Ember.js IN {{ACTION}}

Joachim Haagen Skeie

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Ember.js in Action
version 3

brief contents

PART 1. THE EMBER.JS FUNDAMENTALS

1 Ember.js - powering your next ambitious web application

2 The Ember.js Way

3 Putting it all together using Ember.js Router

4 Automatically updated templates with Handlebars.js

PART 2. BUILDING AMBITIOUS WEB APPS FOR THE REAL WORLD

5 Bringing home the bacon - Interfacing with the Server-side using Ember.js Data

6 Building upon the strengths of URLs

7 Building complex custom components

8 Testing your Ember.js application

PART 3. GETTING READY FOR DEPLOYMENT

9 Packaging and Deployment

10 Interacting with and deploying to the Cloud

1 Ember.js – Powering your next ambitious web application

This chapter covers:

- A brief history about why we have single-page web applications
- An introduction to Ember.js
- What Ember.js will be able to provide to you, as a web developer
- Your very first Ember.js application

This chapter will introduce Ember.js as an application framework, and will touch upon many of the features and the technologies that are part of the Ember.js ecosystem. Most of these topics will be covered in detail in chapters throughout the book, but this chapter will give you a quick overview of what an Ember.js application might look like and what strengths you should be able to get from basing your application around Ember.js.

This chapter serves as an overview of the basic building blocks of an Ember.js application, and will only briefly touch upon the different aspects that form the Ember.js framework. Do not worry if you find any code presented here confusing, or otherwise hard to understand Ember.js do come with a steep learning curve, especially if you are used to writing Web applications that are generated dynamically on the server-side. The code example presented in this chapter, the Notes application, do go through many different concepts of how an Ember.js application might be structured, and each aspect of the development of the source code will be explored in detail throughout the chapters of the book.

1.1 Who is Ember.js for?

Depending on what you want to build, Ember.js might or might not be the framework you are looking for. On one side of the scale you have traditional websites that serves its contents based on the traditional request-response life cycle, such as the websites for New

York Times or Apple.com. On the other end of the scale you have rich internet web applications that aim to either define new application types or otherwise compete with natively installed applications, like Google Maps, Trello and GitHub.

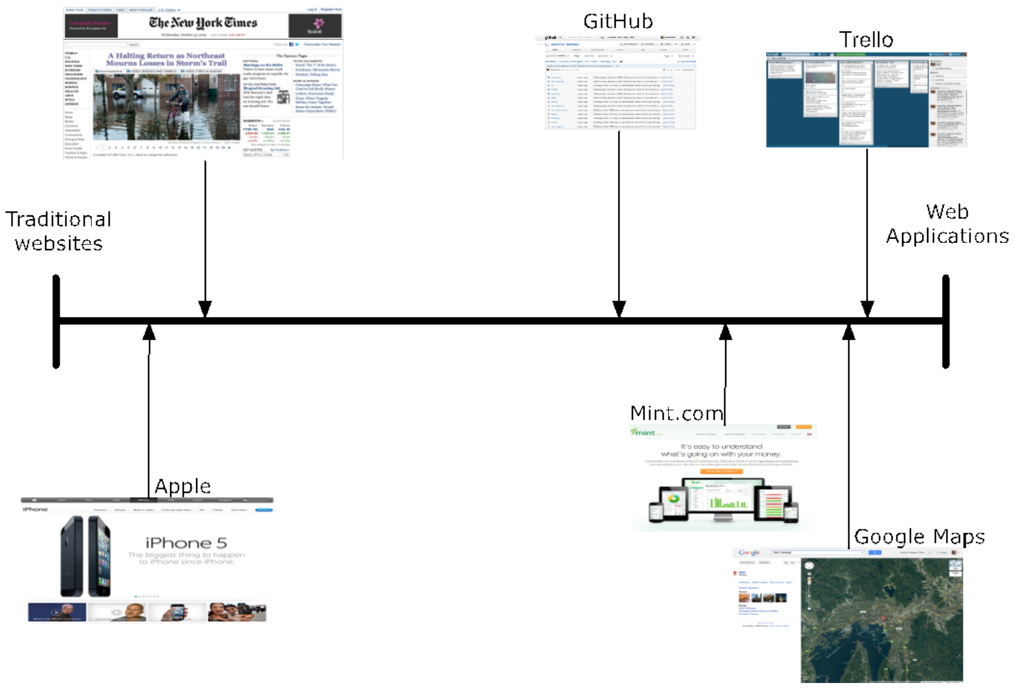


Figure 1.1 The range of web applications

There are strengths and weaknesses on both end of the scale. While the pages near the left of the scale will be easier to cache on the server-side they also tend to be a lot more reliant upon the request-response cycle and full page refresh between user actions. The applications near the right of the scale are all a lot more complex, requires more from the browser in terms of computing power, features and stability, but they also tend to have a much richer user interface and they tend to be applications that their users interact with in a similar manner as with native applications. It is within this domain that Ember.js aims to provide the best solutions to web application developers.

Ember.js is aiming to be a framework that can be used to push the envelope on what's possible to develop for the web, and as such Ember.js fits well with applications that have long lived user sessions and have a rich user interface, while being based on the standard web technologies.

If you are building applications that fit in with the applications to the right of figure 1.1, then Ember.js is definitely for you. On top of that Ember.js makes you sit down and think about how you want to structure your application, and it provides powerful tools with which you can build rich web based applications that push the envelope on what is possible on the web today while providing you with a rich set of features that will enable you to build truly ambitious web applications.

1.2 From static pages to AJAX to Full Featured Web apps

From the time the WWW was introduces up until Ajax became a thing in the early 1990s, each and every website request was static in nature, meaning that the server would respond to any HTTP request with a single HTTP response containing the complete HTML, CSS and JavaScript required to display the complete website, as depicted to the left in figure 1.2 below.

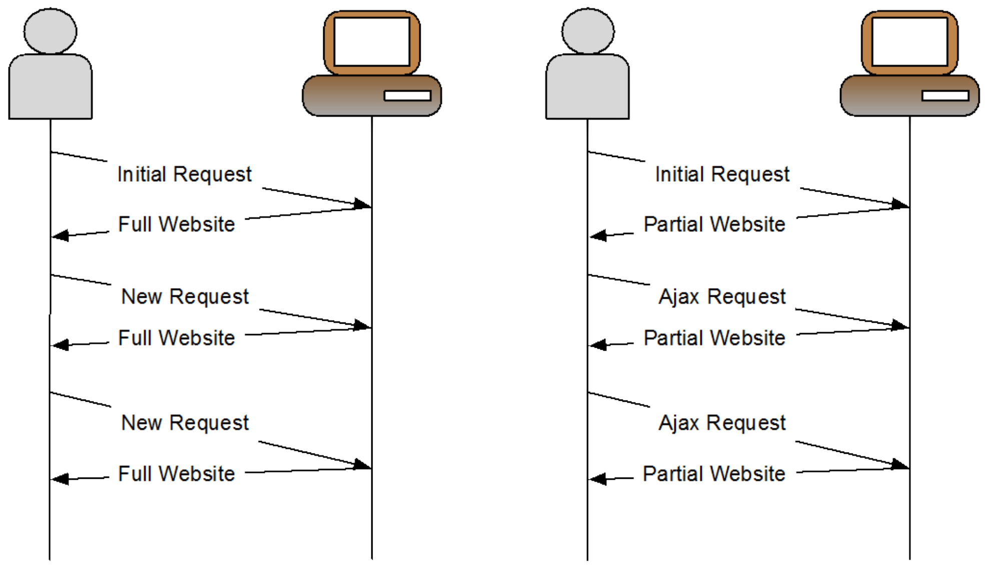


Figure 1.2 – The structure of the early Web vs the promise of AJAX

1.2.1 The rise of Asynchronous

As asynchronous calls was introduced, along with it came the possibility to only send parts of the website for each response. Dedicated JavaScript code would receive this response on the client and would replace the contents of HTML elements within the website, as shown to the right in figure 1.1 above. As nice as this seems, there is a gigantic caveat with this approach. It is trivial to implement a service on the server side that would, given an element type, render the new contents of that element in order to return that back to the browser in an atomic manner. If that only represents what the users of these rich web application wanted, that would have solved the problem. The issue, of course, is that users rarely only wants a single element to be updated at any one time. As an example, if you are browsing and online store, you might have navigated around or searched for items that you wish to add to your shopping cart. As a user of this online store you would reasonably expect that when you add an item to the cart, that both the item you are looking at, as well as the shopping cart summary will be updated to show you the total number of items in your cart, as well as the total amount you have accumulated in your shopping cart thus far.

