes and other properties, diving into great techniques to define your routes when managing applications, and using object and array controllers to encapsulate the application display logic.

Unlike many other books that merely skim the surface, this book has a strong focus on Ember CLI, which will soon be the de facto mode for building apps with Ember.

Who this book is written for
If you are a JavaScript developer who is starting out to build ambitious Ember.js application, or a developer who has prior Ember.js experience and want to transition your application to the latest Ember.js version with Ember CLI, then this book is for you.

What you will learn from this book
- Get started with your first Ember.js application using Ember CLI and learn about its MVC pattern
- Understand the object-oriented design principles used in Ember.js including classes and structures
- Discover how to use the Ember.js templating system and default handlers
- Manage your single-page web application states using Ember.js router and customize templates
- Learn to use controllers to communicate display logic to templates
- Communicate with your backend server using Ember Data models
- Build reusable Ember.js components and learn how to customize them

Suchit Puri

Build ambitious single-page web applications using the power of Ember.js and Ember CLI.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Managing Application State Using Ember.js Routes'
- A synopsis of the book’s content
- More information on Ember.js Web Development with Ember CLI

About the Author

Suchit Puri is a full-stack software developer and holds a master's degree in software systems.

He has held senior development and technical management positions at some of the world's most prominent companies, such as Huawei, Progress Software, and ThoughtWorks. He is currently working for a Delhi-based start-up called Wingify, which is growing rapidly. He specializes in project inceptions, analysis, coding, deployments, and automation, and clients ranging from small, fast-paced start-ups to big multinationals have benefited from his expertise.

He is also a sought-after speaker and has given presentations at various technical conferences, including JSConf 2014, held in Bangalore, where he spoke about his experience with Ember.js.

Suchit was introduced to Ember.js during its pre-beta days and has been leading a team of developers to create applications for different clients, including one of the largest property evaluators in Australia.

He has been a contributor to books such as Talking With Tech Leads by Patrik Kua.

Suchit's blog can be found at http://suchitpuri.com/.
Ember.js Web Development with Ember CLI

*Ember.js Web Development with Ember CLI* is a must read for the next generation of web developers who will enjoy building powerful single-page web applications using the simplicity of Ember CLI and sophistication of the upcoming Ember 2.0. It is expected to be the lightest, thinnest, and most powerful version of Ember.js ever, with stability and backward compatibility important parts of its roadmap. This also means that many of the features of Ember.js 2.0 are already available today. This book will put you in a pole position to leverage them and stay ahead of the pack.

Unlike many other books that merely skim the surface, this book has a strong focus on Ember CLI, which will soon be the de-facto mode to build apps with Ember.js. It is a complete guide to creating powerful, scalable, and maintainable single-page web applications using Ember.js with Ember CLI.

**What This Book Covers**

*Chapter 1, Getting Started with Building Ambitious Ember.js Applications with Ember CLI*, will get you started with your first Ember.js application using Ember CLI and you will learn about its MVC pattern.

*Chapter 2, Understanding Ember.js Object-oriented Patterns*, covers the object-oriented design principles used in Ember.js, including classes and objects.

*Chapter 3, Rendering Using Templates*, shows you how to use the templating system of Ember.js and its default helpers.

*Chapter 4, Managing Application State Using Ember.js Routes*, explains how to manage the state of your application using Ember.Router.

*Chapter 5, Handling Display Logic Using Ember.js Controllers*, covers how to use controllers to communicate display logic to templates.

*Chapter 6, Communicating with the API Server Using ember-data*, shows you how to communicate with your backend server using ember-data models.

*Chapter 7, Building Reusable Components*, explains building reusable Ember.js components and learning how to customize them.
Managing Application State Using Ember.js Routes

In the last chapter, we learned about the Ember.js template layer. We learned how to use the Ember.js template markup, expression, and helpers to create complex templates. In this chapter, we will focus on the Ember.js routes. We will see how to manage your application state, using the Ember.js router.

In this chapter, we will cover the following topics:

- Application state
- Creating your first route
- Resources and nested templates
- Injecting the model for your template
- Making routes dynamic
- Setting up the controller
- Customizing templates to render
- The location API

Application state

URL, or Uniform Resource Locator, is one of the most important parts of any web application. With time, URLs have evolved from just referring a static resource on the server to identifying and managing the complete state of modern web applications.
State, as defined by FOLDOC (http://foldoc.org/state), is how something is—its configuration, attributes, condition, or information content. An application state is created when the user first requests the URL for the web application. As the user interacts with the application, the state of web application changes.

