brief contents

PART 1  EMBER.JS FUNDAMENTALS ............................................ 1

1  ▪  Powering your next ambitious web application  3
2  ▪  The Ember.js way  32
3  ▪  Putting everything together using Ember.js Router  48
4  ▪  Automatically updating templates with Handlebars.js  75

PART 2  BUILDING AMBITIOUS WEB APPS FOR THE REAL WORLD  97

5  ▪  Bringing home the bacon—interfacing with the server side using Ember Data  99
6  ▪  Interfacing with the server side without using Ember Data  123
7  ▪  Writing custom components  142
8  ▪  Testing your Ember.js application  161

PART 3  ADVANCED EMBER.JS TOPICS ............................................. 181

9  ▪  Authentication through a third-party system—Mozilla Persona  183
10  ▪  The Ember.js run loop—Backburner.js  197  
11  ▪  Packaging and deployment  211
Ember.js is a JavaScript MVC framework that helps you organize and structure the source code for large web applications. In comparison to other popular JavaScript application frameworks, it delivers a more complete MVC pattern, while also including features to help you build applications for the web of tomorrow. It’s also one of the more opinionated frameworks available, relying heavily on convention over configuration to help you structure your application.

Because of its large number of features coupled with the conventions that it expects from your application, Ember.js has a steep learning curve. Part 1 of this book, comprised of the first four chapters, eases you into the mindset of Ember.js application development, while ensuring that you build something useful right from the get-go.

The first two chapters focus on the core features that Ember.js brings to the table. Chapter 3 focuses on Ember Router, and chapter 4 focuses on the template library of choice for Ember.js developers: Handlebars.js.
This chapter introduces the Ember.js application framework and touches on many of the features and the technologies in the Ember.js ecosystem. Most of these topics are covered in more detail in later chapters. This chapter gives a quick overview of what an Ember.js application might look like and what strengths you get from basing your application on Ember.js.

The chapter also includes an overview of the building blocks of an Ember.js application and touches on the different aspects of the Ember.js framework. If you initially find any code presented here confusing or hard to understand, don’t worry! All aspects of the source code development are explored in detail, every step of the way, in the book.
Ember.js comes with a steep learning curve if you’re used to writing server-side generated web applications. The code examples presented in this chapter and the Notes application example go through the different concepts of structuring an Ember.js application.

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

If you are building a single-page web application that pushes the envelope of what’s possible on the web, Ember.js is the framework for you!

1.1 Who is Ember.js for?

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

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

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

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

Single-page applications (SPAs) have become more common because RIAs—and SPAs in particular—feel more like native, installable applications, and they have a more responsive user interface with few or no complete page refreshes. Within this domain, Ember.js aims to be a framework that provides the best solutions for web application developers and pushes the envelope of what’s possible to develop for the web. As such, Ember.js fits well with applications that require long-lived user sessions, have rich user interfaces, and are based on standard web technologies.

If you build applications toward the right of the spectrum, Ember.js is for you. Ember.js also makes you stop and think about how you want to structure your application. It provides powerful tools for building rich web-based applications that stretch the limits of what’s possible, while providing a rich set of features that enable you to build truly ambitious web applications.

Before you get started with developing an Ember.js application, let’s discuss why we have frameworks like Ember.js in the first place, as well as the problems that Ember.js promises to solve.
CHAPTER 1  Powering your next ambitious web application

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

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

Although many websites still rely on the full-page refresh approach shown at the left in figure 1.3, more and more developers are building dynamic content into their websites. Today, users expect websites to act and feel like applications with no page refreshes occurring.

1.2.1 The rise of asynchronous web applications

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

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

For example, when you browse an online store, you search for items to add to your shopping cart. When you add an item to the cart, you reasonably expect the item quantity and the shopping cart summary to update simultaneously. This lets you know the total number of items as well as the total price of the items in your shopping cart.
Because it’s difficult to define a set of general rules that define which elements the server will include in each of the Ajax responses, most server-side frameworks revert to sending the complete web page back to the client. The client, on the other hand, knows which elements to replace and swaps out the correct HTML elements.