1.2.2 Moving towards the Ember.js model

As a developer, you can most likely understand that at this point, your are running into some issues with the model presented above where the server side would return the updated markup for single elements on the page. You either have to make the browser fire off additional AJAX requests, one for each element that will be updated on the website, or you have to somehow know on both the client side and on the server side which elements that would end up being updated for each and every action that your users perform in your application, at which point you have to choose between multiplying the number of HTTP calls to your server, or choose to keep client state on *both* the client and on the server.

It turns out, though, that a third option was chosen as the basis of many of the most popular server-side frameworks, shown below in figure 1.3.

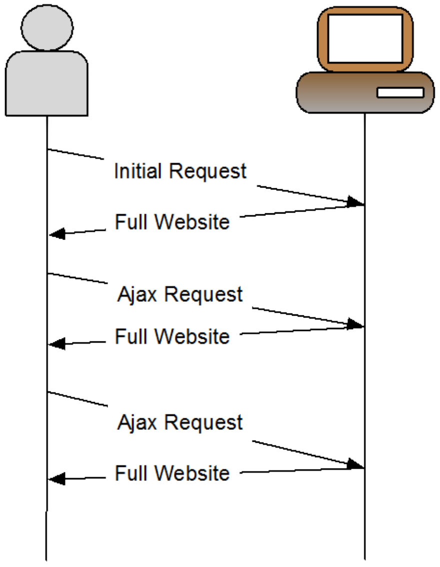


Figure 1.3 – The structure of a server-side framework

Now, you might notice the striking mix between the left and right diagrams of figure 1.1 here. It turns out that this approach was the chosen approach of many of the popular server-side web frameworks. Because there was no trivial way to store the relationships between the elements being changed by the user and the elements requiring updates from the server, many frameworks opted out of this completely. Don't get me wrong; they did support partial page updates, by way of replacing elements based on their element identifier and cherry picking these elements out of the complete markup returned from the server.

You might think this is a gigantic waste of both server-side and client-side resources, and you will be absolutely right. These days, websites rely a whole lot less on passing markup between the server and the client, and a whole lot more on passing data, and it is in this realm that Ember.js comes into play. This process is shown below in figure 1.3.

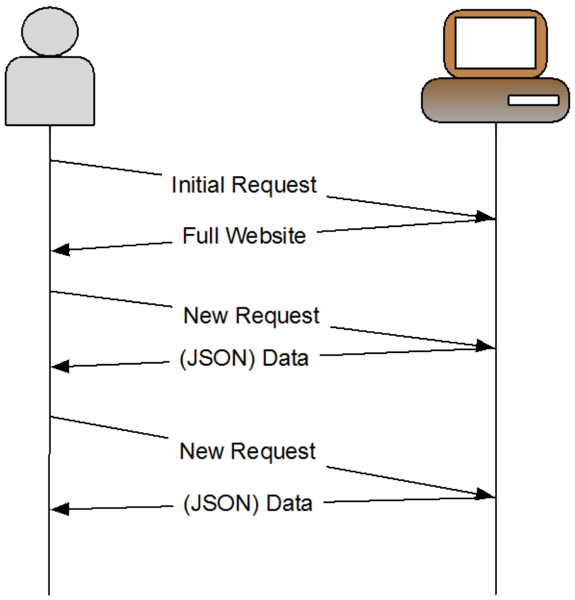


Figure 1.4 – A modern Web Application Approach

Looking at figure 1.4, you might notice that the user only received the full website exactly once, upon the initial request. This leads to two things, increased initial load time and significantly improved performance for each subsequent user action.

In fact, the model presented above, is similar to the traditional client-server model, dating back from the 1970s with two important distinctions; the initial request serves as a highly viable and highly customizable distribution channel for the client application while it also ensures that all clients in this model adhere to a common set of Web standards (HTML, CSS, JavaScript, etc).

Along with the client-server model, though, you will also note that the business logic involving user interaction, the graphical user interface, as well as performance logic, has shifted off the server and onto the client. This might be a security issue for some specialized deployments, but generally as long as the server has control of who has access to the data being requested, the security concerns can be delegated to the server, where it belongs. It also means that the concerns of the client and the server can get back to doing what they do best, serving the user interface and serving the data respectably.

1.3 Overview of Ember.js

Ember.js started its life as the second version of the SproutCore framework. If you are not familiar with SproutCore, SproutCore is a framework developed with a highly component-oriented programming model. SproutCore borrowed most of its concepts from Apple's Cocoa, as much so that Apple has written some of their web applications (Mobile Me and iCloud) on top of SproutCore. Apple has also contributed a large chunk of code back to the SproutCore project. On Tuesday, November 8th 2011, Facebook acquired the team responsible for maintaining SproutCore.

But while working on version 2.0 of SproutCore, it became clear that there was a need for a radical change in the underlying structure of the framework in order to be able to build a framework that was easy to use, applicable to a wide range of target web applications, while still being a rather small framework. In the end, part of the SproutCore core team decided that it would be better to split these changes into a framework separate from its SproutCore origins.

That being said, Ember.js does lend quite a lot 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 away most of the implementation details to their users, Ember.js does what it can to make it clear that HTML and CSS lies at the core of its development model. To me, Ember.js' strengths lie in the fact that it enables developers to structure their JavaScript source code in a consistent and reliable pattern, while it keeps the HTML and CSS easily visible to the developer. Not having to rely on specific build tools to be able to develop, build and assemble your application also means that you, as a developer, have a lot more options when it comes to how you wish to structure your development and which tools you wish to rely on when the time comes to make a decision on assembly and packaging. This book will go through a few of the packaging options that are available in part 3, chapter 11.

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. Ambitious can have a different meaning to different people, but as a general rule, Ember.js aims to be able to help web application developers to push the envelope on what's possible to develop for the web while keeping your application source code 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. Ember.js supports bindings - a mechanism where changes to one variable will propagate its value into other variables and vice versa, computed properties - a mechanism where its possible to mark functions as properties that will automatically update along with the properties they are reliant upon and automatically updated templates – a mechanism that will ensure that your graphical user interface (GUI) will stay up-to-date whenever changes happen to the underlying data. Combine this with a very strong and well thought-out Model-View-Controller architecture and you have got yourself a framework that will be able to deliver on its promise.

1.3.2 The Parts That Make Up an Ember.js Application

If you have spent most of your time developing web applications with server-side generated markup and JavaScript Ember.js, an indeed most of the new JavaScript frameworks, will have a completely different structure than you are used to. Ember.js includes a very rich Model-View-Controller (MVC) model that enrich all of the parts of a standard MVC model. When you build up an Ember.js application you will be separating the concerns of your applications, and you will spend a decent amount of time sitting down, thinking of where you would best place your application logic. Most likely you will opt to follow Ember.js' guidelines, but in some cases you might need to spend some time off the beaten track in order to implement the features of your application just right.