In Ember.js, the URL represents each of the possible states in your application. It can be thought as the serialization of application current state. As the user interacts with the application, the URL governs what the user is presented on screen. This means that there is a direct mapping between the URL and the state of the application. The router in Ember.js is responsible for maintaining this mapping.

When an Ember.js application first loads up, the router is responsible for setting up application state that corresponds to the current URL. The application state for an Ember.js application involves loading data for a specific route, setting up the model and controller, displaying the handlebars template corresponding to the route, and more.

Creating your first route

Till now most of our examples in our previous chapters have been using the index or / route. Real-world web applications seldom have only one route. Ambitious web applications, on the other hand, have a separate URL endpoint mapped to each different state of the application.

The Ember.js framework provides the concept of a router, a route, and a resource to manage the mapping between the URL and the state of the application.

Router in Ember.js is the core and the central part of the framework. It maintains the mapping of the URLs to individual routes. It monitors the URL of the web application and then, based on the mapping, it invokes the individual routes.

Let's see how to add this mapping. We will be adding two new routes, products and about, to our application.

The Router definition present at example1/app/router.js is as follows:

```javascript
import Ember from 'ember';
import config from './config/environment';

var Router = Ember.Router.extend({
  location: config.locationType // "auto"
});

Router.map(function() {
  this.route("products",{ path: "/products" });
});
```
As you can see, we first create the `Router` object by extending `Ember.Router`. The `location` property of the router governs how to build the URLs of the application. We will be talking about the location API later in this chapter.

We use the `map` method of `Ember.Router` to create individual routes. We use the `this.route` method of the router to map URLs with the routes. The `route` method takes in three arguments: `name`, `options`, and `callback`.

The `name` corresponds to the name of the route and helps in identifying this route from other parts of the application.

The `options` argument expects a JavaScript object with your desired options set. In our case, we pass in the `path` attribute in the option's argument object to map our route to a specific URL.

The last argument is the `callback`, which is used for nesting the routes.

As you can see in the preceding code, we defined two routes with names `products` and `about`.

```
this.route("about", { path: "/about" });
});

export default Router;
```

Let's now look at the templates for these corresponding routes. As discussed in the previous chapter, all the templates go inside the `app/templates/` folder.

```html
<h2>Welcome to Ember.js</h2>
<div>{{#link-to 'about'}}About{{/link-to}}</div>
<div>{{#link-to 'products'}}Products{{/link-to}}</div>
{{outlet}}
```

The application template is present at example1/app/templates/application.hbs

```html
<br>
<br>
<div>Products Template</div>
```

The products template is present at example1/app/templates/products.hbs
Managing Application State Using Ember.js Routes

You can see from what precedes that apart from the application template, we have defined two additional templates, products and about. These templates contain the data to be displayed for the products and about pages.

In application.hbs, we are also using a very useful handlebars helper expression, {{#link-to}}. The link-to helper expression helps us avoid hardcoding the URL address in our application. It fetches the URL pattern from the mappings in the router.

So {{#link-to 'products'}} Products {{/link-to}} would generate something like the following:

```html
<a id="ember258" class="ember-view" href="/about">About</a>.
```

You can see that the generated HTML link’s href points to the correct URL path. The link-to helper makes our application transparent to any changes in the URL pattern of the application.

Let us say that later in development cycle of your application, the SEO (search engine optimization) expert comes in and suggests that by changing the /about endpoint to /about-us, you will improve the search engine ranking for your site.

If you had used the {{link-to}} helper, you would just edit the router code to map to about-us instead of about, and your application would work just fine. Had you hardcoded the URL in your application, incorporating this change would have been error-prone and time consuming.

If you run the above code, you will see a screen that now has About and Products links on the homepage.