As you can guess, this approach is inefficient, and it significantly increases the number of HTTP requests that the client sends to the server. This is where the power of Ember.js comes into play. As a developer, you probably understand the issues with the model presented in figure 1.3, in which the server side returns the updated markup for single elements on the page. To update multiple elements, you need to take one of the following approaches:

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

The first option multiplies the number of HTTP calls to your server; the second option requires you to maintain client state on both the client and the server. As a result, you significantly increase the number of HTTP requests that the client issues against the server, but you don’t decrease the amount of work that the server needs to do for each of these requests. Don’t get me wrong, this model supports partial-page updates by replacing elements based on the element identifiers and cherry-picking these elements from the complete markup returned from the server. If you’re thinking this is a waste of both server- and client-side resources, you’re absolutely right. Figure 1.4 shows this structure.

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

### 1.2.2 Moving toward the Ember.js model

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

In figure 1.5, the user receives the full website once, upon the initial request. This leads to two things: increased initial load time but significantly improved performance for each subsequent user action.
In fact, the model presented in figure 1.5 is similar to the traditional client/server model dating back to the 1970s, but with two important distinctions: the initial request serves as a highly viable and customizable distribution channel for the client application while also ensuring that all clients adhere to a common set of web standards (HTML, CSS, JavaScript, and others).

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

Now that you understand which types of web applications Ember.js is created to build, let’s delve into the details of Ember.js.

### 1.3 Overview of Ember.js

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

**What Is SproutCore?**

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

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

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

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

You must be eager to get started with Ember.js by now, but before you move on to create your first Ember.js application, let’s explore what Ember.js is and what parts make up an Ember.js application.

### 1.3.1 What is Ember.js?

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

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

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

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

### 1.3.2 The parts that make up an Ember.js application

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

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

---

CHAPTER 1  Powering your next ambitious web application

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

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

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

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

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

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

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

Figure 1.6  The parts that make up Ember.js and how they fit in with the MVC pattern
The model layer holds the data for the application, which is typically specified through a semi-strict schema. The model layer is responsible for any server-side communication as well as model-specific tasks such as data formatting. The view binds the GUI components against properties on the model objects via a controller.

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

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

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

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

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

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

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

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

Now that you understand the parts that make up an Ember.js application, it’s time to start writing your first app.

1.4 **Your first Ember.js application: Notes**

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

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

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

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

A rough design of what you’re building is shown in figure 1.7.
To get started, download the following libraries. The versions of each library may vary, depending on the current version of Ember.js:

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

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

To start from scratch:

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

   ![Directory Structure](notebook.png)

   - css
   - bootstrap.css
   - master.css
   - index.html
   - js
   - app
   - app.js
   - scripts
     - bootstrap-modal.js
     - bootstrap.js
     - ember-1.0.0.js
     - ember-data-beta-1.js
     - ember-data-localstorage.js
     - handlebars-1.0.0.js
     - jquery-1.10.2.min.js

To get the code from GitHub:

If you’d rather get the source code all packed up and ready to go, download or clone the Git source repository from GitHub: https://github.com/joachimhs/Ember.js-in-Action-Source/tree/master/chapter1. The GitHub repo is the finished result at the end of chapter 2.

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

### 1.4.1 Getting started with the Notes application

Wire the application files together inside your index.html file, as shown in the following listing.
The code in this listing is enough to get you started developing the Notes application.

Where to place your templates

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

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

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

### 1.4.2 Creating a namespace and a router

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

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

The first thing any Ember.js application needs is a namespace that the application lives inside. For your Notes application, you’ll use the namespace Notes.

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

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

```javascript
var Notes = Ember.Application.create({});

Creates a namespace for the application
```

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

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

- An application route
- An application controller
- An application view
- An application template
You don’t need to know what these four objects do at this point in time. What’s important to know is that you can override these default objects to include customized behavior.

To write some text onto the page, you’ll override the default application template with custom markup. Add a script tag inside your head tag of index.html. The type of this script tag needs to be "text/x-handlebars" and must include the name (id) of your template, as shown in the following example.

```html
<script type="text/x-handlebars" id="application">
  Hello Notes Application!
</script>
```

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

### Listing 1.2  Overriding the application template

| Create a Handlebars.js template named application. |
| Contents of template will be written to the screen. |
| Remember to close your script tag! |

**Running your application**

Although you can run the Notes application by dragging the index.html file into the browser, I recommend hosting the application in a proper web server. You can use the web server that you’re most comfortable with. If you want to start up a small lightweight web server to host the current directory you’re in, you can use either the asdf Ruby gem, or a simple Python script.