You have probably guessed that Ember.js has Views, Controllers and Models, but lets have a closer look at the core components that you will utilize when building an Ember.js application. As you can see from figure 1.5 below, Ember.js brings in a couple on extra concepts at each of the layers in the standard MVC model.

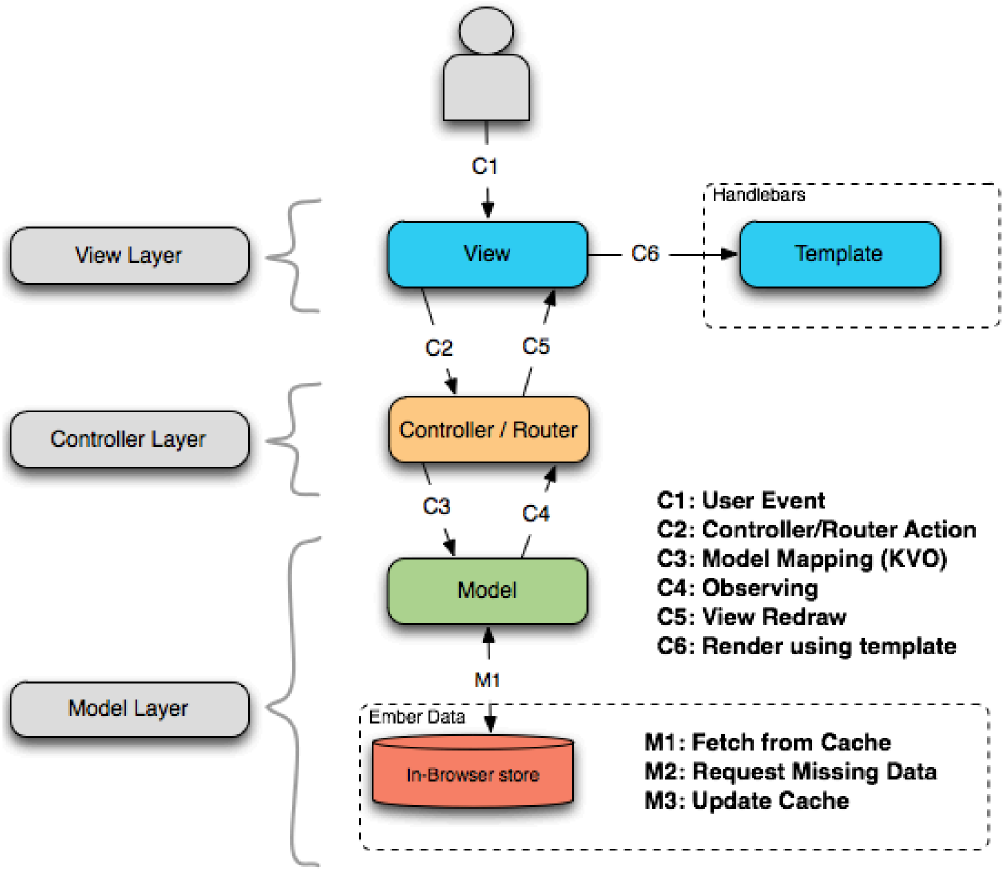


Figure 1.5 – The Parts that make up Ember.js and how they fit in with the MVC model

MODELS AND EMBER DATA

At the bottom of the stack Ember.js uses Ember Data in order to simplify and provide the rest of the application with the rich data model features that you will need in order to build truly rich web based applications. The model layer holds the data for the application. The data objects are generally specified clearly through a semi-strict schema. There is usually very little functionality within the models, and as we will see throughout this book the model object is generally responsible for tasks such as data formatting. The view will bind the GUI components against properties on the model objects, via a controller.

Ember Data, obviously, lives in the model layer and you will use it to both define your model object, your client-to-server side API as well as the transport protocol between your

Ember.js application and the server (jQuery, XHR, WebSockets, etc). You can read more about Ember Data in chapter 5.

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 would use the *ObjectController* if your controller is representing a single object, (like a selected blog post), and the *ArrayController* if your controller represents an array of items (like a list of all blog posts).

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 sub routes and you can use the Router to navigate between states within your application. Ember Router is also the mechanism that Ember.js uses in order to update your applications URL and to listen for URL changes. Using Ember Router you will model all of your applications states into a hierarchical structure that resembles a statechart. Ember Router will be discussed in detail in chapter 3.

VIEWS AND HANDLEBARS.JS

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

By default, each view in Ember.js will have exactly one controller as its context. It will use this controller to fetch its data, and it will by default use this controller as the target for any user actions that occurs on the view.

Also by default, Ember.js uses Handlebars.js as its templating engine. Therefore, most ember.js applications will define its user interface via handlebars.js templates. Each view will utilize exactly one template, which it will use to render its view. Handlebars.js and templates will be discussed in chapter 4.

Ember ships with default views for each of the standard HTML5 elements, and it is generally good practice to use these when you are in need of simple elements. For more complex elements in your web application you can easily create your own custom views that either extend or combine the standard Ember views.

1.4 Your Very First Ember.js Application - Notes

The source code for the Notes application weighs in at just about 200 lines of code and 130 lines of CSS, including the templates and JavaScript source code.

To get under the skin of what to expect from an Ember.js application, lets dive in and write a simple web application that manages notes. The application we will be writing is a simple Notes application where the user will be able to:

- Add new notes
- Select between notes in the system
- Edit existing notes
- Delete existing notes

The application will have an area where the user can add notes to the system. Available notes are going to be presented along the left hand side in a list. Once the user selects a note the user is able to view and edit the contents of that note. If the user chooses, each note can also be deleted. A rough design of what we are building is shown in figure 1.6 below.

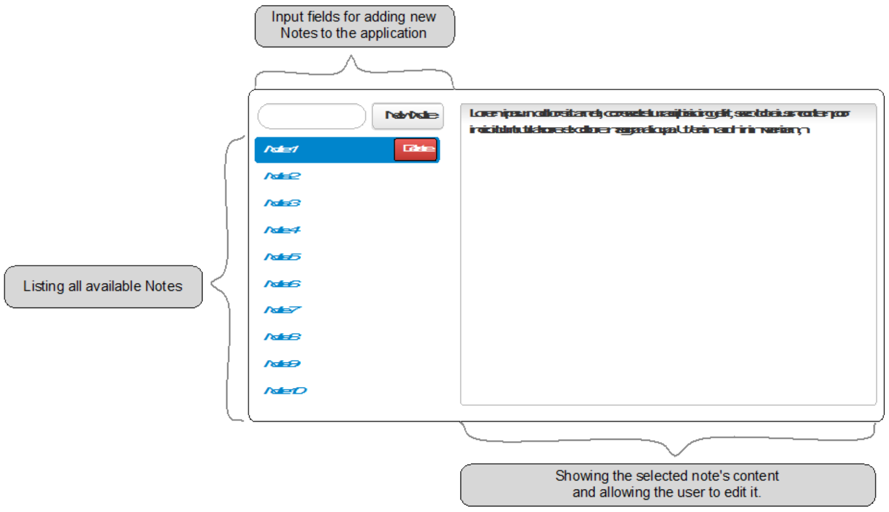

Input fields for adding new Notes to the application
Listing all available Notes
Showing the selected note's content and allowing the user to edit it.