If you click on any of the links, you will see that the respective products or about template appears on the screen, as shown in the following figure:

```
Welcome to Ember.js

About
Products

Products Template
```

The products template rendered when a user clicks on the products link
In Ember.js, the routes of your application should inherit the framework’s Ember.Route class to provide any custom implementation of the route. One thing that you might have noticed in the preceding example would be that we did not create app/routes/products.js or app/routes/about.js in our application. Whenever a user visits the /products or /about URL, the Ember.js framework tries to find the corresponding routes based on naming conventions. For example, for /products, the router will try and instantiate app/routes/products.js for you, and if the framework is not able to find its definition, then it will generate the route for you. So, in our case, since we don’t define the app/routes/products.js or app/routes/about.js, the Ember.js framework generates them for us. Ember.js frameworks rely heavily on naming conventions, and routes are at the core of these conventions. Based on the name of the route supplied to the this.route() function, the framework tries to find the respective controller and template to use.

Let’s see this by an example; in the above example, we created the product route which uses the following:

```
this.route("products", { path: "/products" });
```

When the end user visits the /products path, the framework will try to find the route with the matching name and path, and will look for definition exported in app/routes/products.js, and when it finds one, it will instantiate the route for you and execute the hooks associated with the route (we will be covering the initialization of routes in more detail later in this chapter).

Then, the framework will try to find the controller that matches the route, which in our case should be defined in the app/controllers/products.js file. The framework will finally resolve the handlebars.js template it has to render, which should be defined in the app/templates/products.hbs file.

As you can see, this clearly forms a pattern. An example route declaration will map to the /example URL by default, and the framework will look for its matching route definition exported from the app/routes/example.js file, matching controller in app/controllers/example.js, and render the template defined in app/templates/example.hbs.

Now, since all of the application will follow this convention, it becomes really easy to find where the code for a specific functionality resides. It also makes debugging errors in your application very straightforward.
Resources and nested templates

Till now, we have seen very basic usage of the Ember.js routes; we have been using only top-level routes such as /products and /about, but very seldom do real-world applications have such simple routes.

Real-world applications will have resources or nouns, and actions that can be executed on these nouns that are depicted by verbs.

For example, there could be a /products/2 endpoint to show the product details page for a product with ID as 2, or /products/new and /products/2/edit routes to create and edit a product, respectively.

The Ember.js framework encourages using resource for all the nouns and route for all the verbs. This means that if you are creating your application for a specific domain area, all the entities of that domain should map to a resource and all the actions on the domain entities should translate to routes. A resource then becomes a collection of routes.

You can create a new resource using this.resource in the router. It expects two arguments, the first one is the name of the resource and other is a function that defines the nested routes, if any. If you don't have any nested routes in the resource, you can omit this argument. The following code snippet shows how to create a resource with nested routes:

```javascript
Router.map(function() {
    this.resource("products", function() {
        this.route("new");
    });
    this.route("about");
});
```

The nested route is defined in example2/app/router.js

Here, we have created a resource products and a nested route, new. To be able to display these routes, we will have to create the corresponding nested templates.

One very important thing to note here is that the nesting of resources/routes also means that their templates should also be nested in a similar fashion. Let's make it more clearer by defining our products, products/new and about templates:

```html
<br>
<br>
<div>Products Template</div>
{{outlet}}
```

The products template is present at example2/app/templates/products.hbs
Chapter 4

About Template

This Template should contain some information about us

The about template is present at example2/app/templates/about.hbs

Create a New Product

Product name:

{{input value=name}}

Product Description:

{{textarea value=description}}

Create

The products.new template is present at example2/app/templates/products/new.hbs

You can see above that we have defined two templates, one for products and the other for the new route that is nested under the products resource.

When you first look at the products template, you will notice that we have now added {{outlet}} at the end of the template. This outlet will enable nesting for this template, and all the nested routes that are present under the products resource will first render the products template present at app/templates/products.hbs, and then render the nested route's template in the outlet provided by the products template. If you remove the {{outlet}} from the products template, you will notice that the products.new template is never rendered, as it could not get the parent outlet to render the child route template.

You might also have noticed by now that we are referring to the new route that is nested inside the products resource by products.new. Ember.js follows this convention to refer to the nested routes in handlebars and other helpers, which is <<parent resource>>.<<nested route>>.