**If you have Ruby installed**

1. Install the asdf gem by typing `gem install asdf` into your terminal (Mac or Linux) or command prompt (Windows).
2. Once the gem is installed, host the current directory by executing `asdf -port 8080` in your terminal or command prompt.

**If you have Python installed**

1. Execute `python -m SimpleHTTPServer 8088` in your terminal or command prompt.

Now that you’ve got some text on the screen, you can move on to defining the setup for the rest of the Notes application. To do this, you’ll need to think about which states (routes) your application can be in.
But first delete the application template you added in listing 1.2. For the rest of the chapter, you won’t need to override the default application template, so go ahead and delete it.

### 1.4.3 Defining application routes

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

The Ember Router and how routes work are thoroughly explained in chapter 3. For now, add the following route definition to your app.js file as shown in the next listing.

**Listing 1.3  Defining the application’s router**

```javascript
Notes.Router.map(function () {
  this.resource('notes', {path: '/'}, function() {
    this.route('note', {path: '/note/:note_id'});
  });
});
```

This code creates a map of your application’s routes inside the Notes.Router class. Your router has two routes. One is named notes and belongs to the URL “/”; the other is named note and is a subroute of the notes route. A route that can have subroutes is referred to as a resource in Ember.js, whereas a leaf route is referred to as a route.

Both resources and routes derive their fully qualified names as combinations of their parent route names and their own names. For example, the note route in the listing is referred to as the notes.note route. This convention extends to controllers, views, and templates, too. Based on the router you defined, Ember.js creates the following default object implementations:

- Notes.NotesRoute
- Notes.NotesController
- Notes.NotesView
- notes template
- Notes.NotesNoteRoute
- Notes.NotesNoteController
- Notes.NotesNoteView
- notes/note template

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

**NOTE** Even though Ember.js creates default implementations of each of the files listed above, you only need to override the files you want to modify. As a result, your Notes application won’t have an implementation for all the classes listed.

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

**Listing 1.4 Defining the application’s routes**

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

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

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

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

Now, you’ll add some real content to your application.

### 1.4.4 Creating and listing notes

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

Because you already defined your routes, all you need to get started is to add a new template called `notes`. The following code shows the text field and the button added to `index.html`. 
You wrap the contents of your notes template in a div element with the id of notes to ensure that the correct CSS styling is applied to the list of notes. Inside this div element, you add the text field and the button. For now, they don’t have any functionality because you haven’t told Ember.js what to do with the text entered in the text field or what to do when the user clicks the button.

To get text onto your application, you created a custom implementation of the application template in listing 1.2. Because you no longer want to include this text in your Notes application, go ahead and delete the application template you created. Relying on Ember.js’s standard application template works just fine for the Notes application.

**NOTE**   Whenever Ember.js requests a template that you haven’t defined yourself, it uses a default implementation, which contains only a single {{outlet}} expression.

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

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

### Listing 1.6  Creating the NotesController

```javascript
Notes.NotesController = Ember.ArrayController.extend({
  newNoteName: null,
  actions: {
    createNewNote: function() {
      var content = this.get('content');
      var newNoteName = this.get('newNoteName');
      var unique = newNoteName && newNoteName.length > 1;
      content.forEach(function(note) {
        if (newNoteName === note.get('name')) {
          unique = false; return;
        }
      });
    }
  }
});```

```text
Defines the actions on controller
Extends Ember.ArrayController
Binds newNoteName property to text field
Defines createNewNote action
Ensures that note name is unique
```
If name is unique, creates note using Ember Data createRecord, persists note to browser’s local storage, and resets contents of text field

```javascript
if (unique) {
    var newNote = this.store.createRecord('note');
    newNote.set('id', newNoteName);
    newNote.set('name', newNoteName);
    newNote.save();
    this.set('newNoteName', null);
} else {
    alert('Note must have a unique name of at least 2 characters!');
}
}
```

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

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

The contents of the `createNewNote` action are straightforward:

- Verify that the name of the new note contains at least two characters.
- Ensure that no other notes exist in the system with the same name.
- Once the new note name has been verified, create a new note and persist it into the browser local storage.

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

### Listing 1.7 Initializing Ember Data