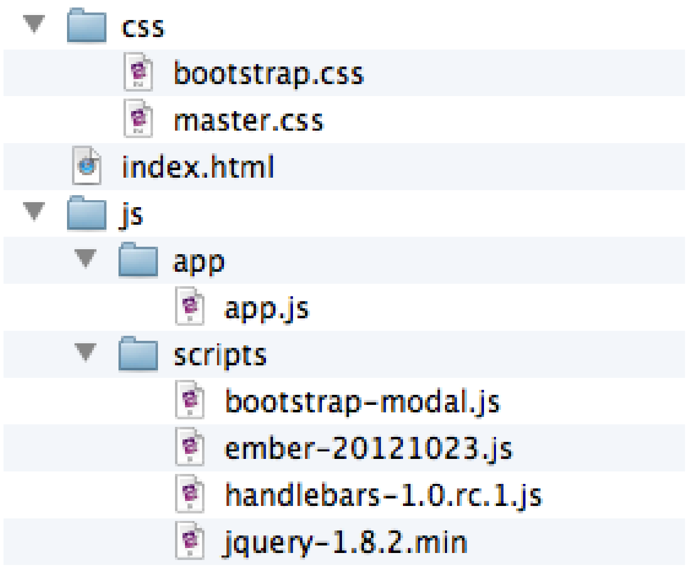
Figure 1.6 – The Basic Design of Our Notes Application

In order to get started, you are going to need download the following Libraries:

- Ember.js
- Handlebars
- jQuery
- Twitter Bootstrap CSS
- Twitter Bootstrap Modal

START FROM SCRATCH

Create a directory on your hard drive that will contain the files that will make up your application. Inside this folder, create the following directory structure:



```
css
    bootstrap.css
    master.css
index.html
js
    app
        app.js
    scripts
        bootstrap-modal.js
        ember-20121023.js
        handlebars-1.0.rc.1.js
        jquery-1.8.2.min
```

Figure 1.7 – The Notes Application Structure

GET THE CODE FROM GITHUB

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

GETTING STARTED WITH THE NOTES APPLICATION

We will wire these files together inside our index.html file, as shown in listing 1.1 below.

Listing 1.1 – The index.html file

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"                        #A
   "http://www.w3.org/TR/html4/strict.dtd">

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

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

    <script src="js/scripts/jquery-1.8.2.min.js" type="text/javascript"
charset="utf-8"></script>
    <script src="js/scripts/bootstrap-modal.js" type="text/javascript"
charset="utf-8"></script>
    <script src="js/scripts/handlebars-1.0.rc.2.js" type="text/javascript"
charset="utf-8"></script>
    <script src="js/scripts/ember.js" type="text/javascript" charset="utf-
8"></script>
    <script src="js/app/app.js" type="text/javascript" charset="utf-
8"></script>                                                            #E

</head>
<body bgcolor="#ffffff">

</body>
</html>
```

#A: The standard doctype declaration
#B: Starting the document with HTML and HEAD HTML Element
#C: Linking in the Twitter Bootstrap CSS file
#D: A Custom CSS file containing the customized CSS for this application
#E: Adding in the source code for the Notes Application

The code above is all of the code that this application will ever have inside the index.html file. This might be very different from the web development that you are used to, it certainly was for me before I was introduced to SproutCore a few years back. Unless you specify anything else, by default Ember.js applications will place its contents inside the body tag of your HTML document.

There is nothing-special going on in the code above. The document starts out by defining the doctype before starting the HTML element with the standard HEAD element. Inside of the HEAD element we are setting the page's title, along with links to both the Twitter Bootstrap CSS, as well as a separate CSS file where we will put the custom CSS required by our Notes application. The last four elements of the HEAD tag defines links to the four scripts that our application is dependent on, while the last script tag links in the source code for the Notes application we are developing throughout the rest of this chapter.

1.4.1 Getting your application started

In this section we will build the first part of the Notes application with the basic web application layout in place. The complete source code for this section is available as "app1.js" in the code source, or online at GitHub https://github.com/joachimhs/Ember.js-in-Action-Source/blob/master/chapter1/notes/js/app/app1.js.

The very first thing any Ember.js Application needs is a namespace that the application can live inside. For our Notes application this namespace will simply be `Notes`. After our namespace is created, we need to create a Router that knows how our application is structured. Using the Router is strictly not a requirement, but as you will see throughout this book it greatly simplifies and manages the structure of your entire application. You can think of the Router as the glue that holds your application in place and that connects different parts of your application together. Listing 1.2 shows the minimum amount of code required to get our Notes application up and running in order to serve a blank website.

Listing 1.2 Getting started with the Notes Application

```
var Notes = Ember.Application.create();                                     #A

Notes.Router = Ember.Router.extend();                                       #B

Notes.Router.map(function () {                                              #C
    this.route('notes', {path: "/"});

});

Notes.NotesRoute = Ember.Route.extend({                                     #D
    setupController: function(controller) {
        controller.set('content', []);                                      #E
        var selectedNoteController = this.controllerFor('selectedNote');
        selectedNoteController.set('notesController', controller);      #F
    }
});
```

#A: Creating a namespace for the Notes Application
#B: Defines our application's Router
#C: Defining the Routers single route, notes responding to the URL "/"
#D: Defining the NotesRoute to override setupController and renderTemplate functions
#E: Initializing the NotesControllers content
#F: Connecting SelectedNoteController to NotesController

The code above starte out creating our Notes namespace on the very first line via `Ember.Application.create()`. Any code that we write that are related to this application will be contained within this namespace, in order to keep the code related to our application completely separate from any other code that we might bring in via thrid party libraries or even inline in our JavaScript file.

The rest of the code define our applications Router. Going through the code for this section briefly, it starts out by creating a new *`Notes.Router`* class. This router has exactly one route named *notes* that will belong to the URL *"/"*. Each subsequent route will bind itself to a relative URL path for two way access, meaning that it will respond as expected to URL changes, while at the same time updating the URL when you transition between states programatically. Our Notes application has exactly one state at this time, called `notes`, that are bound to the relative URL `"/"`.

In The next part we are defining how `NotesState` will connect with the rest of our application, which is where the magic happens. As you can see here, we are connecting two controllers while also rendering two views to the screen. Ember Router has a strich naming convention, meaning that, for the `notes` route, it will automatically create or connect the route with classes names `Notes.NotesController`, `Notes.NotesView` and a template named `notes`. We only need to override these when we have requirements that go outside of the automatically generated classes. Ember Router also expects to find both a class named `ApplicationController`, which will serve as the controller for the router itself and a class named `ApplicationView`, which will server as the view where the outlets will be connected. Again, these will be generated automatically with default content if you are not providing them yourself.

Inside `renderTemplate`, we are rendering two views, `notes` and `selectedNote` into the outlets `notes` and `selectedNotes` respectably. Ember Router expects to find these outlets in the `application` template, which we get to shortly. We are also specifying that we would like the `selectedNote` view to have a different controller than the default, namely the `Notes.SelectedNoteController`.

In order to get a better sense of what is going on here, lets have a look at the code for the controllers, as shown in listing 1.3.

Listing 1.3 The Notes Applications Controllers

```
/** Controllers **/
Notes.ApplicationController = Ember.Controller.extend({});             #A

Notes.NotesController = Ember.ArrayController.extend({                 #B
    content: []
});