To link this nested route using the {{link-to}} handlebars helper, we would do something like the following:

{{#link-to 'products.new'}}Create a new product{{/link-to}}
Managing Application State Using Ember.js Routes

The following table shows the mapping of the different routes that you defined in your router to respective controller, route, and template files:

<table>
<thead>
<tr>
<th>URL</th>
<th>Route name</th>
<th>Controller</th>
<th>Route</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>index</td>
<td>app/controllers/index.js</td>
<td>app/routes/index.js</td>
<td>app/templates/index.hbs</td>
</tr>
<tr>
<td>N/A</td>
<td>products</td>
<td>app/controllers/products.js</td>
<td>app/routes/products.js</td>
<td>app/templates/products.hbs</td>
</tr>
<tr>
<td>/products</td>
<td>products.index</td>
<td>app/controllers/products.js</td>
<td>app/routes/products.js</td>
<td>app/templates/products.hbs</td>
</tr>
<tr>
<td>/products/new</td>
<td>products.new</td>
<td>app/controllers/products.js</td>
<td>app/routes/products.js</td>
<td>app/templates/products/new.hbs</td>
</tr>
<tr>
<td>/about</td>
<td>about</td>
<td>app/controllers/about.js</td>
<td>app/routes/about.js</td>
<td>app/templates/about.hbs</td>
</tr>
</tbody>
</table>

Let us see how the routes, which we have defined in our router above, map and initialize the controller, route, and template. Let us start from products.new route. Now, since the new route is defined inside the products resource, we would refer it by products.new. As we discussed earlier, route nesting also means controller and template nesting. This means that when the user visits the /products/new route, first of all, the products template is rendered from app/templates/products.hbs using the route exported from app/routes/products.js, and this route injects the controller from app/controllers/products.js to back the template.

After rendering the products template, the framework will look for the products.new template in app/template/products/new.hbs to render it in the {{outlet}} provided by the products template.
The controller exported at `app/controllers/products/new.js` and the route exported at `app/routes/products/new.js` will back the `app/templates/products/new.hbs` template.

The general rule of thumb is that first the parent resource is rendered, using its route and controller. Then, the nested child route is rendered in the parent template's outlet.

The nested or child route has its own controller and route, just like an independent route. The location of the nested route should be inside a folder whose name is the name of the resource it is nested in, for example, `app/routes/<<resource>>/<<nested route>>`.

It may seem a bit odd at first, but the trick here is to think of your application layout as nested templates rather than independent ones.

One thing that you might have noticed in the above table would be the `products` route, which is not mapped to any URL, and the `products.index` route, which we did not define anywhere. Let's revisit the `products` and `products.index` routes again here:

<table>
<thead>
<tr>
<th>URL</th>
<th>Route name</th>
<th>Controller</th>
<th>Route</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>products</td>
<td><code>app/controllers/products.js</code></td>
<td><code>app/routes/products.js</code></td>
<td><code>app/templates/products.hbs</code></td>
</tr>
<tr>
<td>/products</td>
<td>products.index</td>
<td><code>app/controllers/products.js</code></td>
<td><code>app/routes/products.js</code></td>
<td><code>app/templates/products.hbs</code></td>
</tr>
</tbody>
</table>

The `products` route is not mapped to any URL and is always invoked when a user visits `/products` or any of its child routes. It is very much like the application route present in `app/routes/application.js`, but only for all the routes that are nested inside the `products` resource.

If we want to handle errors or add in a common functionality for all the products, then its `app/routes/products.js` or `app/controllers/products.js` routes would be the right place to put in the common behavior.

For example, if we want to handle the errors that originate from the products pages in a specific way, we would put this specific behavior in the `products` route.
Similarly, whenever you create a nested route, you get `resource.index` that maps to `/resource` automatically. You just need to override the default implementation of `ResourceIndexRoute` and `ResourceIndexController` by defining them in your application with its custom behavior. This behavior is in line with the event bubbling topic we discussed in the `Action event bubbling` section of Chapter 3, *Rendering Using Templates*.

This hierarchy of controllers and routes keeps the entities focused toward providing functionality to one section, rather than putting everything in one place and later finding it difficult to maintain that. Another view of the template hierarchy of our application is shown in the following image:

![Template hierarchy of our application](image)

Now, you should be in a better position to understand the overall routes and template nesting in Ember.js.