```javascript
Notes.Store = DS.Store.extend({
    adapter: DS.LSAdapter
});

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

Now that your application is set up to use the browser’s local storage through Ember Data, you can bind your text field value and button action to the `Notes.NotesController`. The following listing shows the updated `notes` template in `index.html`. 
Now that you can add new notes to the application, you also want the ability to list all notes in the Notes application. To implement this functionality, edit the notes template with the code shown in the next listing.

The additions to the code are straightforward, if a little unfamiliar at this point. You use the {{#each}} Handlebars.js expression to iterate over each of the notes inside the Notes.NotesController. For each of the notes, you print out the name. You use Twitter Bootstrap to style your user interface. Figure 1.9 shows the results of loading the updated index.html.
At this point, you might think that you went through quite an ordeal to get a list of notes onto the screen, but as you’ll soon discover, all that hard work will pay dividends.

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

1.4.5 Selecting and viewing a note
What’s a notes application without the ability to write text into the individual notes? By the end of this section, you’ll have implemented this part of the application.

NOTE The complete source code for this section is available as app2.js in either your source code directory or online via GitHub: https://github.com/joachimhs/Ember.js-in-Action-Source/blob/master/chapter1/notes/js/app/app2.js. You’ll find the complete source code for this section in the index2.html and app2.js files. This example uses Ember.js 1.0.0, and thus the `{{#linkTo}}` helper. In newer versions of Ember.js, this helper has been renamed to `{{#link-to}}`. If you are using a newer version than 1.0.0, Ember.js will tell you that the `{{#linkTo}}` helper has been deprecated.

**Listing 1.10** Linking each note to the `notes.note` route

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

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

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

In the case of your Notes application, you want to transition the user from the `NotesRoute` to the `NotesNoteRoute` whenever the user clicks the name of a note. In addition, you want to pass in the selected note to the `NotesNoteRoute`. 
Refresh your application and select a note by clicking it. Each note name in the list is now an HTML hyperlink. When you click a note, notice that the application URL updates to reflect which note you're currently viewing (see figure 1.10).

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

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

**Listing 1.11 Adding an outlet to the notes template**

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

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

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

List of all available notes
Input field for adding new notes to the application
URL is updated to reflect which note is selected

List of available notes
Input field for adding new notes
Link to view each note
Browser URL updates when a note is selected.
After telling the notes template where to render the notes.note route, you can add a template that shows the selected note. Create the new template inside your index.html file with the id notes/note, the contents of which are shown in the following listing.

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

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

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

Figure 1.11 shows the updated application.

Figure 1.11 The selected note’s content appears at the right side of the figure.
Before we move on to the deletion of notes, you’ll fix two issues:

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

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

### Listing 1.13  Highlighting the selected note

```
<script type="text/x-handlebars" id="notes">
  <div id="notes" class="azureBlueBackground azureBlueBorderThin">
    {{input valueBinding="newNoteName"}}
    <button class="btn btn-default btn-xs" {{action "createNewNote">New Note
  </button>
  <div class="list-group" style="margin-top: 10px;">
    {{#each controller}}
      {{#linkTo "notes.note" this class="list-group-item"}}
        {{name}}
      {{/linkTo}}
    {{/each}}
  </div>
</div>
</script>
```

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

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

### Listing 1.14  Adding a button to the `notes/note` template

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

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

---

*Your first Ember.js application: Notes*
Listing 1.15  Adding a NotesNote controller to update the note

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

Your application should now look like figure 1.12. Note that the selected note is highlighted on the left side of the figure, the URL is updated to reflect this, and an Update button now appears below the text area.

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

### 1.4.6 Deleting notes

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

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

![Figure 1.12](image-url)  

**Figure 1.12** The application now indicates which note is selected and saves updates made to the selected note.
To delete notes, add a Delete button to the selected note in the list at left. When the user clicks this button, the Notes application presents a modal panel asking for confirmation before the note is deleted. Once the user confirms that the note deserves to be deleted, the note is removed from the `Notes.NotesController`'s `content` property and the `selectedNote` property is reset to `null`. To implement this feature, add a modal panel to your application, which is available from the Twitter Bootstrap framework. You also need to add a couple of new actions to the `Notes.NotesController`.