Notes.SelectedNoteController = Ember.ObjectController.extend({
    contentBinding: 'notesController.selectedNote',                    #C
    notesController: null
});
```

#A: Declaration of the ApplicationController
#B: Declaration of the NotesController, keeping track of all notes
#C: Declarationof the SelectedNotesController, keeping track of the selected note.

Our `ApplicationController` is declared via `Ember.Controller.extend({}),` which is a special controller that you will most likely only use directly once as defined here. Most of your other controllers will either be of type `ObjectController`, for controllers that serve a single object, or `ArrayController`, for controllers that serve a list of objects. If your application is complex enough it might be necessary to extend one of these controllers in order to build in your required business logic.

As you can see, we are declaring `Notes.NotesController` to be an `ArrayController`, while Notes.`SelectedNoteController is an ObjectController`, just as we might expect. Both of our controllers define a single property, `content`, one being an empty list upon completion while the other is bound to the property `notesController.selectedNote`. You can probably remember from above that we connected our two controllers together. Ember Router will ensure that once the application is initialized, that the `notesController` property is linked to the correct instance of the `Notes.NotesController`. As a result we are now able to bind our `Notes.SelectedNoteController's` content property directly to the property `selectedNote` of the `Notes.NotesController`. In effect this means that the content of `Notes.SelectedNoteController` will be automatically updated each time that the `selectedNote` property of the `Notes.NotesController` change. Bindings are one of the fundamental features of Ember.js and will be explored in detail in chapter 2.

The only thing missing from our application at this point is the views and the templates, shown below in listing 1.4.

Listing 1.4 The Notes Applications Views

```
//** Views **/
Notes.ApplicationView = Ember.View.extend({                      #A
    templateName: 'applicationTemplate'                          #B
});

Notes.NotesView = Ember.View.extend({                            #C
    elementId: 'notes',                                          #D
    classNames: ['azureBlueBackground', 'azureBlueBorderThin']   #E
});

Notes.SelectedNoteView = Ember.View.extend({                     #F
    elementId: 'selectedNote',
});
```

#A: The ApplicationView responsible for defining the outlets
#B: The name of the template that defines the ApplicationView
#C: The View for the list of notes
#D: The HTML element identifier for the NotesView
#E: The CSS class names to attach to this HTML element
#F: The View for the selected note

As mentioned above, our application at this point have exactly three views, `Notes.ApplicationView`, `Notes.NotesView` and `Notes.SelectedNoteView`. The ApplicationView is responsible for defining the two outlets that the Router requested above. As you can see our `ApplicationView` is extending the `Ember.View` class, and it is defining a `templateName` property that the view will use to draw its user interface. The `Notes.NotesView` view is responsible for drawing the left hand part of the UI including the list of available notes, as depicted in figure 1.5 above. As most other frameworks for creatig web applications, either client-side or server-side, Ember.js will automatically attach an id to each and every HTML element that it inserts into the DOM. These ids will start with the string “ember” appended with a unique number to ensure that no two elements share the same id. For some elements you might want to specify an explicit id instead. Ember.js allows you to do just that by specifying the views `elementId` property. `Notes.NotesView` specifies both an explicit `elementId`, while also specifying which template that it will use to render the view and which CSS class names to apply to the rendered HTML element.

There is nothing new added to the `Notes.SelectedNoteView`, as it simply specifies an explicit identifier, and which template it will use to render its view. Since none of our views specify a `tagName` attribute, our three views will be rendered as div-tags, which is the default tag in Ember.js views.

As each of our views specify a template, let’s have a look into the contents of these templates, shown below in listing 1.5.

Listing 1.5 The Notes Applications Templates

```
//** Templates **/
Ember.TEMPLATES['application'] = Ember.Handlebars.compile('' +
    '{{outlet}}{{render selectedNote}}'                                 #A
);

Ember.TEMPLATES['notes'] = Ember.Handlebars.compile('');                #B

Ember.TEMPLATES['selectedNote'] = Ember.Handlebars.compile('');         #C
```

#A: The application view specifies two outlets names notes and selectedNote
#B: The notesTemplate is empty at this point
#C: The selectedNoteTemplate is empty at this point

As you can see, two of our templates are empty, as we have not provided content for either of them yet. The `application` template, however, one outlet and one render expression. The contents of the template also reveal another important feature of Ember.js’ preferred template engine, Handlebars. Any dynamic expression of a template needs to be wrapped inside double curly braces. The very first parameter inside the expression tells Handlebars.js what type of expression it is, while the rest of the parameters serve as input to Handlebars.js. Here, our `application`, defines a default anonymous outlet expression and a render expression. The outlet expression enables Notes.NotesRoute to render the Notes.NotesView into it, while the render expression will use its own names router, controller and view.

Handlebars.js will be explored in more detail in chapter 4.

In order to be able to run the application there is a single line missing, the initialization of the entire Notes application, shown in listing 1.6

Listing 1.6 Initializing the Notes Application

```
Notes.initialize();                                                    #A
```

#A: Initializing the Notes Application

Initialization is simply done via the `Namespace.initialize()` function. This expression will tell Ember.js that everything needed for the application have loaded and that it is safe to initialize the application's router. At this point in time, Ember.js will go through the statements in the router and make sure that it initializes every view and every controller that is requested via the code inside each of the routers states. For our Notes application this means that Ember.js will, at this point in time, instantiate and wire up the following components:

- `Notes.ApplicationController` will be instantiated
- `Notes.ApplicationView` will be instantiated
- `Notes.NotesController` will be instantiated
- `Notes.SelectedNoteController` will be instantiated
- `Notes.ApplicationView` will be instantiated and added to `Ember.View.views` and its template compiled
- `Notes.NotesView` will be instantiated and added to `Ember.View.views` and its template compiled
- `Notes.SelectedNoteView` will be instantiated and added to `Ember.View.views` and its template compiled

If you load the index.html file in your browser at this point, you will be greeted with the view shown in figure 1.8 below.

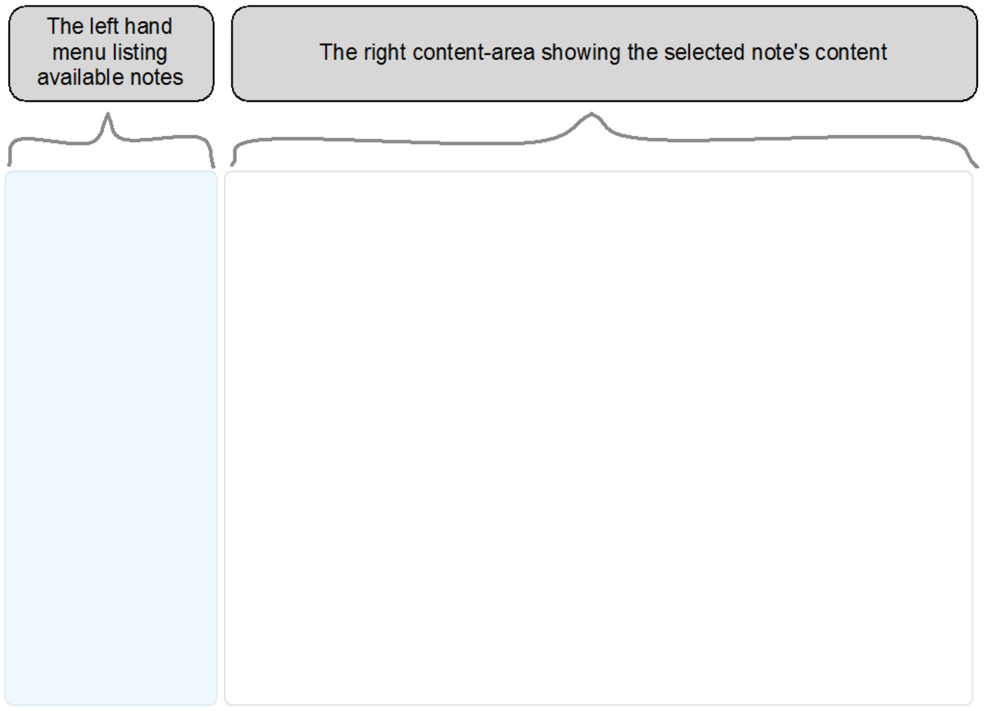

The left hand menu listing available notes
The right content-area showing the selected note's content

Figure 1.8 The Initial Notes Application

At this point, you might think that we went through quite an ordeal in order to get two empty rectangles onto the screen, but as you'll soon discover all of that hard work will pay off in dividends before we reach the end of this chapter.

1.4.2 Adding Notes to the Notes Application

By the end of this section, we will have implemented the second part of the application. You can view the complete source code as "app2.js" within 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.

What is a Notes application without the possibility to add new notes to it? So our next step is to be able to add some notes to our application. For this we will need a couple of things. First we need an input field that the user can type in the name of the note and a button that the user can click in order to create a new note. We will create one new view inside our Notes application, `Notes.TextField` to accept input from the user, while we will use the HTML button tag to create the button. We will also extend the template `notesTemplate` with some content and add a property `newNoteName` to the `Notes.NotesController`. The updated classes are shown in listing 1.7 below.

Listing 1.7 Adding an input field and a button to the view