### Injecting the model for your template

Till now, almost all of the examples we have seen so far have one thing in common: they all render static data that is returned from the `model` method of the corresponding route, something like the following:

```javascript
export default Ember.Route.extend({
  model: function() {
    return ['red', 'yellow', 'blue'];
  }
});
```

As we saw in Chapter 2, *Understanding Ember.js Object-oriented Patterns*, one of the very important advantages of using Ember.js framework is that it tries to build an application that uses components that are highly decoupled, yet internally cohesive.
The templates present the data that is fetched from the models. Routes play a very important role in this process. Routes help you to decide which model to fetch and how to customize it. As shown in the preceding snippet, the model can return a static list of colors or it can also fetch a list of colors from a remote server. All of this remains transparent to the templates that focus on displaying the list of item(s) returned from the model method.

Up until now, we have seen the model method in the route returning static data to the templates. But it will seldom be the case where your models return static data, and most of the time the data is fetched from the server and displayed to the user.

For the next example, we will be using GitHub's public API (found at https://developer.github.com/v3/) to fetch the commits in the Ember.js repository.

If you open the https://api.github.com/repos/emberjs/ember.js/commits link in a browser, you will get the commit data in JSON format for the Ember.js repository on GitHub, something like the following:

```json
[
  {
    "sha": "2da6e0b981ee20d2e2361102fcf7b8cb3ef812c5",
    "commit": {
      "author": {
        "name": "Stefan Penner",
        "email": "stefan.penner@gmail.com",
        "date": "2014-12-06T17:38:03Z"
      },
      "committer": {
        "name": "Stefan Penner",
        "email": "stefan.penner@gmail.com",
        "date": "2014-12-06T17:38:03Z"
      },
      "message": "Merge pull request #9826 from twokul/brocfile-dup-funct

Removes duplicate function",
    "tree": {
      "sha": "3eaae01753f4a2a919921232013ba32da65bab",
      "url": "https://api.github.com/repos/emberjs/ember.js/git/trees/3eaae01753f4a2a919921232013ba32da65bab"
    },
    "url": "https://api.github.com/repos/emberjs/ember.js/git/commits/2da6e0b981ee20d2e2361102fcf7b8cb3ef812c5",
    "comment_count": 0
  }
]
```
Now, let's consume this data and make it presentable for an end user.

We shall create two routes here: the `commits.index` route and the application `index` route.

As we don't have anything at present to show on the homepage of our application, we shall use the `redirect` method to transition from the `index` route to the `commits.index` route when anyone hits the `/` or the root URL. This is how the `commits.index` route will look:

```javascript
export default Ember.Route.extend({
  model: function() {
    var url = 'https://api.github.com/repos/emberjs/ember.js/commits';
    return Ember.$.getJSON(url);
  }
});
```

`Commits index route is present at example3/app/routes/commits/index.js`

The following code shows how we can redirect the index route of our application to a committed route:

```javascript
export default Ember.Route.extend({
  redirect: function() {
    this.transitionTo('commits.index');
  }
});
```

`Application index route is present at example3/app/routes/index.js`

As you can see in the preceding code snippet, we are using the jQuery `$.getJSON()` method to retrieve the data from the server. Now, instead of static text, our `model` function retrieves the commit data from the server and returns it to the template.

Now let's look at the two templates we will have for our application. One is for the application that will contain our application name. The application template can contain things that are common for the entire application. The other `commits.index` template can contain the code to display the list of commits of the repository. These commits can be retrieved from the model object set in the `commits.index` route, as shown in the following:

```html
<h1>Ember.js Repo</h2>
{{outlet}}
```

`Application template is present at example3/app/templates/application.hbs`
Typing asynchronously from the model

The commits.index template is present at example3/app/templates/commits/index.hbs

As you can see in the example3/app/templates/commits/index.hbs file, to show the list of commits, we just iterate over the model object and then just output the contents of each commit to the user. If you run the preceding code, you will see the output as shown in the following screenshot:

Ember.js Repo

<table>
<thead>
<tr>
<th>Commit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sha: 2da6f579f91ea202da296112062772b53e813c5</td>
<td>Merge pull request #9625 from twiku/brandaad-funct Removes duplicate function</td>
</tr>
<tr>
<td>Sha: 8f330e5077f701d954d8169650d7a44eb</td>
<td>Author: Alex Kovanadejyan</td>
</tr>
<tr>
<td>Sha: 48633a33d1cd4db2029156311467765eaa478f50</td>
<td>Author: Stefan Penner</td>
</tr>
<tr>
<td>Sha: cc0f4e7eac9d8cd7f05201289420a9d9f</td>
<td>Author: Stefan Penner</td>
</tr>
<tr>
<td>Sha: 9e20518a9a18e8e186babf10f908362b</td>
<td>Author: Stefan Penner</td>
</tr>
<tr>
<td>Sha: 97ee5577ba53b9708f1123f39e154f6d</td>
<td>Author: Stefan Penner</td>
</tr>
<tr>
<td>Sha: ab6f10d3cb4e206e0b96c922a149be80</td>
<td>Author: Stefan Penner</td>
</tr>
</tbody>
</table>

Using asynchronous data from the model

Making routes dynamic

Now we have successfully returned the data fetched from the server on the invocation of a route. Let's move ahead and see how can we make our routes dynamic. Till now, the URL that maps to our routes has been fixed. There may be situations in which we may want to read parts of URL and then act according to what was received. For example, we may want to make the /products/:product_id route, which shows the product page with ID equal to :product_id, or, like in our above example, we may want to create a new route /commits/:sha to display the information of a specific commit.
Managing Application State Using Ember.js Routes

The above dynamism in Ember.js routes is achieved through **dynamic segments**. Dynamic segments in routes start with `:`. Let’s see this by an example. We shall continue the above example and create a new screen to link to a template that shows information about a specific commit.

Let’s look at the JavaScript code. First, in the router, we need to define two routes—one for the complete list of the commits and one for individual detail of a specific commit:

```
Router.map(function() {
  this.resource("commits", function() {
    this.route("commit", {path: ":sha"});
  });
});
```

*The application router is present at example3/app/router.js*

This would result in a route hierarchy that is similar to the following figure:

![Route hierarchy of the GitHub commits application](image)

You can see that at the top level is the application route. We get the index route for free, but we are not using that route in our application, as we would focus on the `commits` resource here.

Now, since we created `commits` as a resource, we would get `commits.index` route automatically by the framework, which would map to the `/commits/` URL.

The commit route created inside the `commits` resource is meant to show the details of a particular commit. As you can see in the router code, we have made this route dynamic by including the `:sha` dynamic segment into it:

```
this.route("commit", {path: ":sha"});
```
Let us now look at the route's code:

```javascript
import Ember from "ember";

export default Ember.Route.extend({
  model: function(params)
  {
    console.log("model hook called");
    var url = 'https://api.github.com/repos/emberjs/ember.js/commits/' + params.sha;
    return Ember.$.getJSON(url);
  },
  serialize: function(model)
  {
    return {sha: model.sha};
  }
});
```

The commits.commit dynamic route to get the information for a specific commit is present at example3/app/routes/commits/commit.js

The commits.commit route fetches the details of a particular commit that is identified by a unique SHA and returns a model that contains the commit object.

This SHA makes our route dynamic and hence we would need to handle that in our routes. For that, we would need to get the dynamic segment from the params variable that is passed on to the model method.

```javascript
var url = 'https://api.github.com/repos/emberjs/ember.js/commits/' + params.sha;
```

You can see that we use params.sha to access the dynamic segment we defined in the route. Please note here that the name of the dynamic segment is the one that we defined in the router, such as the following:

```javascript
this.route("commit", {path:"sha"});
```

We just need to omit the : to get the variable name. Let's make it more clearer by another example.

If you defined a route with dynamic route as follows:

```javascript
this.route("product", {path:"/products/:category/:name"});
```
Here, we have defined two dynamic segments in the URL: name and category. We can access these dynamic segments in our route by using params.category and params.name variables, respectively.

**Route's serialize method**

Now, as we are using dynamic segment (:sha) in our commits.commit route whenever we programmatically call this route using the {{#link-to}} helper, we need a way to extract the URL params from the model object passed to the {{#link-to}} helper, so that the framework can construct the right URL for the application. The serialize method is used exactly that same purpose. This method takes in the model and returns the equivalent of the params object that can be used to generate the URLs of the given route. In our case, we would need to return an object that would have a sha property and its corresponding value.