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

### Listing 1.16  Adding a Delete button to the `notes` template

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

    <div class="list-group" style="margin-top: 10px;">
      {{#each controller}}
        {{#linkTo "notes.note" this class="list-group-item"}}
          {{name}}
        </a>
        <button class="btn btn-danger btn-xs pull-right" {{action "doDeleteNote" this}}>
          Delete
        </button>
      {{/linkTo}}
    </div>
  </div>
</script>
```

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

### Listing 1.17  Adding the `doDeleteNote` action to the `NotesController`

```javascript
Notes.NotesController = Ember.ArrayController.extend({
  needs: ['notesNote'],
  newNoteName: null,
  actions: {
    createNewNote: function () {
```

Adds button that fires `doDeleteNote` action on `NotesController`
The `doDeleteNote` action now takes a single parameter. Because you passed in the
note you want to delete in the third argument of the `{{action}}` expression, Ember.js
makes sure that this object is passed into your action. At this point, you don’t want to
delete the note without first making sure that this is what the user wants to do. Before
you display a confirmation message to the user, you need to temporarily store which
note the user wants to delete. Once that’s done, you show the user the modal panel,
which you’ll create next.

Because the HTML code to render the Twitter Bootstrap modal panel is slightly ver-
bose and you can potentially reuse it in multiple parts of your application, you’ll cre-
ate a new template that renders the modal panel onto the screen. Start by creating a
template called `confirmDialog` inside `index.html`, as shown in the following listing.

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

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

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

Listing 1.19 Rendering the confirmDialog template

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

  {{outlet}}

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

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

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

Listing 1.20 Implementing the doCancelDelete and doConfirmDelete actions

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

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

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

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

    doConfirmDelete: function () {
      var selectedNote = this.get('noteForDeletion');
      this.set('noteForDeletion', null);
      if (selectedNote) {
        this.store.deleteRecord(selectedNote);
        selectedNote.save();
      }
    }
  },
});
```

If user has a note to delete, deletes note and persists changes into local storage

Retrieves note for deletion based on noteForDeletion property

Hides modal panel

Calls for access to Notes.NotesNoteController

Implements doCancelDelete

Implements doConfirmDelete

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

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

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

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

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

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

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

That completes the functionality of the Notes application for this chapter. You’ll continue working with the Notes application while delving deeper into the Ember.js core concepts in chapter 2.
This chapter provided an overview of the building blocks that Ember.js is based on and introduced the most important concepts of an Ember.js application. I hope that you gained a better understanding of the Ember.js framework, as well as why it exists and where it’s most applicable to you as a web developer.

In this introductory chapter to Ember.js, I guided you through the development process of a simple web application, touching on the important aspects of the framework along the way. The goal of the Notes application was to show you as many of the basic Ember.js features as possible without complicating the application’s source code.

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

In the next chapter, you’ll reuse and extend the code you wrote in this chapter to thoroughly understand the core features that Ember.js provides.
Ember.js is a JavaScript MVC framework that handles important tasks like managing code modules, maintaining state, and expediting reliable data flow. It provides the patterns, components, and scaffolding you need to build web applications.

**Ember.js in Action** introduces the Ember.js framework and shows you how to build full-featured, desktop-quality web applications. You’ll begin with the basic architecture: client- and server-side MVC and how to integrate Ember.js with your favorite back end. Then you’ll explore the amazing Handlebars templating engine that automatically updates your apps when the data behind them changes. Along the way, you’ll develop a complete Ember.js application and learn how to deploy, administer, and update it efficiently.

**What’s Inside**
- Working with Ember Data
- Mastering Handlebars templates
- Advanced JavaScript techniques
- Covers Ember.js 1.0

Readers of this book need to know JavaScript. No prior experience with Ember.js is required.

**Joachim Haagen Skeie** is an experienced web application developer and the author of Montric, an open source monitoring tool built using Ember.js.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/Ember.jsinAction