```
Notes.NotesController = Ember.ArrayController.extend({
    content: [],
    newNoteName: null                                                   #A
});

Notes.TextField = Ember.TextField.extend(Ember.TargetActionSupport, {   #B
    insertNewline: function() {
        this.triggerAction();                                           #C
    }
});

Ember.TEMPLATES['notesTemplate'] = Ember.Handlebars.compile('' +        #D
    '{{view Notes.TextField target="controller" action="createNewNote"
classNames="input-small search-query mediumTopPadding"
valueBinding="controller.newNoteName"}}' +                              #E
    '<button class="btn" {{action createNewNote}}>New Note</button>'    #F
);
```

#A: A property newNoteName on the NotesController to hold the name of the new note
#B: Extending Ember.TextField in order to add an action on carriage return
#C: Calling triggerAction when carriage return is pressed inside the textfield
#D: The templates for the notesTemplate
#E: Creating a Notes.TextField with target, action and valueBinding
#F: Adding a button to allow using the mouse to add a new note

The new `Notes.TextField` class extends the standard `Ember.TextField` class in order to render its view as a text field. There are a couple of new concepts added to this class definition. The first this you might notice is the fact that we are applying the mixin `Ember.TargetActionSupport` to our `Notes.TextField` class. This will make our view able to trigger an action onto a specified target within our application via two properties, target and action. We are also overriding the `insertNewLine` function inside of which we are triggering an action. If you look at the first line of the template `notesTemplate` you will notice that we have defined an action `createNewNote` on the target `controller`. What this all means is that we will be triggering an action this view controller named `createNewNote` each time we hit carriage return while the text field is the active element on the website.

Note also that we have bound the value of our text field to the `controller.newNoteName` property. This means that while the user is typing into the text field that our controller's `newNoteName` property will keep up to date via the `valueBinding` property of the `Notes.TextField`. Ember.js is extra nice in this regard as any changes to the controller's `newNoteName` property will also automatically propagate all the way out to the view.

The final line inside our template defines a button with a handlebars `action` expression. This will ensure that whenever the button is clicked the `createNewNote` action will be fired. This way the user can choose whether to click the button or to hit enter while typing to add new notes to the system.

But there are a couple of things missing. First of all, we haven't created the createNewNote action yet and second we have not provided the application with enough functionality to actually display a list of the notes that we have created, so lets go ahead and add this. The updated code is shown below in listing 1.8.

Listing 1.8 Being Able To Add a New Note

```
Notes.NotesController = Ember.ArrayController.extend({
    content: [],
    newNoteName: null,

    createNewNote: function() {                                          #A
        var content = this.get('content');
        var newNoteName = this.get('newNoteName');

        content.pushObject(                                              #B
            Ember.Object.create({"name": newNoteName, "value": ""})
        );

        this.set('newNoteName', null);                                   #C
    }
});

Ember.TEMPLATES['notesTemplate'] = Ember.Handlebars.compile('' +
    '{{view Notes.TextField target="Notes.router" action="createNewNote"
classNames="input-small search-query mediumTopPadding"
valueBinding="controller.newNoteName"}}' +
    '<button class="btn" {{action createNewNote}}>New Note</button>' +
    '{{view Notes.NoteListView}}'                                        #D
);
```

At this point we have added an action to `Notes.NotesController` called `createNewNote` in order to create a new note. The `createNewNote` function starts out by getting the two variables it needs, the controllers content and the value of the text field, which it uses to create a new note and push it to the end of the content array. Before the function returns it makes sure to reset the text field, leaving it blank so that the application is ready to accept a new note from the user.

Finally, we have added a new view to the template `notesTemplate` that will list out the available notes. The only thing missing in order to take our application to the next phase is to define the `Notes.NoteListView`, shown below in listing 1.9.

Listing 1.9 Adding The Notes.NoteListView

```
Notes.NoteListView = Ember.View.extend({
    elementId: 'noteList',
    template: Ember.Handlebars.compile('' +                             #A
        '{{#each controller}}' +                                        #B
            '{{view Notes.NoteListItemView contentBinding="this"}}' +   #C
        '{{/each}}')                                                    #D
});

Notes.NoteListItemView = Ember.View.extend({
    template: Ember.Handlebars.compile('{{name}}'),
    classNames: ['pointer', 'noteListItem'],                            #E

    classNameBindings: "isSelected",                                    #F

    isSelected: function() {                                            #G
        return this.get('controller.selectedNote.name') ===
this.get('content.name');
    }.property('controller.selectedNote.name'),                         #H