In our template, you will see that we use the {{#link-to}} helper to link the SHA of a commit to its route:

```handlebars
{{#link-to 'commits.commit' c}}{{c.sha}}{{/link-to}}
```

When we use this link-to helper, it will use the App.CommitsCommit route's serialize method to build the dynamic URL.

If you don't return the required object from the serialize method, you would notice that the generated URL will be of the form /commits/undefined, which is not the expected behavior.

**Setting up the controller**

In all of the above examples, we assume that the model object that is returned from the model method of the corresponding route is automatically available in the templates. Here, we will look at how this automatic behavior is implemented in Ember.js, and if you want to change this behavior, how should you do it.

 Ember.Route defines a hook that can be used to alter this default behavior. This hook is defined in the setupController method. The default implementation of setupController sets the model property of the controller. This model property is fetched from the model method in the corresponding route definition:

```javascript
setupController: function(controller, model) {
    controller.set('model', model);
}
```

The default implementation of route's setupController function
So, if you want to set a different property than `model` on the controller to be accessible by the templates, you will have to change that behavior by providing your own implementation of the `setupController` method in your route.

There are times when you would want to add or set additional properties on the controller that maps to the route. All the additional properties changes should go in your route's `setupController` function:

```javascript
setupController: function(controller, model) {
    this._super(controller, model);
    controller.set("myProperty","myValue");
}
```

You can see that we set an additional property on the controller here using `controller.set`.

One thing you need to be sure of is that when you provide your own implementation of `setupController`, the default behavior will not be executed. To keep the default behavior of setting the `model` property on the controller, you should call the `this._super` method, which will execute the `Ember.RouterssetupController` function and will keep the default behavior intact.

**Customizing templates to render**

Up until now, we have seen that when you visit a URL `/example`, which maps to route `app/routes/example.js` and `app/routes/example/index.js`, this renders `app/templates/example.hbs` and `app/templates/example/index.hbs` templates. To change this behavior, Ember.js provides us with a hook that is used to resolve which templates to render for the current route.

One common use case for customizing which templates to render is when your application template has multiple outlets and you want to render a particular route on an outlet other than the default one.

Let's see this by an example. We will be using the `bootstrap` library in our project. For this, we need to first install the `bootstrap` dependency in our project by using the following:

```
bower install --save-dev bootstrap
```
So, in the chapter-4/example4 directory, we need to run the preceding command. This should fetch and install the bootstrap library in the example4/bower-components/ directory. Now we need to tell our asset pipeline broccoli to include the bootstrap assets (css, js, fonts, images) in our vendor.js and vendor.css files. To do this, we will have to make changes in example4/app/Brocfile.js, and we will have to add the following lines in the file:

```javascript
app.import('bower_components/bootstrap/dist/css/bootstrap.css');
app.import('bower_components/bootstrap/dist/css/ bootstrap.css.map', { destDir: 'assets' });
app.import('bower_components/bootstrap/dist/fonts/ glyphicons-halflings-regular.eot', { destDir: 'fonts' });
app.import('bower_components/bootstrap/dist/fonts/ glyphicons-halflings-regular.ttf', { destDir: 'fonts' });
app.import('bower_components/bootstrap/dist/fonts/ glyphicons-halflings-regular.svg', { destDir: 'fonts' });
app.import('bower_components/bootstrap/dist/fonts/ glyphicons-halflings-regular.woff', { destDir: 'fonts' });
app.import('bower_components/bootstrap/dist/js/bootstrap.js');
```

The bootstrap-specific files which we need to add to Brocfile.js

Now, we are good to go and we can now start using the bootstrap glyph icons, css, and js in our code.

We will consider a use case in which our application has a sidebar and a main body content. Both the sidebar and the main body should be customizable. This also means that our different routes can have different sidebars and body content.

The code for this example is present at https://github.com/suchitpuri/emberjs-essentials/tree/master/chapter-4/example4.

Since our application has a sidebar and a main body, let us first create the application template that accommodates these two components. Since these components are customizable, we will have to define two outlets to render route-specific content in these two components.