    click: function() {                                                 #I
        this.get('controller').set('selectedNote', this.get('content'));
    }
});
```

#A: Using an inline template instead of a templateName
#B: The #each handlebars helper acts as an iterator over each of the items in the controllers content array
#C: Creating a new view for each of the notes
#D: Making sure to close the each-helper
#E: Adding CSS classes to the rendred DOM element
#F: Appending a CSS class is-selected when the computed property isSelected returns tru
#G: Defining computed property isSelected that will keep up to date with the currently selected note
#H: Keeping the returned value of isSelected updated wheneer controller.selectedNote.name changes
#I: Adding a click handler to change which note is selected

Starting with the `Notes.NoteListView` we are now using a different path in order to get our template compiled into the view. Rather than defining a `templateName` attribute and relying on having this template be available elsewhere in the application we are compiling the template directly into the template property. There are two reasons for this, first I tend to think of views that can be reused in different parts of the application or even in other Ember.js applications differently than views that are defined solely for this applications purpose. Component views like the one above, I tend to use an inline template, directly compiled as shown here. Second, I wanted to show that there are multiple ways of specifying the templates for the views.

The template inside `Notes.NoteListView` uses a new Handlebars.js expression, `#each`. The hash-symbol, #, tells Handlebars.js that this is a block expression and Handlebars.js expects you to end your block with a slash. In this template we are simply iterating over each of the controllers content array and rendering a `Notes.NoteListItemView` for each of the notes we have created in our Notes application. Note that we bind each `Notes.NotelistItemView's` content property with the current note via the `contentBinding="this"` property.

The `Notes.NoteListItemView` has a very simple template, as it only displays the name of its content object. On the next line we are appending the CSS classes `pointer` and `noteListItem` to rendered div in the same manner as we did above in listing 1.4. The next three statements, however, is where the dynamic part of the list is defined. Via the `classNameBinding`: "`isSelected`" expression we are telling our view that it will append the CSS class `is-selected` to this view if the `isSelected` property returns true, which it only does when the name of the selected note is the same as name property of this views content.

In addition we are defining this function to be a computed property via the `.property('controller.selectedNote.name')` function call. We will go through computed properties in detail in chapter 2, but for now think of this as a way to keep the `isSelected` property updated and re-calculated each time that the `controller.selectedNote.name` property changes. This is the mechanism that will ensure that the correct note will be highlighted in our Notes application.

The final line of the `Notes.NoteListItemView` class tell Ember.js to update the controllers `selectedNote` property to this views content whenever a new note is selected by the user via a mouse click.

Once a note is selected, we want to display a notepad where the user can enter text, but we only want to show this notepad while a note is selected. In order to achieve this we need to update the `selectedNoteTemplate`, shown below in listing 1.10.

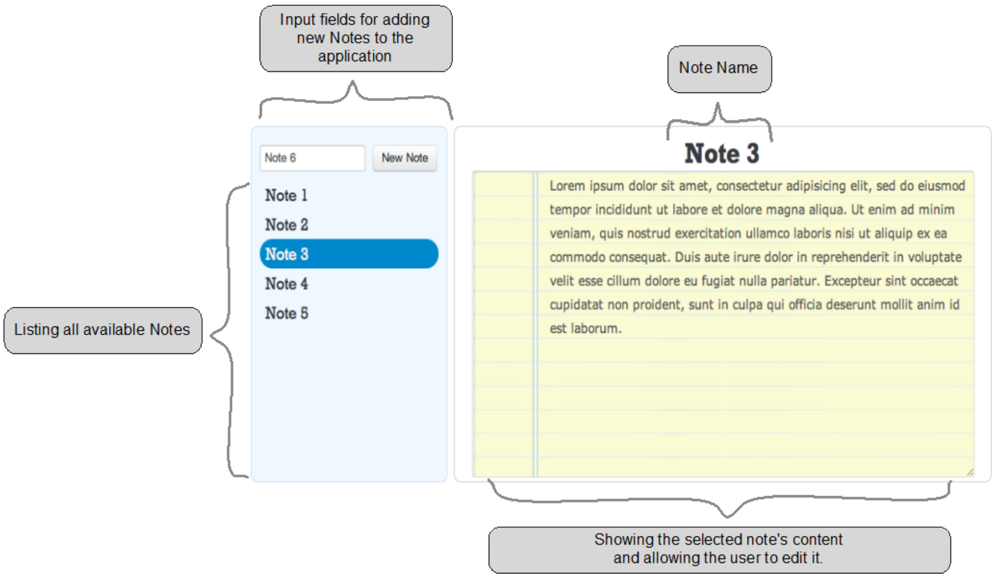
Listing 1.10 Showing the contents of the selected note

```
Ember.TEMPLATES['selectedNoteTemplate'] = Ember.Handlebars.compile('' +
    '{{#if controller.content}}' +                                     #A
        '<h1>{{name}}</h1>' +
        '{{view Ember.TextArea valueBinding="value"}}' +
    '{{/if}}'
);
```

#A: Using the if-expression to control what is displayed within the template

The template uses the Handlebars.js `#if` expression to toggle whether or not to display its contents or not. If the controller's content property is not `null` or `undefined`, the template will show the name of the note inside a `h1` header tag, while showing a text area where the user can type in the contents of the note.

If you reload the application now you are able to add new notes to the application, as well as selecting a note to highlight it in blue. The result is shown below in figure 1.9.


Input fields for adding new Notes to the application
Note Name
Note 6
New Note
Note 1
Note 2
Note 3
Note 4
Note 5
Listing all available Notes
Note 3
Showing the selected note's content and allowing the user to edit it.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 1.9 - Adding and highlighting notes

1.4.3 Deleting Notes

In this section we will have implemented the third and last part of the Notes application. You can view the complete source code for the application as "app3.js" in 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.

In order to delete notes we will be adding a delete button to the selected note in the list to the left. When the user click on this button the Notes application will present the user with a modal panel asking for confirmation before the note is deleted. Once the user have confirmed that the note really deserves to be deleted, the note will be removed from the `Notes.NotesController's` content property and the `selectedNote` property will be reset back to null. For this to work we need to add a modal panel to our application, which we will get from the Twitter Bootstrap framework. We also need to add a couple of new actions to the `Notes.NotesController`. The updated code shown below in listing 1.11 below.

Listing 1.11 Adding delete actions

```
Notes.NotesController = Ember.ArrayController.extend({
    content: [],
    newNoteName: null,

    createNewNote: function() { … },

    doDeleteNote: function() {                                          #A
        $("#confirmDeleteConfirmDialog").modal({show: true});
    },

    doConfirmDelete: function() {                                       #B
        var selectedNote = this.get('selectedNote');
        if (selectedNote) {                                             #C
            this.get('content').removeObject(selectedNote);             #D
            this.set('selectedNote', null);                             #E
        }
        $("#confirmDeleteConfirmDialog").modal('hide');
    },

    doCancelDelete: function() {                                        #F
        $("#confirmDeleteConfirmDialog").modal('hide');
    },

    deleteSelectedNote: function() {
        var selectedNote = this.get('selectedNote');
        if (selectedNote) {
            this.get('content').removeObject(selectedNote);
            this.set('selectedNote', null);
        }
    }
});
```

#A: Adding an action that will fire when the user clicks the delete button
#B: Adding an action that will fire when the user confirms the deletion
#C: Checking to see if there is a selected note
#D: Removing the object using the removeObject function
#E: Resetting which note is selected by setting selectedNote to null
#F: Adding a cancel action in case the user cancels the deletion

Here, we have added three new actions to the controller. The `doDeleteNote` action will be called from when the user clicked on the delete button in the list and will make sure that the modal panel is displayed to the user. Meanwhile the `doConfirmDelete` and `doCancelDelete` actions will be triggered from the modal panel, both hiding the modal panel. The `doConfirmDelete` action will perform the actual delete functionality.

The `deleteSelectedNote` function simply checks to see if there is a `selectedNote`, and if there is it will remove that object from the controllers content array and reset the `selectedNote` back to `null`.

Next we need to add the delete button to `Notes.NoteListItemView`'s template, shown below in listing 1.12.

Listing 1.12 Adding the delete button

```
Notes.NoteListItemView = Ember.View.extend({
    template: Ember.Handlebars.compile('' +
        '{{name}}' +
        '{{#if view.isSelected}}' +                                     #A
            '<button {{action doDeleteNote}} class="btn btn-mini floatRight
btn-danger smallMarginBottom">Delete</button>' +
        '{{/if}}'),
});
```

#A: Only displaying the delete button if this note is selected

In the template we will again rely in the `isSelected` property of the view. If the view is the selected view we will display a delete button, styled via Twitter Bootstrap's `btn` and `btn-mini` CSS classes. This button will call the `doDeleteNote` action on `Notes.router`.

Before permanently deleting the note from the system, we will display the modal panel so that the user can confirm that this action is really wanted. Show below in listing 1.13.

Listing 1.13 Defining the modal panel

```
Ember.TEMPLATES['notes'] = Ember.Handlebars.compile('' +
    '{{view Notes.TextField target="controller" action="createNewNote"
classNames="input-small search-query mediumTopPadding"
valueBinding="controller.newNoteName"}}' +
    '<button class="btn" {{action createNewNote}}>New Note</button>' +
    '{{view Notes.NoteListView}}' +