If you look at application.hbs, present at example4/app/templates/application.hbs, you will notice that we have defined two templates: one for the sidebar content and one for the body:

```html
<ul class="sidebar-nav" id="sidebar">
  {{outlet sidebar}}
</ul>
<div class="row">
  <div class="col-md-12">
    {{outlet}}
  </div>
</div>
```
Now, we need to tell our routes to render different templates in different outlets. Ember.js provides us with a hook to override which template to render in different outlets.

The `renderTemplate` method of `Ember.Route` provides this hook. The default behavior of this hook is to render the matching template in the `application` outlet of its parent route.

Now, as our requirement is different, we will have to override this method to provide our implementation of rendering the template matching this route.

Let's look at the `index` route of our application, as follows:

```javascript
import Ember from "ember";
export default Ember.Route.extend({
  renderTemplate: function(){
    this.render('sidebar', {
      outlet: "sidebar"
    });
    this.render('index')
  }
});
```

The `index` route is present at example4/app/routes/index.js

In the preceding code, we use `this.render` method to tell the router about the name and options of the template to render. The first argument is the name of the template to render. The options object can supply the following properties:

```javascript
{
  into: 'favoritePost',// the template to render into
  outlet: 'comment',// The name of the outlet in that template
  controller: 'blogPost'//The controller to use for that template
}
```

The `into` property specifies which template to render the current template in, the `outlet` property specifies which outlet to use in the above template, and the `controller` property specifies which controller to use when rendering the specified template.
Managing Application State Using Ember.js Routes

As you can see from the above index route code, when anyone visits the / URL, we inform the index route to render the sidebar template in the sidebar outlet and the index template in the default application template.

Similarly, we can create other additional routes that render different templates inside their respective outlets. For example, in our example4 application, we define another route named as about, which renders the about_sidebar template in the sidebar outlet, and the about template in the default unnamed outlet present in the about template.

```javascript
import Ember from "ember";
export default Ember.Route.extend({
  renderTemplate: function(){
    this.render('about_sidebar', {
      outlet: "sidebar"
    });
    this.render('about')
  }
});
```

The about route is present in example4/app/routes/about.js

When you run this application and open http://localhost:4200/ on your browser, you will see something like the following screenshot:

![The running application with sidebar](image)

Now, when you navigate to the http://localhost:4200/about, you will see that both the sidebar and the page content change, as shown in the following screenshot:

![Different sidebar and body content being rendered when you visit the about route in your application](image)
The location API

At the start of this chapter, when we were discussing about the router.js, or the application router that manages different URL states of the application, we passed in the location property to our Router object. The property was being read from config/environment.js file:

```javascript
var Router = Ember.Router.extend({
  location: config.locationType
});
```

The Router definition is present at example1/app/router.js

If you look into the config/environment.js file, you will see that the location property is set to auto for development and production environments and for test environments it is set to none.

The location API governs how to generate URLs for your application. The `location` property can be assigned one of these four values: hash, history, auto, and none.

If you set the location property of your router to hash, then the generated URLs of your application will have a # in them and will be of the form /#route. This type of location tracking depends on the hashchange event existing in the browser to detect any changes in the URL.

The history location type is more recent and is available in latest browsers (IE 10+, Firefox 31+, Chrome 31+) only. This will result in URLs that are similar to the normal URLs, that is, without the #. This technique uses the browser’s history API to keep track of URL changes. The resulting URLs will be of the form /route.

The auto-location type decides on which type to use, based on the browser. If the browser supports the history API, the application will use history as the location type, otherwise it will fall back to hash location type.

The last location type available is none. If you set your router's location type to none, then your router will not store the application URL state in the URL and the URL will remain constant throughout the application. This type of location is generally used for testing purposes.
Summary

In this chapter, you learned about navigating to different states of your application using Ember.Router. You saw how to create our own routes and resources. You learned how to use dynamic segments in your routes, thereby making your routes read properties from the URL. Then, you learned about how to create nested routes and the naming conventions associated with it. You looked at how easy it is to customize the route's behavior to set or change properties on the controller backing the template. Then, you learned about how you can render templates into different outlets of our application. Finally, you looked at what location API means and what the four different types of values that can be assigned to the location property of our router are.

In the next chapter, you will learn about how to effectively communicate with the API server using Ember Data.
Where to buy this book

You can buy Ember.js Web Development with Ember CLI from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.