    '{{view Notes.ConfirmDialogView ' +                                 #A
        'elementId="confirmDeleteConfirmDialog" ' +
        'okAction="doConfirmDelete" ' +
        'cancelAction="doCancelDelete" ' +
        'target="controller" ' +
        'header="Delete selected note?" ' +
        'message="Are you sure you want to delete the selected Note? This
action cannot be be undone!"' +
    '}}'
);

Notes.ConfirmDialogView = Ember.View.extend({                           #B
    templateName: 'confirmDialog',                                      #C
    classNames: ['modal', 'hide'],

    cancelButtonLabel: 'Cancel',
    cancelAction: null,
    okButtonLabel: "OK",
    okAction: null,
    header: null,
    message: null,
    target: null
});

Ember.TEMPLATES['confirmDialog'] = Ember.Handlebars.compile(
    '<div class="modal-header centerAlign">' +
        '<button type="button" class="close" data-dismiss="modal"
class="floatRight">×</button>' +
        '<h1 class="centerAlign">{{view.header}}</h1>' +
    '</div>' +
    '<div class="modal-body">' +
        '{{view.message}}' +
    '</div>' +
    '<div class="modal-footer">' +
        {{#if view.cancelAction}}' +
            '{{view Notes.BootstrapButton ' +                   #D
                'contentBinding="view.cancelButtonLabel" ' +
                'actionBinding="view.cancelAction" ' +
                'targetBinding="view.target"}}' +
        '{{/if}}' +
        '{{#if view.okAction}}' +
            '{{view Notes.BootstrapButton ' +                   #E
                'contentBinding="view.okButtonLabel" ' +
                'actionBinding="view.okAction" ' +
                'targetBinding="view.target"}}' +
        '{{/if}}' +
    '</div>'
);
```

#A: Extending the notes template with a modal panel
#B: The definition of the ModalPanelView
#C: The modal panel uses an external template
#D and #E: Using the Notes.BootstrapButton to display Twitter Bootstrap styled buttons with custom actions attached.

The first thing we are doing is to extend the `notes` template with a new view, the `Notes.ConfirmDialogView`. This view takes in an `elementId`, which we are using to refer to it in order to show and hide the modal panel, as well as properties to initialize the modal panel with the text that it will display and the actions that it will call when the user hits the Cancel and OK buttons. This view uses an external template, for the simple reason that the template stretches over many lines.

The `confirmDialog` template introduces a new component, the `Notes.BootstrapButton`, which we will have a closer look at later on. What is interesting about this modal panel is that it is extensible in regards to both the text and information it provides, but also in regards to the actions that it will perform and which buttons it will provide through its user interface. The `Notes.BootstrapButton` takes care of handling calling the correct action when the user clicks on a button, shown below in listing 1.14.

Listing 1.14 The Notes.Bootstrap Button

```
Notes.BootstrapButton = Ember.View.extend(Ember.TargetActionSupport, {   #A
    tagName: 'button',
    classNames: ['button'],
    disabled: false,

    click: function() {
        if (!this.get('disabled')) {
            this.triggerAction();                                        #B
        }
    },

    template: Ember.Handlebars.compile('{{#if view.iconName}}<i {{bindAttr
class="view.iconName"}}></i>{{/if}}{{view.content}}')
});
```

#A: Using the TargetActionSupport
#B: Triggering an action when the button is clicked

There isn't a lot going on in the `Notes.BootstrapButtonView`, and it has a similar structure as our `Notes.TextField` view has. They both use the `Ember.TargetActionSupport` mixin to be able to accept a target and an action, and they both trigger an action when the correct event is caught from the user, a click in this scenario. Browse back to listing 1.7 to peruse the code and the explanation for the `Notes.TextField`.

1.4.4 Ensuring that there's no duplicate notes

The way we have structured our `Notes.NotesController` means that notes that share the same name will be highlighted together and worse only the top-most matching note will be deleted, even if that is not the selected note. In order to fix this, lets add some code to the `createNewNote` function to ensure that we won't get more than one note with the same name.

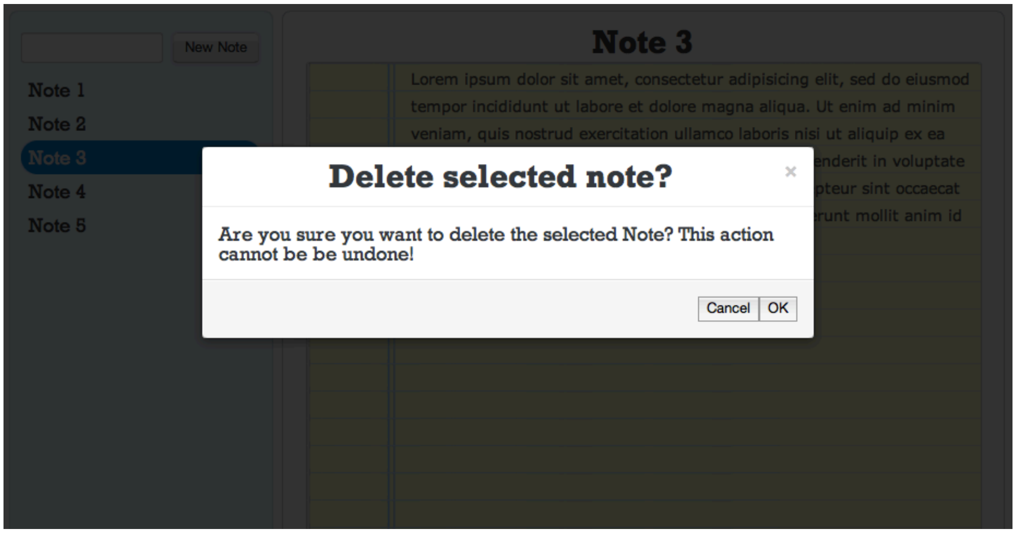
Listing 1.15 The updated createNewNote

```
Notes.NotesController = Ember.ArrayController.extend({
    content: [],
    newNoteName: null,

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

        content.forEach(function(note) {                                 #A
            if (newNoteName === note.get('name')) {
                unique = false; return;                                  #B
            }
        });
        if (unique) {                                                    #C
            content.pushObject(
                Ember.Object.create({"name": newNoteName, "value": ""})
            );
            this.set('newNoteName', null);
        } else {
            alert('Note must have a unique name');                       #D
        }
    },

    deleteSelectedNote: function() { ... }
});
```

#A: Iterating over the content array
#B: Setting the flag unique to false if a note with the same name exists
#C: Creates a new note if the new note's name us unique
#D: Displaying an alert to the user if the new note's name is not unique

This code is rather naïve in its approach. It loops over each note in the content array and flags a Boolean variable named `unique` as false if it finds a note with the same name. Later, it will only add a new note if the `unique` flag is true, and it will alert the user if the `unique` flag is set to false.

Now that out Notes application is completed, it should look like figure 1.10 if you reload the application in your web browser.



Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
enderit in voluptate
pteur sint occaecat
runt mollit anim id

Delete selected note?

Are you sure you want to delete the selected Note? This action cannot be be undone!

Figure 1.10 - The completed Notes application with Delete Modal panel

1.5 Summary

Throughout this chapter I have tried to give an overview of the building blocks that Ember.js is built up from, and the most important concepts of an Ember.js application. Throughout this chapter I hope that you have received a better understanding of the Ember.js framework and, why it exists and where it will be most applicable to you as a developer.

As an introductory chapter into Ember.js I have tried to guide your 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 features as possible of Ember.js while still trying not to be overwhelming.

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

We have introduces a lot of the core features of Ember.js in this chapter very briefly without a thorough description of them. The next chapter will reuse and extend the code we have written in this chapter slightly, in order to thoroughly explain the core features that Ember.js provides.